

For better learning follow these:

- Be attentive
- Mute mic and off the video to save bandwidth unless asked to do otherwise
- Don't use chat box to communicate with each other
- When you have doubt raise hand and clear doubts when asked to do so.
- Attendance will be called at random during the session.
- Take notes and participate in the learning process.
- HAVE A GOOD LEARNING EXPERIENCE

Unit 1

Database and Database users

- ❖ Introduction
- ❖ View of data
- ❖ Advantages
- ❖ Actors and architecture

History of Database Systems(not in syllabus)

- 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 begins Ingres prototype
 - High-performance (for the era) transaction processing

History (cont.)

1980s:

- Research relational prototypes evolve into commercial systems
 - SQL becomes industrial standard
 - Parallel and distributed database systems
 - Object-oriented database systems
- 1990s:
 - Large decision support and data-mining applications
 - Large multi-terabyte data warehouses
 - Emergence of Web commerce
- Early 2000s:
 - XML and XQuery standards
 - Automated database administration
- Later 2000s:
 - Giant data storage systems
 - Google BigTable, Yahoo PNuts, Amazon, ..

Database Management System (DBMS)

- DBMS contains information about a particular enterprise
 - Collection of interrelated data
 - Set of programs to access the data
 - An environment that is both *convenient* and *efficient* to use

- Database Applications:
 - Banking: 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

University Database Example

- Application program examples
 - Add new students, instructors, and courses
 - Register students for courses,
 - Assign grades to students, compute grade point averages (GPA) and generate transcripts
- In the early days, database applications were built directly on top of file systems

Drawbacks of using file systems to store data

- **Data redundancy and inconsistency**
 - Multiple file formats, duplication of information in different files
- **Difficulty in accessing data**
 - Need to write a new program to carry out each new task
- **Data isolation** — data is scattered 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

Drawbacks of using file systems to store data (Cont.)

- **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
 - Example: 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 offer solutions to all the above problems

Simplified database system environment

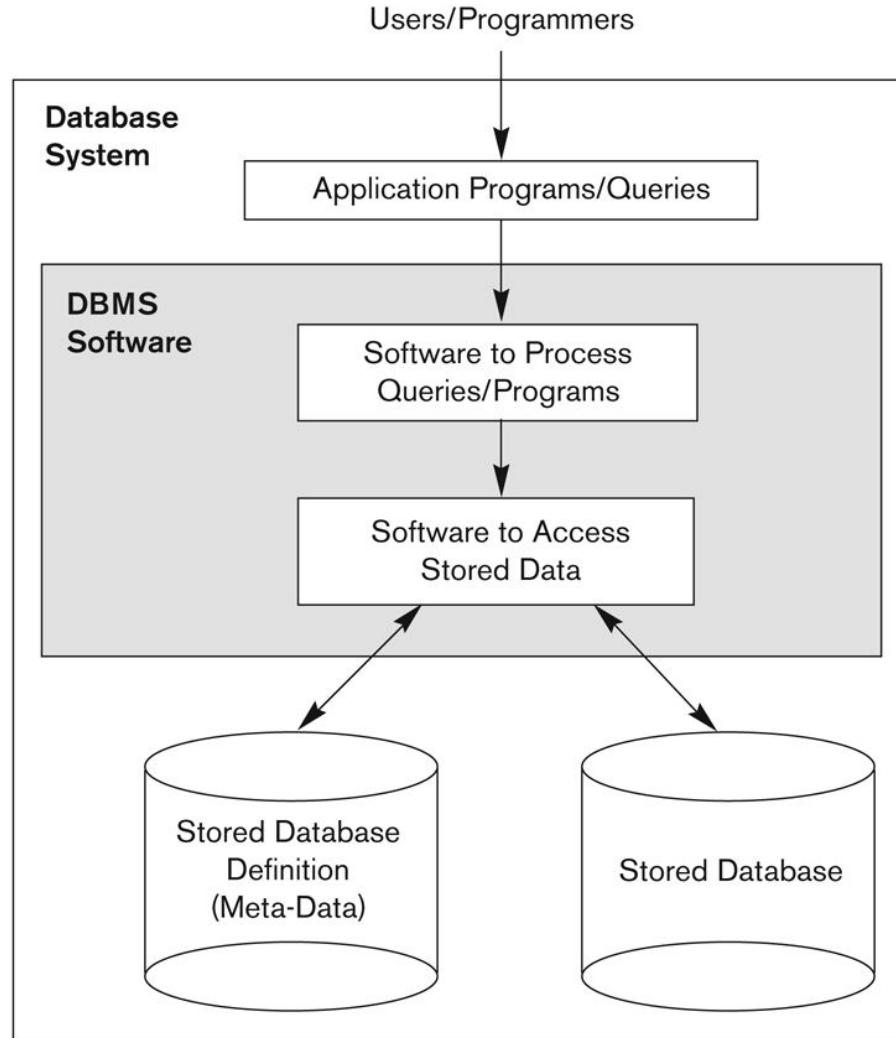


Figure 1.1
A simplified database system environment.

View of the data

- DBS is a collection of interrelated data and set of programs that allows users to access and modify these data.
- DBS provides abstract view, i.e., hides how data is stored and maintained.

Levels of Abstraction

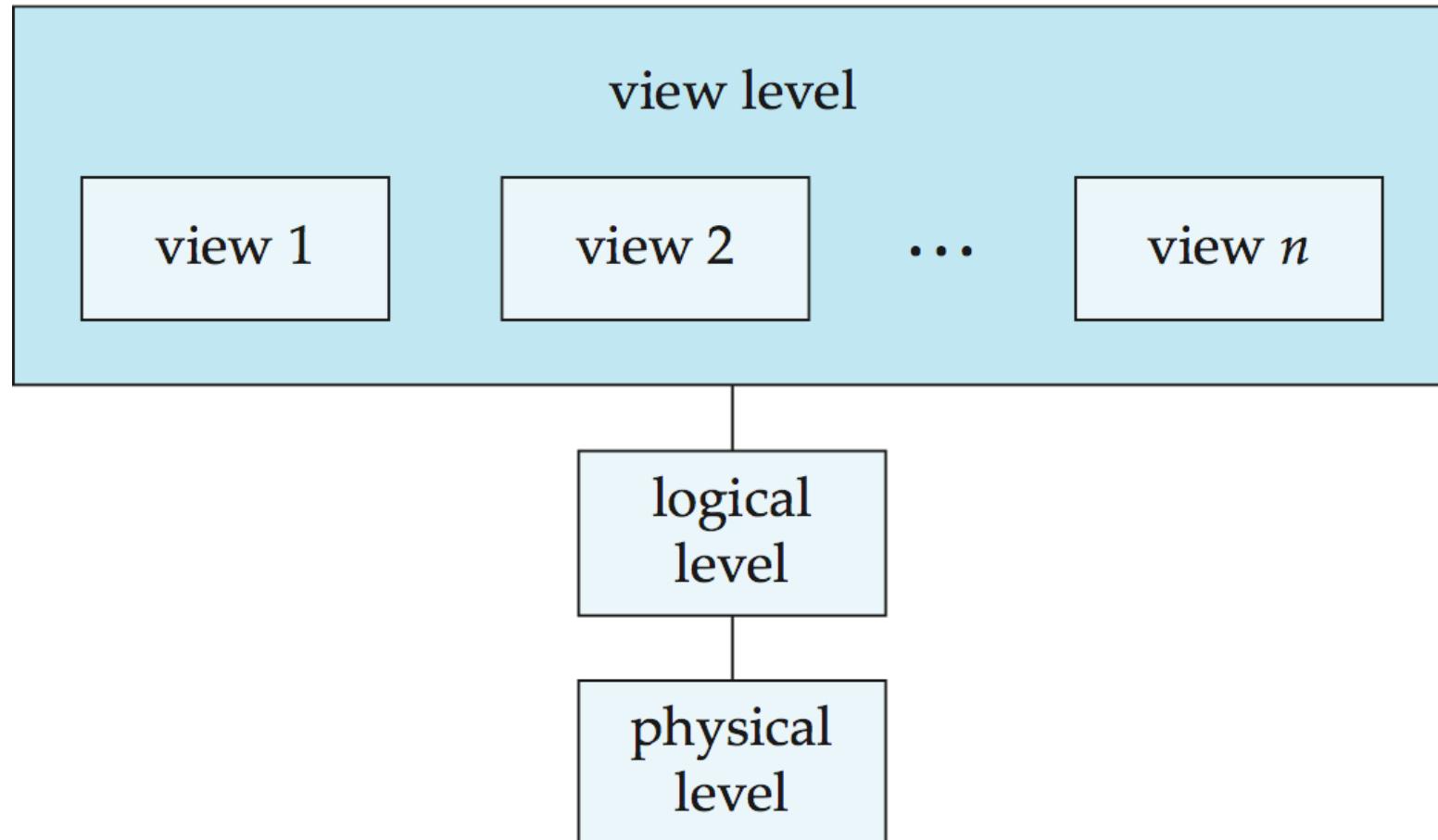
- **Physical level:** describes how a data (e.g., customer detail) is stored and various data structures used.
- **Logical level:** describes what data are stored in database, and the relationships among the data.

```
type instructor = record
    ID : string;
    name : string;
    dept_name : string;
    salary : integer;
end;
```

- **View level:** application programs hide details of data types. Views can also hide information (such as an employee's salary) for security purposes.

View of Data

An architecture for a database system



Example of a simple Relational database

COURSE

| Course_name | Course_number | Credit_hours | Department |
|---------------------------|---------------|--------------|------------|
| Intro to Computer Science | CS1310 | 4 | CS |
| Data Structures | CS3320 | 4 | CS |
| Discrete Mathematics | MATH2410 | 3 | MATH |
| Database | CS3380 | 3 | CS |

SECTION

| Section_identifier | Course_number | Semester | Year | Instructor |
|--------------------|---------------|----------|------|------------|
| 85 | MATH2410 | Fall | 04 | King |
| 92 | CS1310 | Fall | 04 | Anderson |
| 102 | CS3320 | Spring | 05 | Knuth |
| 112 | MATH2410 | Fall | 05 | Chang |
| 119 | CS1310 | Fall | 05 | Anderson |
| 135 | CS3380 | Fall | 05 | Stone |

GRADE_REPORT

| Student_number | Section_identifier | Grade |
|----------------|--------------------|-------|
| 17 | 112 | B |
| 17 | 119 | C |
| 8 | 85 | A |
| 8 | 92 | A |
| 8 | 102 | B |
| 8 | 135 | A |

PREREQUISITE

| Course_number | Prerequisite_number |
|---------------|---------------------|
| CS3380 | CS3320 |
| CS3380 | MATH2410 |
| CS3320 | CS1310 |

Figure 1.2

A database that stores student and course information.

Database languages:

- Data definition language(DDL)
- Data manipulation language(DML)

Data Definition Language (DDL)

- Specification notation for defining the database schema

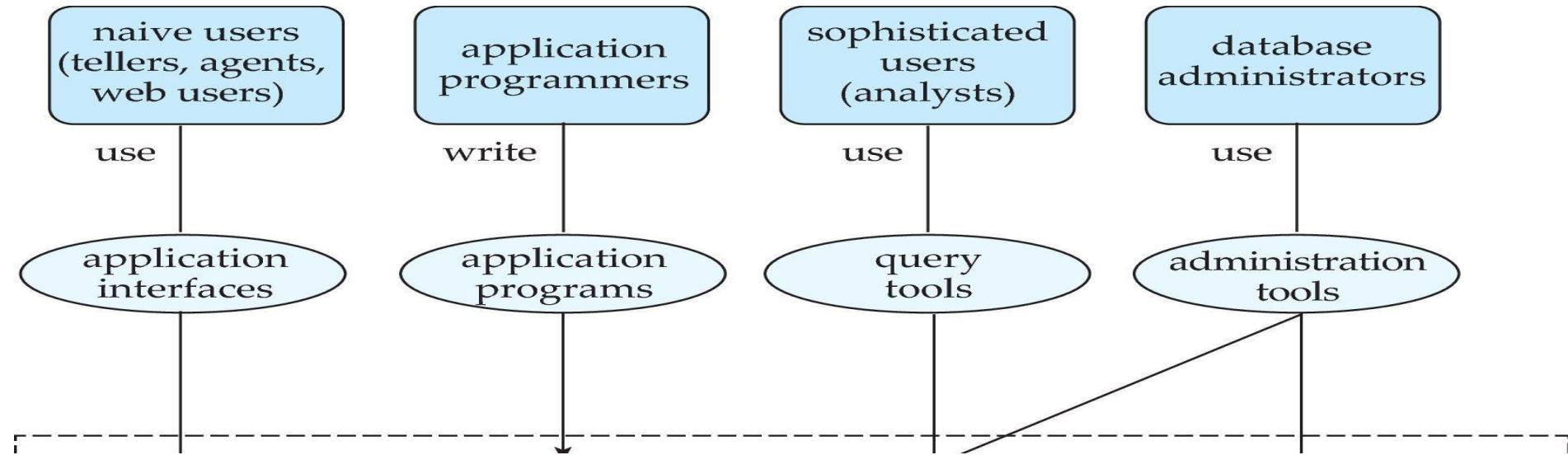
Example: `create table instructor (`
 `ID char(5),`
 `name varchar(20),`
 `dept_name varchar(20),`
 `salary numeric(8,2))`

- 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)
 - Referential integrity (**references** constraint in SQL)
 - e.g. `dept_name` value in any *instructor* tuple must appear in *department* relation
 - Authorization

Data Manipulation Language (DML)

- Language for accessing and manipulating the data organized by the appropriate data model
 - DML also known as query language
- Two classes of languages
 - **Procedural** – user specifies what data is required and how to get those data
 - **Declarative (nonprocedural)** – user specifies what data is required without specifying how to get those data
- SQL is the most widely used query language

Database Users and Administrators



Database

Database Users

- Users may be divided into
 - Those who actually use and control the **database** content, and those who design, develop and maintain database applications (called “**Actors on the Scene**”), and
 - Those who design and develop the **DBMS** software and related tools, and the computer systems operators (called “**Workers Behind the Scene**”).

Database Users

- **Actors on the scene**

1. Database administrators:

- Responsible for **authorizing access** to the database, for coordinating and monitoring its use, acquiring software and hardware resources, controlling its use and **monitoring efficiency** of operations.

2. Database Designers:

- Responsible to define the content, the structure, the constraints, and functions or transactions against the database. They must communicate with the end-users and understand their needs.

Actors on the scene (continued)

3. End-users: They use the data for queries, reports and some of them update the database content. End-users can be categorized into:

- **Casual**: access database **occasionally** when needed
- **Naïve** or Parametric: they make up a large section of the end-user population.
 - They use previously well-defined functions in the form of “canned transactions” against the database.
 - Examples are **bank-tellers or reservation clerks** who do this activity for an entire shift of operations.

Categories of End-users (continued)

- **Sophisticated:**

- These include business analysts, scientists, engineers, others thoroughly familiar with the system capabilities.
- Many use tools in the form of software packages that work closely with the stored database.

- **Stand-alone:**

- Maintain **personal databases** using ready-to-use packaged applications.
- An example is a **tax program** user that creates its own internal database.
- Another example is a user that maintains an **address book**

Actors..

4. System analyst and application programmers

- system analyst determine the requirements and develop specification for these
- Application programmers implement these specifications

Workers behind the Scene

- 1. DBMS system designers and implementers** are persons who design and implement the **DBMS modules** and **interfaces** as a software package.
- 2. Tool developers** include persons who design and implement **tools**—the software packages that facilitate database system design and use, and help improve performance.
- 3. Operators and maintenance personnel** (system administration personnel) are responsible for the actual running and maintenance of the hardware and software environment for the database system.

Advantages of Using the Database Approach

1. Controlling redundancy

Traditional file system has following issues;

1. Multiple updates
2. Storage wastage
3. Inconsistency

Solution: data normalization and controlled redundancy

2. Restricting unauthorized access to data- security and authorization subsystem is provided
3. Providing persistent storage for program objects: object oriented DB

4. Providing Storage Structures (e.g. indexes) for efficient Query Processing

- Indexes could be provided
 - Buffering or caching modules are available
 - Query processing and optimizing module

Advantages of Using the Database Approach (continued)

5. Providing backup and recovery services.
 - on failure, DB has to be restored to state of consistency or resumed
6. Providing multiple interfaces to different classes of users.
 - forms, command, codes, menu driven , natural language interface
7. Representing complex relationships among data.
 - should have capability to represent variety of relationships

8. Enforcing integrity constraints on the database.

- Datatype should be provided
- Referential integrity
- Uniqueness of data

9. Permitting inferences and action using rules

-infer new information through rules(codes)

- Triggers
- Stored procedures

- 10. other benefits:
 - Enforcing standards
 - Reducing application development time
 - Flexibility
 - Up to date information
 - Economies of scale

Database Architecture

The architecture of a database systems is greatly influenced by the underlying computer system on which the database is running:

- Centralized
- Client-server
- Parallel (multi-processor)
- Distributed

Database System Internals

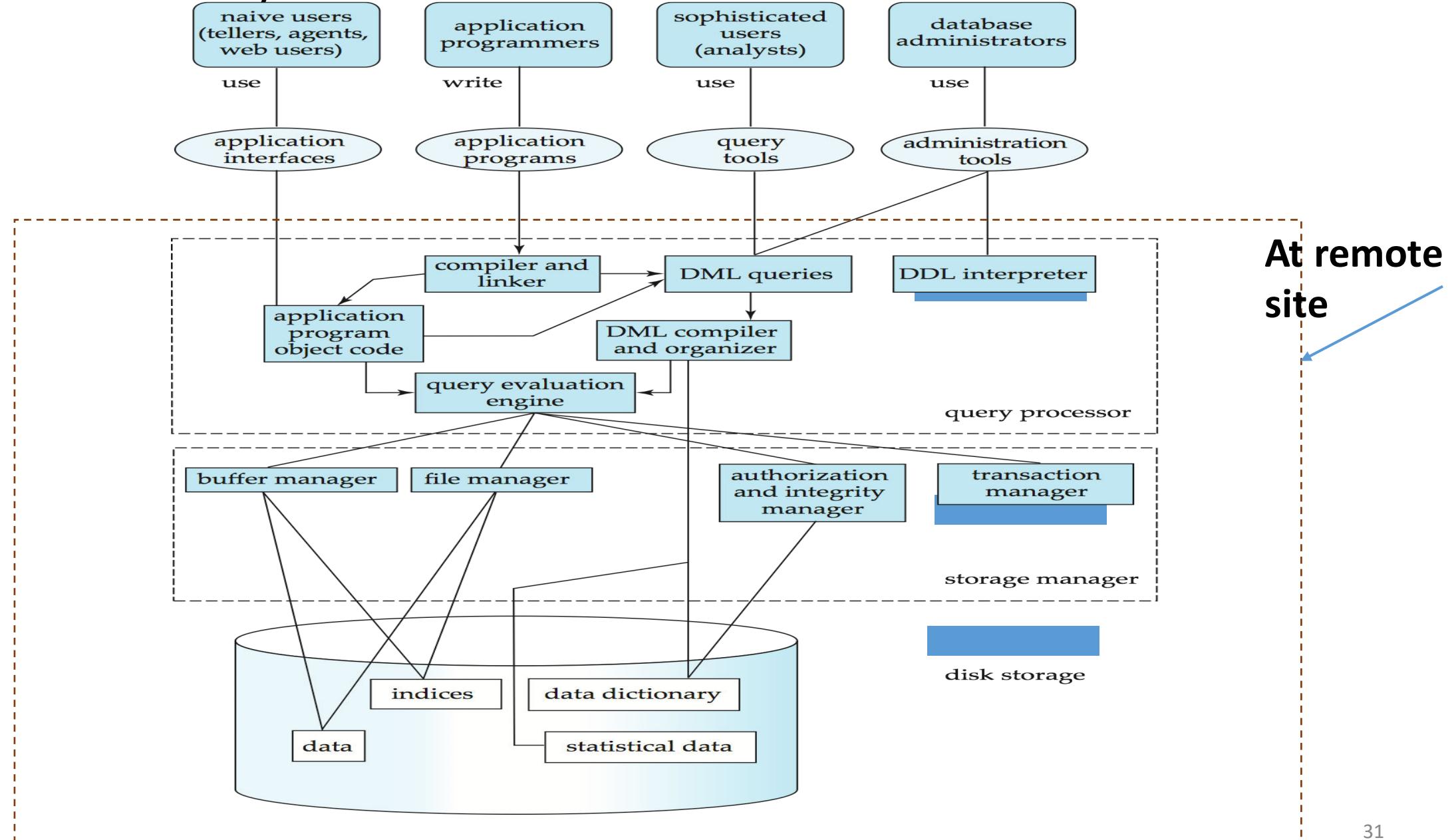
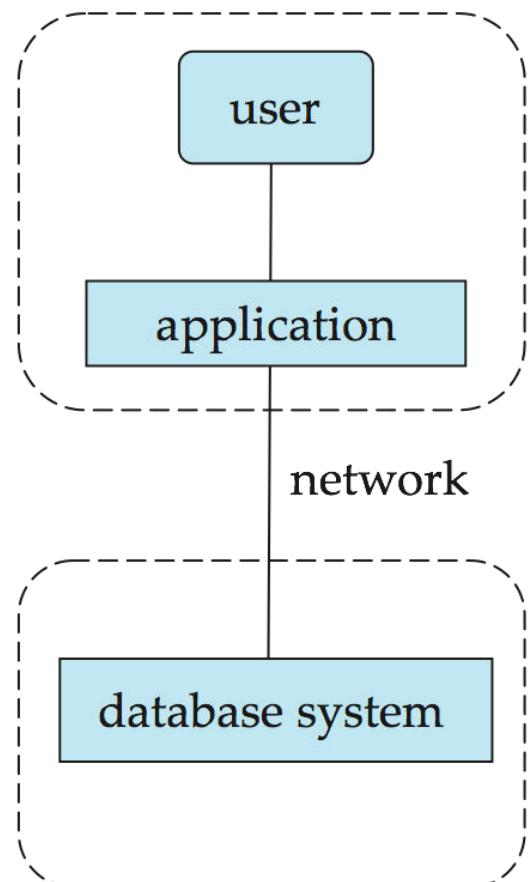
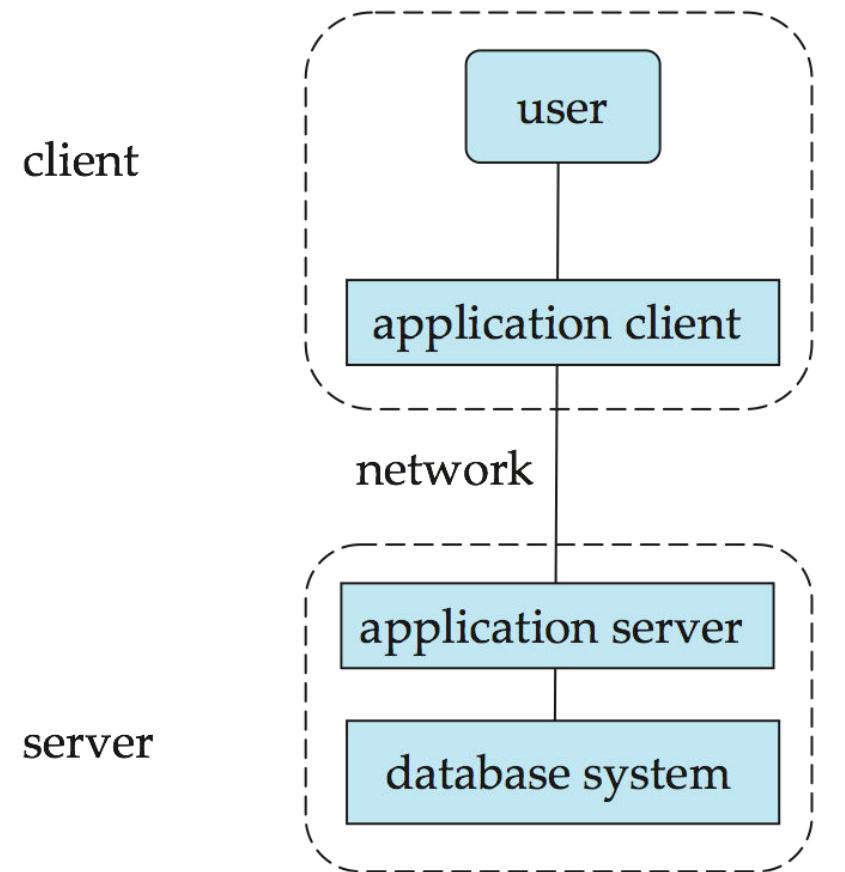


Figure 1.06



(a) Two-tier architecture



(b) Three-tier architecture

- Database applications are partitioned into 2 or 3 parts
- In a two tier architecture, the application resides at the client machine where it invokes database system functionality at the server machine through queries.
- API standards such as ODBC, JDBC are used for interaction between the client and the server.

- In a 3 tier architecture, the client machine acts as a front end.
- The client machine communicates with the application server, usually through forms
- The application server in turn communicates with a database system to access data
- All logic is embedded in the application server.



Das Bild kann zurzeit nicht angezeigt werden.

Chapter 2: Introduction to Relational Model

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

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



- Structure
- Schema
- Query language
- Relational algebra



Basic Unit of the Relational Model

Example of a Relation/Table:

| <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 |

attributes
(or columns)

tuples
(or rows)



| course_id | title | | dept_name | credits |
|-----------|----------------------------|--|------------|---------|
| BIO-101 | Intro. to Biology | | Biology | 4 |
| BIO-301 | Genetics | | Biology | 4 |
| BIO-399 | Computational Biology | | Biology | 3 |
| CS-101 | Intro. to Computer Science | | Comp. Sci. | 4 |
| CS-190 | Game Design | | Comp. Sci. | 4 |
| CS-315 | Robotics | | Comp. Sci. | 3 |
| CS-319 | Image Processing | | Comp. Sci. | 3 |
| CS-347 | Database System Concepts | | Comp. Sci. | 3 |
| EE-181 | Intro. to Digital Systems | | Elec. Eng. | 3 |
| FIN-201 | Investment Banking | | Finance | 3 |
| HIS-351 | World History | | History | 3 |
| MU-199 | Music Video Production | | Music | 3 |
| PHY-101 | Physical Principles | | Physics | 4 |

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

| course_id | prereq_id |
|-----------|-----------|
| BIO-301 | BIO-101 |
| BIO-399 | BIO-101 |
| CS-190 | CS-101 |
| CS-315 | CS-101 |
| CS-319 | CS-101 |
| CS-347 | CS-101 |
| EE-181 | PHY-101 |

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

| ID | course_id | sec_id | semester | year |
|-------|-----------|--------|----------|------|
| 10101 | CS-101 | 1 | Fall | 2009 |
| 10101 | CS-315 | 1 | Spring | 2010 |
| 10101 | CS-347 | 1 | Fall | 2009 |
| 12121 | FIN-201 | 1 | Spring | 2010 |
| 15151 | MU-199 | 1 | Spring | 2010 |
| 22222 | PHY-101 | 1 | Fall | 2009 |
| 32343 | HIS-351 | 1 | Spring | 2010 |
| 45565 | CS-101 | 1 | Spring | 2010 |
| 45565 | CS-319 | 1 | Spring | 2010 |
| 76766 | BIO-101 | 1 | Summer | 2009 |
| 76766 | BIO-301 | 1 | Summer | 2010 |
| 83821 | CS-190 | 1 | Spring | 2009 |
| 83821 | CS-190 | 2 | Spring | 2009 |
| 83821 | CS-319 | 2 | Spring | 2010 |
| 98345 | EE-181 | 1 | Spring | 2009 |



Table/Relation

- In general, a **row** in a table represents a relationship among a set of values.
- In the relational model the term **relation** is used to refer to a **table**, while the term **tuple** is used to refer to a **row**.
- Similarly, the term **attribute** refers to a **column of a table**.
- The term **relation instance** is used to refer to a specific instance of a relation, i.e., containing a specific set of rows.



Attribute Types

- The set of permitted values for each attribute is called the **domain** of the attribute.
- Attribute values are (normally) required to be **atomic**; that is, indivisible
- The **null** value is a special value that signifies that the value is unknown or does not exist



Relation Schema and Instance

- The term **relation schema** is attribute set and their domains
 - whereas the **relation instance** corresponds to the values which the variable takes.
-
- A *relation schema syntax*: $R = (A_1 :D_1, A_2 :D_2, \dots, A_n :D_n)$
- Example:
- instructor = (ID: int, name :String, dept_name : String, salary: float)***

A **database** consists of multiple relations



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

- A database consists of multiple relations
- Information about an enterprise is broken up into parts

instructor

student

advisor

- **Bad design: Putting everything in one table**

univ (instructor-ID, name, dept_name, salary, student_Id, ...)

results in

- repetition of information (e.g., two students have the same instructor)
 - the need for null values (e.g., represent an student with no advisor)
- Normalization theory (Chapter 7) deals with how to design “good” relational schemas



Keys

- The values of the attribute values of a tuple must be such that they can uniquely identify the tuple.
- A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation.
- 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*.



| <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 |



- Superkey K is a **candidate key** if K is minimal

Example: $\{ID\}$ is a candidate key for *Instructor*

- It is possible that several distinct sets of attributes could serve as a **candidate key**.
- Suppose that a combination of name and dept_name is sufficient to distinguish among members of the instructor relation. Then, both $\{ID\}$ and $\{\text{name, dept_name}\}$ are **candidate keys**.



Candidate and Primary Key

- The attributes ID and name together can distinguish instructor tuples, their combination, **{ID, name}** does not form a candidate key, since the attribute {ID} alone is a candidate key.
- One of the candidate keys is selected to be the **primary key**.
- The **primary key** should be chosen such that its attribute values are **never, or very rarely changed and it should not have null value**.

- It is customary to list the primary key attributes of a relation schema before the other attributes.
- Primary key attributes are also **underlined**



Foreign Key

- A relation, say $r1$, may include the **primary key** of another **relation**, say $r2$. This attribute is called a **foreign key** from $r1$, referencing $r2$.
- **Referencing relation – $r1$**
- **Referenced relation – $r2$**
- For example, the attribute *dept_name* in *instructor* is a foreign key from *instructor*, referencing *department*.



DEPARTMENT

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

INSTRUCTOR

| ID | name | dept_name | salary |
|--------|------------|------------|--------|
| 222222 | 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 |

Referencing relation – r1- INSTRUCTOR
Referenced relation – r2- DEPARTMENT

Example of a Database (with a Conceptual Data Model)

- **Mini-world for the example:**
 - Part of a UNIVERSITY environment.
- **Some mini-world entities:**
 - STUDENTs
 - COURSEs
 - SECTIONs (of COURSEs)
 - (academic) DEPARTMENTs
 - INSTRUCTORs

Example of a Database (with a Conceptual Data Model)

- Some mini-world *relationships*:
 - SECTIONs are *of specific* COURSEs
 - STUDENTs *take* SECTIONs
 - COURSEs *have prerequisite* COURSEs
 - INSTRUCTORs *teach* SECTIONs
 - COURSEs *are offered by* DEPARTMENTs
 - STUDENTs *major in* DEPARTMENTs
- Note: The above entities and relationships are typically expressed in a conceptual data model, such as the ENTITY-RELATIONSHIP data model (see Chapters 3, 4)

Example of a simple database

COURSE

| Course_name | Course_number | Credit_hours | Department |
|---------------------------|---------------|--------------|------------|
| Intro to Computer Science | CS1310 | 4 | CS |
| Data Structures | CS3320 | 4 | CS |
| Discrete Mathematics | MATH2410 | 3 | MATH |
| Database | CS3380 | 3 | CS |

SECTION

| Section_identifier | Course_number | Semester | Year | Instructor |
|--------------------|---------------|----------|------|------------|
| 85 | MATH2410 | Fall | 04 | King |
| 92 | CS1310 | Fall | 04 | Anderson |
| 102 | CS3320 | Spring | 05 | Knuth |
| 112 | MATH2410 | Fall | 05 | Chang |
| 119 | CS1310 | Fall | 05 | Anderson |
| 135 | CS3380 | Fall | 05 | Stone |

GRADE_REPORT

| Student_number | Section_identifier | Grade |
|----------------|--------------------|-------|
| 17 | 112 | B |
| 17 | 119 | C |
| 8 | 85 | A |
| 8 | 92 | A |
| 8 | 102 | B |
| 8 | 135 | A |

PREREQUISITE

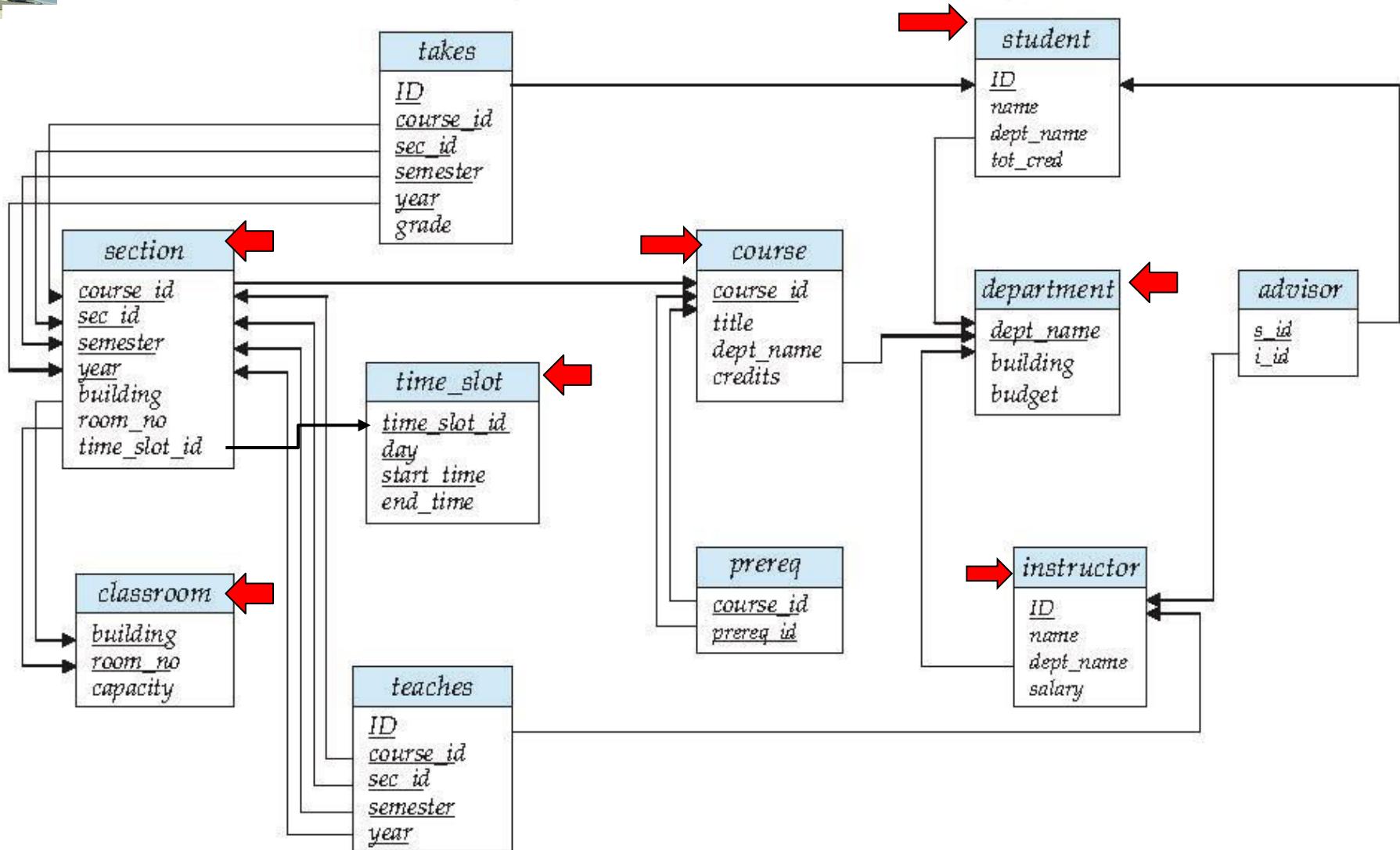
| Course_number | Prerequisite_number |
|---------------|---------------------|
| CS3380 | CS3320 |
| CS3380 | MATH2410 |
| CS3320 | CS1310 |

Figure 1.2

A database that stores student and course information.



Schema Diagram for University Database





Schema

- *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(SID, name, dept_name, tot_cred)*
- *takes(SID, course_id, sec_id, semester, year, grade)*
- *advisor(sID, iID)*
- *Time_slot(time_slot_id, day, start_time, end_time)*
- *prereq(course_id, prereq_id)*



■ TEACHES

| <i>ID</i> | <i>course_id</i> | <i>sec_id</i> | <i>semester</i> | <i>year</i> |
|-----------|------------------|---------------|-----------------|-------------|
| 10101 | CS-101 | 1 | Fall | 2009 |
| 10101 | CS-315 | 1 | Spring | 2010 |
| 10101 | CS-347 | 1 | Fall | 2009 |
| 12121 | FIN-201 | 1 | Spring | 2010 |
| 15151 | MU-199 | 1 | Spring | 2010 |
| 22222 | PHY-101 | 1 | Fall | 2009 |
| 32343 | HIS-351 | 1 | Spring | 2010 |
| 45565 | CS-101 | 1 | Spring | 2010 |
| 45565 | CS-319 | 1 | Spring | 2010 |
| 76766 | BIO-101 | 1 | Summer | 2009 |
| 76766 | BIO-301 | 1 | Summer | 2010 |
| 83821 | CS-190 | 1 | Spring | 2009 |
| 83821 | CS-190 | 2 | Spring | 2009 |
| 83821 | CS-319 | 2 | Spring | 2010 |
| 98345 | EE-181 | 1 | Spring | 2009 |



Relational Query Languages

- A **query language** is a language in which a **user** requests information from the database.
- Categorized as procedural or non procedural

- **Procedural:** The user instructs the system to perform a **sequence of operations** on the database to compute the desired result.
- Nonprocedural: user describes the desired information without giving a specific procedure for obtaining that information.



- In a procedural query language, like Relational Algebra, you write a query as an expression consisting of relations and Algebra Operators, like join, cross product, projection, restriction, etc.
- On the contrary, query SQL are called “non procedural” since they express the expected result only through its properties, and not the order of the operators to be performed to produce it



Relational Algebra

- Created by Edgar F Codd at IBM in 1970
- Procedural language
- Six basic operators
 - select: σ
 - project: Π
 - union: \cup
 - set difference: $-$
 - Cartesian product: \times
 - rename: ρ
- The operators take one or two relations as inputs and produce a new relation as a result



Select Operation

- Notation: $\sigma_p(r)$
- p is called the **selection predicate**
- Defined as:

$$\sigma_p(r) = \{t \mid t \in r \text{ and } p(t)\}$$

where p is a formula in propositional calculus consisting of **terms** connected by : \wedge (and), \vee (or), \neg (not)

| A | B | C | D |
|----------|----------|----|----|
| α | α | 1 | 7 |
| α | β | 5 | 7 |
| β | β | 12 | 3 |
| β | β | 23 | 10 |

Each **term** is one of:

<attribute> op <attribute> or <constant>

where op is one of: $=, \neq, >, \geq, <, \leq$

- Example of selection:

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

| A | B | C | D |
|----------|----------|----|----|
| α | α | 1 | 7 |
| β | β | 23 | 10 |

$\sigma_{A=B \wedge D > 5}(r)$

Project Operation

- Notation:

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

where A_1, A_2 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
- Example: To eliminate the *dept_name* attribute of *instructor*

$$\Pi_{ID, name, salary}(instructor)$$

| A | B | C |
|----------|----|---|
| α | 10 | 1 |
| α | 20 | 1 |
| β | 30 | 1 |
| β | 40 | 2 |

| A | C |
|----------|---|
| α | 1 |
| α | 1 |
| β | 1 |
| β | 2 |

=

| A | C |
|----------|---|
| α | 1 |
| β | 1 |

| A | C |
|---------|---|
| β | 2 |

$$\Pi_{A,C}(r)$$



Union Operation

- Notation: $r \cup s$
- Defined as:

$$r \cup s = \{t \mid t \in r \text{ or } t \in s\}$$

- For $r \cup s$ to be valid.
 - r, s must have the same **arity** (same number of attributes)
 - 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 2009 semester, or in the Spring 2010 semester, or in both

$$\Pi_{course_id} (\sigma_{semester='Fall'} \wedge year=2009 (section)) \cup$$
$$\Pi_{course_id} (\sigma_{semester='Spring'} \wedge year=2010 (section))$$

| A | B |
|----------|---|
| α | 1 |
| α | 2 |
| β | 1 |

 r

| A | B |
|----------|---|
| α | 2 |
| β | 3 |

 s

| A | B |
|----------|---|
| α | 1 |
| α | 2 |
| β | 1 |
| β | 3 |

 $r \cup s$



Set Difference Operation

- Notation $r - s$
- Defined as:

$$r - s = \{t \mid t \in r \text{ and } t \notin s\}$$

- 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 2009 semester, but not in the Spring 2010 semester

$$\Pi_{course_id} (\sigma_{semester="Fall"} \wedge year=2009 (section)) -$$
$$\Pi_{course_id} (\sigma_{semester="Spring"} \wedge year=2010 (section))$$

| A | B |
|----------|---|
| α | 1 |
| α | 2 |
| β | 1 |

 r

| A | B |
|----------|---|
| α | 2 |
| β | 3 |

 s

| A | B |
|----------|---|
| α | 1 |
| β | 1 |

 $r - s$



Set-Intersection Operation

- Notation: $r \cap s$

- Defined as:

$$r \cap s = \{ t \mid t \in r \text{ and } t \in s \}$$

- Assume:

- r, s have the *same arity*
- attributes of r and s are compatible

- Note: $r \cap s = r - (r - s)$

| A | B |
|----------|---|
| α | 1 |
| α | 2 |
| β | 1 |

r

| A | B |
|----------|---|
| α | 2 |
| β | 3 |

s

| A | B |
|----------|---|
| α | 2 |

$r \cap s$



Cartesian-Product Operation

- Notation $r \times s$
- Defined as:

$$r \times s = \{t q \mid t \in r \text{ and } q \in s\}$$

- Assume that attributes of $r(R)$ and $s(S)$ are disjoint. (That is, $R \cap S = \emptyset$)
- If attributes of $r(R)$ and $s(S)$ are not disjoint, then renaming must be used

| A | B | C | D | E |
|----------|---|----------|----|---|
| α | 1 | α | 10 | a |
| β | 2 | β | 10 | a |
| γ | | β | 20 | b |
| γ | | γ | 10 | b |
| α | 1 | α | 10 | a |
| α | 1 | β | 10 | a |
| α | 1 | β | 20 | b |
| α | 1 | γ | 10 | b |
| β | 2 | α | 10 | a |
| β | 2 | β | 10 | a |
| β | 2 | β | 20 | b |
| β | 2 | γ | 10 | b |

 $r \times s$



Rename Operation

- Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
- Allows us to refer to a relation by more than one name.
- Example:

$$\rho_X(E)$$

returns the expression E under the name X

- If a relational-algebra expression E has arity n , then

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

returns the result of expression E under the name X , and with the attributes renamed to A_1, A_2, \dots, A_n .



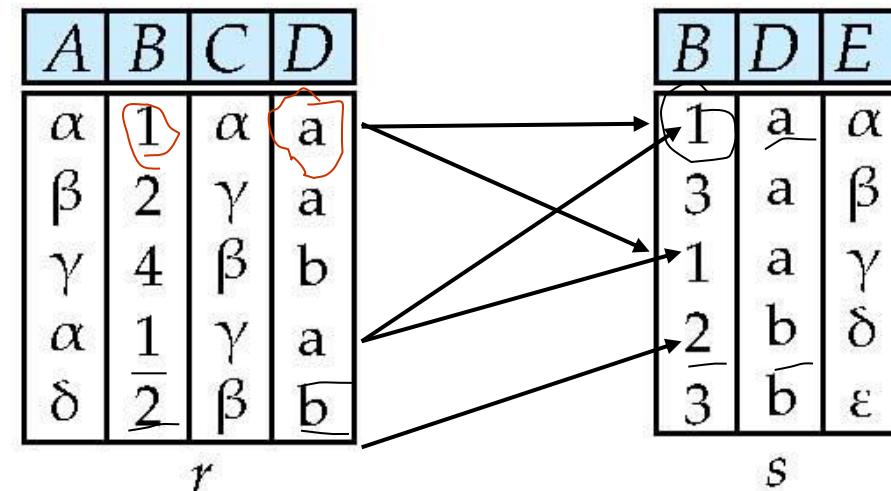
Natural Join- Example

- The natural join operation on two relations matches tuples whose values are the same on **all attribute names that are common to both relations.**

- Relations r, s:

- Natural Join

 - $r \bowtie s$



| A | B | C | D | E |
|----------|---|----------|---|----------|
| α | 1 | α | a | α |
| α | 1 | α | a | γ |
| α | 1 | γ | a | α |
| α | 1 | γ | a | γ |
| δ | 2 | β | b | δ |



- JOIN operation
- Joins two relation instructor and department on a COMMON attribute

| <i>ID</i> | <i>name</i> | <i>salary</i> | <i>dept_name</i> | <i>building</i> | <i>budget</i> |
|-----------|-------------|---------------|------------------|-----------------|---------------|
| 10101 | Srinivasan | 65000 | Comp. Sci. | Taylor | 100000 |
| 12121 | Wu | 90000 | Finance | Painter | 120000 |
| 15151 | Mozart | 40000 | Music | Packard | 80000 |
| 22222 | Einstein | 95000 | Physics | Watson | 70000 |
| 32343 | El Said | 60000 | History | Painter | 50000 |
| 33456 | Gold | 87000 | Physics | Watson | 70000 |
| 45565 | Katz | 75000 | Comp. Sci. | Taylor | 100000 |
| 58583 | Califieri | 62000 | History | Painter | 50000 |
| 76543 | Singh | 80000 | Finance | Painter | 120000 |
| 76766 | Crick | 72000 | Biology | Watson | 90000 |
| 83821 | Brandt | 92000 | Comp. Sci. | Taylor | 100000 |
| 98345 | Kim | 80000 | Elec. Eng. | Taylor | 85000 |



Exercise for students

Player relation

| Player Id | Team Id | Country | Age | Runs | Wickets |
|-----------|---------|-----------|-----|-------|---------|
| 1001 | 101 | India | 25 | 10000 | 300 |
| 1004 | 101 | India | 28 | 20000 | 200 |
| 1006 | 101 | India | 22 | 15000 | 150 |
| 1005 | 101 | India | 21 | 12000 | 400 |
| 1008 | 101 | India | 22 | 15000 | 150 |
| 1009 | 103 | England | 24 | 6000 | 90 |
| 1010 | 104 | Australia | 35 | 1300 | 0 |



1. Select all the tuples for which runs are greater than or equal to 15000.
2. Select all the players whose runs are greater than or equal to 6000 and age is less than 25
3. List all the countries in Player relation.



Deposit relation

| Acc. No. | Cust-name |
|----------|-----------|
| A 231 | Rahul |
| A 432 | Omkar |
| R 321 | Sachin |
| S 231 | Raj |
| T 239 | Sumit |

Borrower relation

| Loan No. | Cust-name |
|----------|-----------|
| P-3261 | Sachin |
| Q-6934 | Raj |
| S-4321 | Ramesh |
| T-6281 | Anil |



1. Find all the customers having an account but not the loan.
2. Find all the customers having a loan but not the account.
3. Rename Player relation to PlayerList.



Figure in-2.1

| Symbol (Name) | Example of Use |
|---------------------------------|---|
| σ (Selection) | $\sigma_{\text{salary} \geq 85000}(\text{instructor})$ Return rows of the input relation that satisfy the predicate. |
| Π (Projection) | $\Pi_{ID, \text{salary}}(\text{instructor})$ Output specified attributes from all rows of the input relation. Remove duplicate tuples from the output. |
| \bowtie (Natural Join) | $\text{instructor} \bowtie \text{department}$ Output pairs of rows from the two input relations that have the same value on all attributes that have the same name. |
| \times (Cartesian Product) | $\text{instructor} \times \text{department}$ Output all pairs of rows from the two input relations (regardless of whether or not they have the same values on common attributes) |
| \cup (Union) | $\Pi_{name}(\text{instructor}) \cup \Pi_{name}(\text{student})$ Output the union of tuples from the two input relations. |



Das Bild kann zurzeit nicht angezeigt werden.

End of Chapter 2

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

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



Figure 2.01

| <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 |



Figure 2.02

| <i>course_id</i> | <i>title</i> | <i>dept_name</i> | <i>credits</i> |
|------------------|----------------------------|------------------|----------------|
| BIO-101 | Intro. to Biology | Biology | 4 |
| BIO-301 | Genetics | Biology | 4 |
| BIO-399 | Computational Biology | Biology | 3 |
| CS-101 | Intro. to Computer Science | Comp. Sci. | 4 |
| CS-190 | Game Design | Comp. Sci. | 4 |
| CS-315 | Robotics | Comp. Sci. | 3 |
| CS-319 | Image Processing | Comp. Sci. | 3 |
| CS-347 | Database System Concepts | Comp. Sci. | 3 |
| EE-181 | Intro. to Digital Systems | Elec. Eng. | 3 |
| FIN-201 | Investment Banking | Finance | 3 |
| HIS-351 | World History | History | 3 |
| MU-199 | Music Video Production | Music | 3 |
| PHY-101 | Physical Principles | Physics | 4 |



Figure 2.03

| <i>course_id</i> | <i>prereq_id</i> |
|------------------|------------------|
| BIO-301 | BIO-101 |
| BIO-399 | BIO-101 |
| CS-190 | CS-101 |
| CS-315 | CS-101 |
| CS-319 | CS-101 |
| CS-347 | CS-101 |
| EE-181 | PHY-101 |



Figure 2.04

| <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 |



Figure 2.05

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



Figure 2.07

| <i>ID</i> | <i>course_id</i> | <i>sec_id</i> | <i>semester</i> | <i>year</i> |
|-----------|------------------|---------------|-----------------|-------------|
| 10101 | CS-101 | 1 | Fall | 2009 |
| 10101 | CS-315 | 1 | Spring | 2010 |
| 10101 | CS-347 | 1 | Fall | 2009 |
| 12121 | FIN-201 | 1 | Spring | 2010 |
| 15151 | MU-199 | 1 | Spring | 2010 |
| 22222 | PHY-101 | 1 | Fall | 2009 |
| 32343 | HIS-351 | 1 | Spring | 2010 |
| 45565 | CS-101 | 1 | Spring | 2010 |
| 45565 | CS-319 | 1 | Spring | 2010 |
| 76766 | BIO-101 | 1 | Summer | 2009 |
| 76766 | BIO-301 | 1 | Summer | 2010 |
| 83821 | CS-190 | 1 | Spring | 2009 |
| 83821 | CS-190 | 2 | Spring | 2009 |
| 83821 | CS-319 | 2 | Spring | 2010 |
| 98345 | EE-181 | 1 | Spring | 2009 |



Figure 2.10

| <i>ID</i> | <i>name</i> | <i>dept_name</i> | <i>salary</i> |
|-----------|-------------|------------------|---------------|
| 12121 | Wu | Finance | 90000 |
| 22222 | Einstein | Physics | 95000 |
| 33456 | Gold | Physics | 87000 |
| 83821 | Brandt | Comp. Sci. | 92000 |



Figure 2.11



Figure 2.12

| <i>ID</i> | <i>name</i> | <i>salary</i> | <i>dept_name</i> | <i>building</i> | <i>budget</i> |
|-----------|-------------|---------------|------------------|-----------------|---------------|
| 10101 | Srinivasan | 65000 | Comp. Sci. | Taylor | 100000 |
| 12121 | Wu | 90000 | Finance | Painter | 120000 |
| 15151 | Mozart | 40000 | Music | Packard | 80000 |
| 22222 | Einstein | 95000 | Physics | Watson | 70000 |
| 32343 | El Said | 60000 | History | Painter | 50000 |
| 33456 | Gold | 87000 | Physics | Watson | 70000 |
| 45565 | Katz | 75000 | Comp. Sci. | Taylor | 100000 |
| 58583 | Califieri | 62000 | History | Painter | 50000 |
| 76543 | Singh | 80000 | Finance | Painter | 120000 |
| 76766 | Crick | 72000 | Biology | Watson | 90000 |
| 83821 | Brandt | 92000 | Comp. Sci. | Taylor | 100000 |
| 98345 | Kim | 80000 | Elec. Eng. | Taylor | 85000 |



Figure 2.13

| <i>ID</i> | <i>salary</i> |
|-----------|---------------|
| 12121 | 90000 |
| 22222 | 95000 |
| 33456 | 87000 |
| 83821 | 92000 |

Chp 1-part2

Characteristics of database systems

Main Characteristics of the Database Approach

- **Self-describing nature of a database system:**
 - A DBMS **catalog** stores the description of a particular database (e.g. data structures, types, and constraints)
 - The description is called **meta-data**.
 - This allows the DBMS software to work with different database applications.
- **Insulation between programs and data:**
 - Called **program-data independence**.
 - Allows changing data structures and storage organization without having to change the database access programs.
 - Eg. Expanding table by adding or deleting new field or vice versa, change constraints,

Main Characteristics of the Database Approach (continued)

- **Data Abstraction:**

- A **data model** is used to hide storage details and present the users with a conceptual view of the database.
- Programs refer to the data model constructs rather than data storage details

| Data Item Name | Starting Position in Record | Length in Characters (bytes) |
|----------------|-----------------------------|------------------------------|
| Name | 1 | 30 |
| Student_number | 31 | 4 |
| Class | 35 | 1 |
| Major | 36 | 4 |

Figure 1.4
Internal storage format for a STUDENT record, based on the database catalog in Figure 1.3.

Main Characteristics of the Database Approach (continued)

- **Support of multiple views of the data:**
 - Each user may see a different view of the database, which describes **only** the data of interest to that user.

Main Characteristics of the Database Approach (continued)

- **Sharing of data and multi-user transaction processing:**
 - Allowing a set of **concurrent users** to retrieve from and to update the database.
 - *Concurrency control* within the DBMS guarantees that each **transaction** is correctly executed or aborted
 - *Recovery* subsystem ensures each completed transaction has its effect permanently recorded in the database
 - **OLTP** (Online Transaction Processing) is a major part of database applications. This allows hundreds of concurrent transactions to execute per second.

Chapter 3: Introduction to SQL

Chapter 3: Introduction to SQL

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

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
 - SQL-86, SQL-89, SQL-92
 - SQL:1999, SQL:2003, SQL:2008
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
 - Not all examples here may work on your particular system.

Data Definition Language

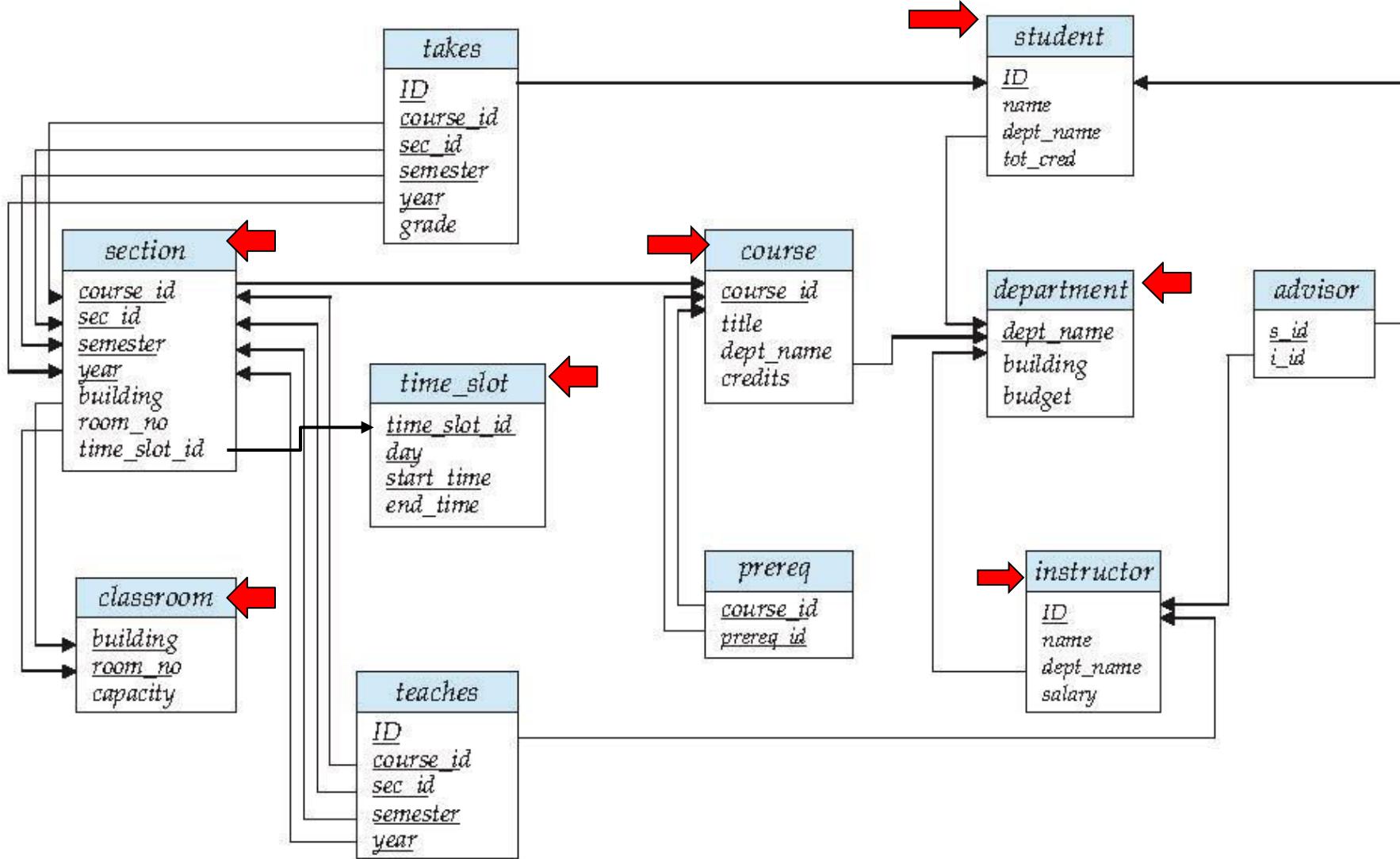
The SQL **data-definition language (DDL)** allows the specification of information about relations, including:

- The **schema** for each relation.
- The **domain** of values associated with each attribute.
- Integrity **constraints**
- And as we will see later, also other information such as
 - The set of **indices** to be maintained for each relations.
 - **Security and authorization** information for each relation.
 - The **physical storage structure** of each relation on disk.

Basic 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.
- **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.

Schem



Create Table Construct

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

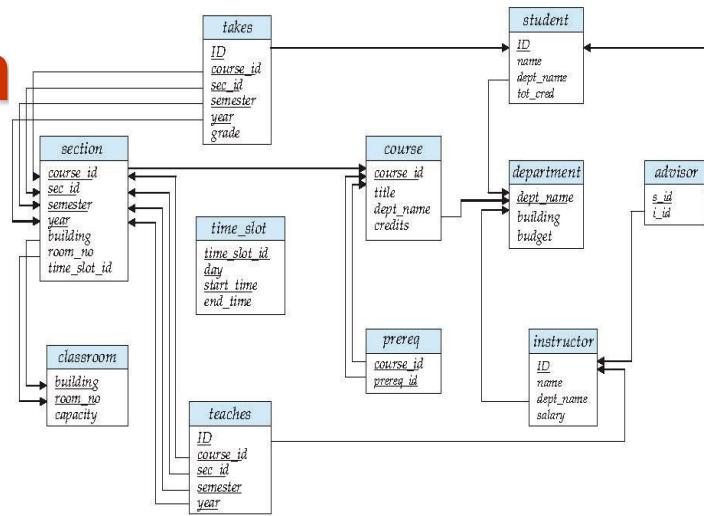
```
create table r ( $A_1 D_1, A_2 D_2, \dots, A_n D_n,$ 
                (integrity-constraint1),
                ...,
                (integrity-constraintk))
```

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

```
create table instructor(
    ID          char(5),
    name        varchar(20) not null,
    dept_name   varchar(20),
    salary      numeric(8,2))
```

Integrity Constraints in

- **not null**
- **primary key (A_1, \dots, A_n)**
- **foreign key (A_m, \dots, A_n) references r**



Example: Declare *ID* as the primary key for *instructor*

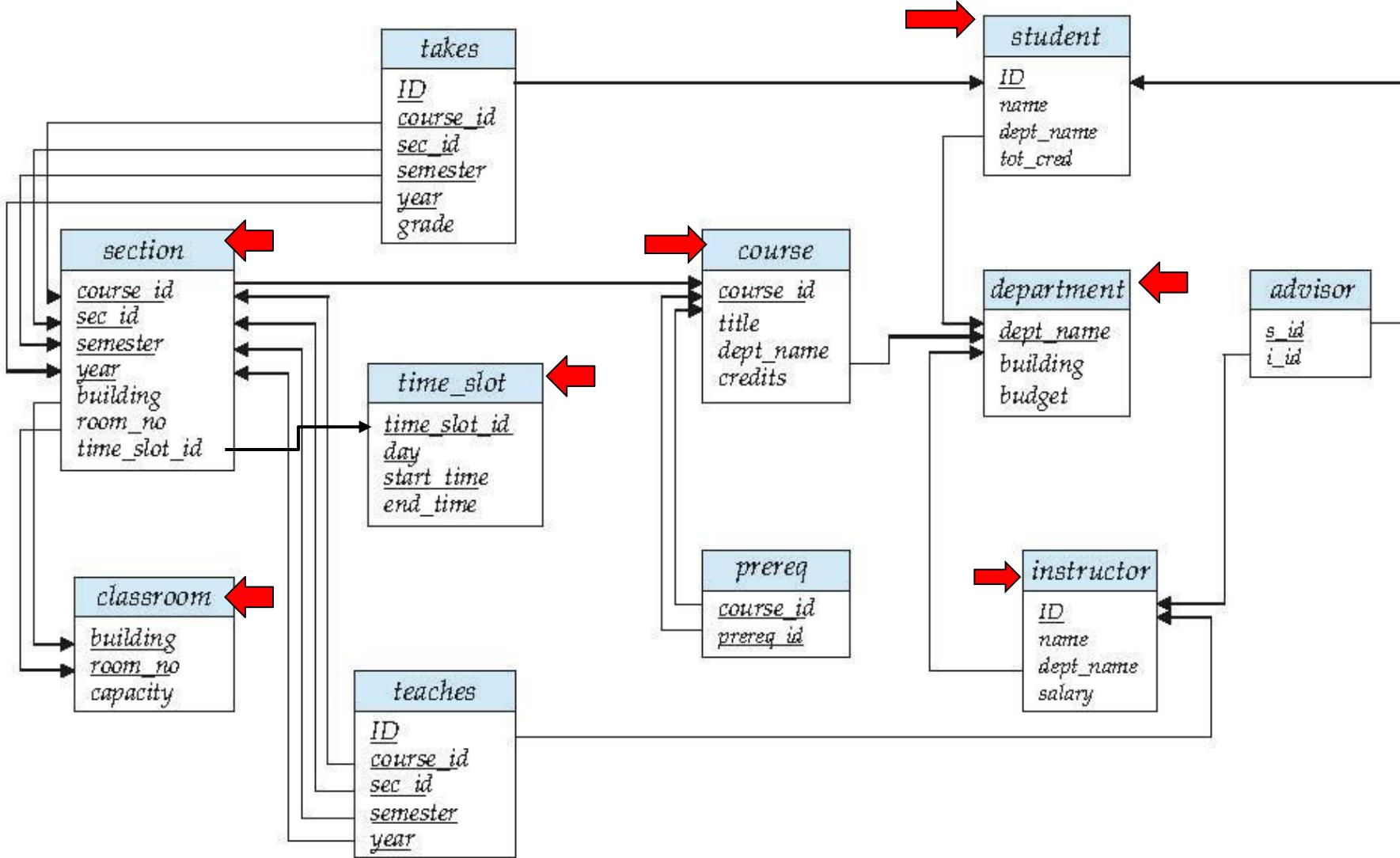
```
create table instructor (
```

| | |
|---|-------------------------------|
| <i>ID</i> | char(5), |
| <i>name</i> | varchar(20) not null , |
| <i>dept_name</i> | varchar(20), |
| <i>salary</i> | numeric(8,2), |
| primary key (<i>ID</i>), | |
| foreign key (<i>dept_name</i>) references department) | |

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

Class work

Schem



for Work out

And a Few More Relation Definitions

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

- `create table takes (`
 `ID varchar(5),`
 `course_id varchar(8),`
 `sec_id varchar(8),`
 `semester varchar(6),`
 `year numeric(4,0),`
 `grade varchar(2),`
 `primary key (ID, course_id, sec_id, semester, year),`
 `foreign key (ID) references student,`
 `foreign key (course_id, sec_id, semester, year) references section);`

- Note: `sec_id` can be dropped from primary key above, to ensure a student cannot be registered for two sections of the same course in the same semester

And still more

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

- Primary key declaration can be combined with attribute declaration as shown above

Changes to the table

- **insert into** *instructor* **values** ('10211', 'Smith', 'Biology', 66000);
- **drop table** *student*
 - Deletes the table and its contents
- **delete from** *student* [*where conditions*];
 - Deletes all contents of table, **but retains table**
- **alter table**
 - **alter table** *r* **add** *A D*
 - ▶ where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A*.
 - ▶ All tuples in the relation are assigned *null* as the value for the new attribute.
 - **alter table** *r* **drop** *A*
 - ▶ where *A* is the name of an attribute of relation *r*
 - ▶ Dropping of attributes not supported by many databases

Drop and Alter Table Constructs

■ alter table

- **Alter table table_Name Modify Col_Name datatype constraint(s)**
- ***Alter table r modify A D C(s)***
 - ▶ where **A** is the name of the attribute to be added to relation **r** and **D** is the domain of **A**.

- Update the contents

```
UPDATE table_name SET column_name = new_value WHERE some_condition;
```

Lets take a sample table **student**,

| student_id | name | age |
|------------|-------|-----|
| 101 | Adam | 15 |
| 102 | Alex | |
| 103 | chris | 14 |

```
UPDATE student SET age=18 WHERE student_id=102;
```

| S_id | S_Name | age |
|------|--------|-----|
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | chris | 14 |

Part 2

SELECT... FROM... WHERE QUERY

Basic Query Structure

- The SQL **data-manipulation language (DML)** provides the ability to query information, and insert, delete and update tuples
- A typical SQL query has the form:

```
select  $A_1, A_2, \dots, A_n$ 
from  $r_1, r_2, \dots, r_m$ 
where  $P$ 
```

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

The select Clause

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

`select name from instructor;`

Queries on a Single Relation

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



| Name |
|------------|
| Einstein |
| Wu |
| El Said |
| Katz |
| Kim |
| Crick |
| Srinivasan |
| Califieri |
| Brandt |
| Mozart |
| Gold |
| Singh |

Select dept_name from inst

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

The select Clause (Cont.)

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

```
select distinct dept_name  
from instructor
```

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

```
select all dept_name  
from instructor
```

The select Clause (Cont.)

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

```
select *
from instructor
```

- 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.

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 with salary > 80000

```
select name  
from instructor  
where dept_name = 'Comp. Sci.' and salary > 80000
```

- Comparison results can be combined using the logical connectives **and**, **or**, **not**
- Comparisons can be applied to results of arithmetic expressions.

workout

-- create a table

```
CREATE TABLE students ( id INTEGER PRIMARY KEY,  
                      name TEXT NOT NULL,  
                      gender TEXT NOT NULL);
```

-- insert some values

```
INSERT INTO students VALUES (1, 'Ryan', 'M');  
INSERT INTO students VALUES (2, 'Joanna', 'F');
```

-- fetch some values

```
SELECT * FROM students WHERE gender = 'F';
```

Querying

An example, suppose we want to answer the query "Retrieve the names of all instructors, along with their department names and department building name."

Technique of the schema of the relation *instructor* is used here.

Looking at the schema of the relation *instructor*, we realize that we can get the department name from the attribute **dept name**, but the department **building name** is present in the attribute *building* of the relation *department*.

To answer the query, each tuple in the *instructor* relation must be matched with the tuple in the *department* relation whose *dept name* value matches the *dept name* value of the *instructor* tuple.

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

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

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

Cartesian Product: *instructor X teaches*

instructor

| <i>ID</i> | <i>name</i> | <i>dept_name</i> | <i>salary</i> |
|-----------|-------------|------------------|---------------|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 12121 | Wu | Finance | 90000 |
| 15151 | Mozart | Music | 40000 |
| 22222 | Einstein | Physics | 95000 |
| 32343 | El Said | History | 60000 |
| 22456 | Gopal | Finance | 87000 |

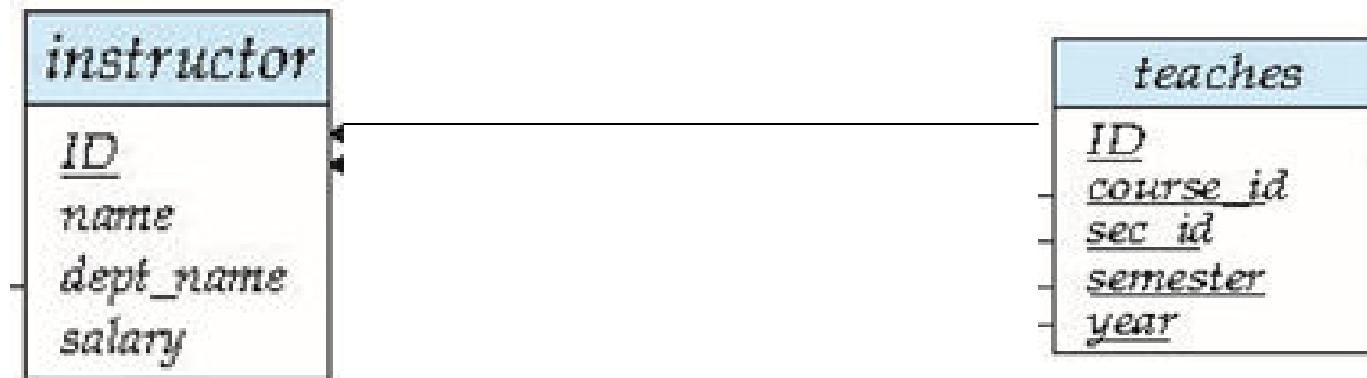
teaches

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

joins

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

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



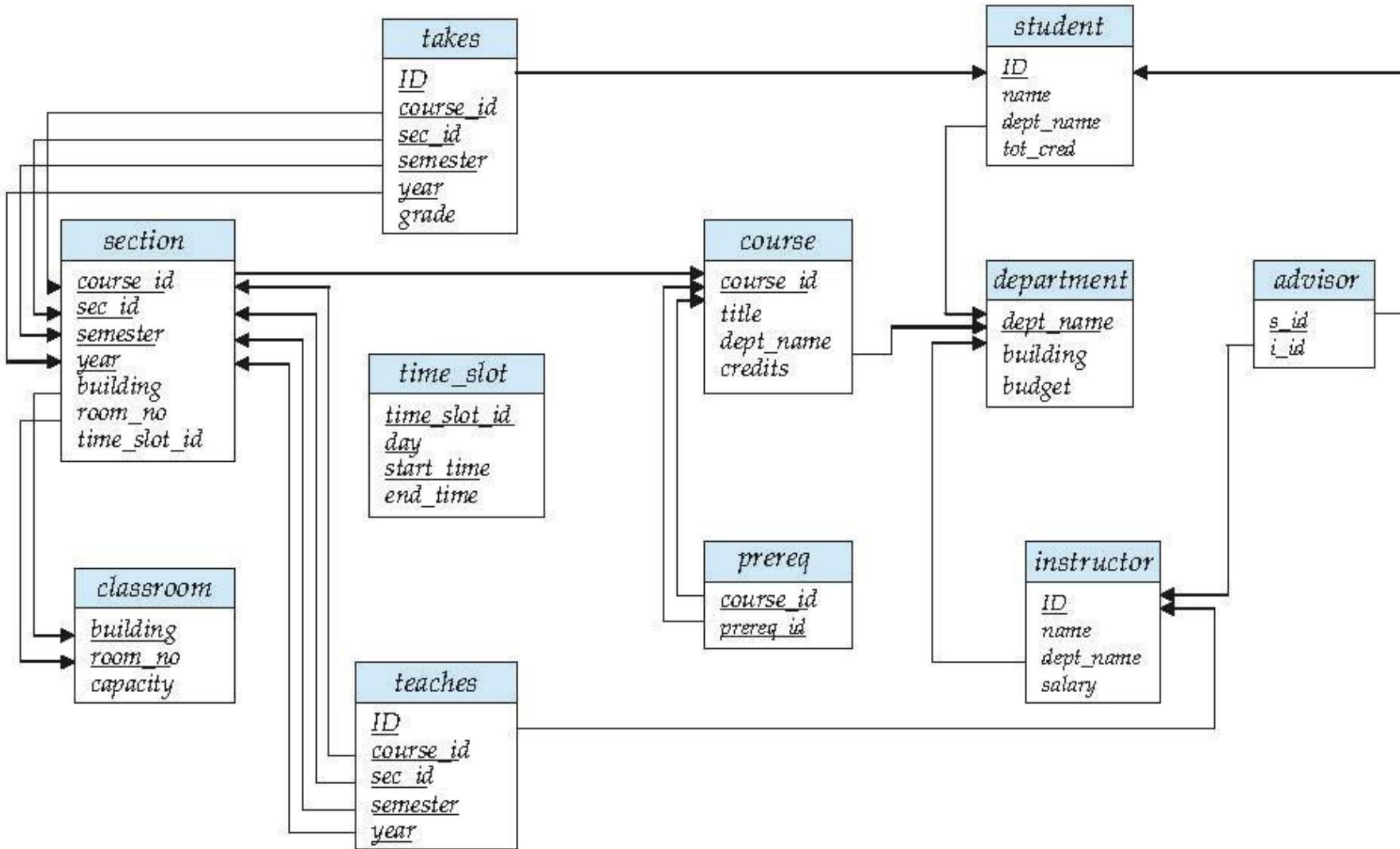
Joins

- Find the course ID, semester, year and title of each course offered by the Comp. Sci. department

```
select section.course_id, semester, year, title  
from section, course  
where section.course_id = course.course_id and  
      dept_name = 'Comp. Sci.'
```



Try Writing Some Queries in SQL



1. Display id of students who have taken the course offered by physics department

2. List out the 4 credit courses offered by ICT department

3. Display names of all advisors from Maths department

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

5. Display Student and their respective advisor information

6. Display Course and its prerequisite information

7. Display the computer science courses which are held in room no=105 and building AB5

8. Display the student' details who scored A or A+ grade in DBS course.

(solution in doc file)

Basic sql- part 3

Natural join to null values in aggregate function

Natural Join

- Natural join matches tuples with **the same values for all common attributes**, and retains only one copy of each common column
- **select *from instructor natural join teaches;**

| <i>ID</i> | <i>name</i> | <i>dept_name</i> | <i>salary</i> | <i>course_id</i> | <i>sec_id</i> | <i>semester</i> | <i>year</i> |
|-----------|-------------|------------------|---------------|------------------|---------------|-----------------|-------------|
| 10101 | Srinivasan | Comp. Sci. | 65000 | CS-101 | 1 | Fall | 2009 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | CS-315 | 1 | Spring | 2010 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | CS-347 | 1 | Fall | 2009 |
| 12121 | Wu | Finance | 90000 | FIN-201 | 1 | Spring | 2010 |
| 15151 | Mozart | Music | 40000 | MU-199 | 1 | Spring | 2010 |
| 22222 | Einstein | Physics | 95000 | PHY-101 | 1 | Fall | 2009 |
| 32343 | El Said | History | 60000 | HIS-351 | 1 | Spring | 2010 |
| 45565 | Katz | Comp. Sci. | 75000 | CS-101 | 1 | Spring | 2010 |
| 45565 | Katz | Comp. Sci. | 75000 | CS-319 | 1 | Spring | 2010 |
| 76766 | Crick | Biology | 72000 | BIO-101 | 1 | Summer | 2009 |
| 76766 | Crick | Biology | 72000 | BIO-301 | 1 | Summer | 2010 |

Natural Join Example

- List the names of instructors along with the course ID of the courses that they taught.
- In two ways we can write the Query:
 - **select name, course_id from instructor, teaches where instructor.ID = teaches.ID;**

OR

- **select name, course_id
from instructor natural join teaches;**

- A **from** clause in an SQL query can have multiple relations combined using natural join, as shown here:
- **select** A_1, A_2, \dots, A_n
- **from** r_1 **natural join** r_2 **natural join** \dots **natural join** r_m
- **where** $P;$

- Retrieve the names of all instructors, along with their department names and department building name.
- `instructor(ID, name, dept_name, salary)`
- `department(dept_name, building, budget)`
- Query:
- Select name, dept_name , building from instructor, department where instructor.dept_name=department.dept_name
- **Using natural join**

- Display the **computer science courses** which are held in **room no=105** and building AB5
 - course(course_id, title, dept_name, credits)
 - section(course_id, sec_id, semester, year, building, room_number, time_slot_id)
-
- select course_id from course, section where course.course_id=section.course_id and building='AB5' and room_number=105 and dept_name='CS';
 - **using natural join:**

Natural Join (Cont.)

- Danger in natural join: beware of unrelated attributes with same name which get equated incorrectly
- course(course_id, title, dept_name, credits)
- teaches(ID, course_id, sec_id, semester, year)
- instructor(ID, name, dept_name, salary)
- List the names of instructors along with the titles of courses that they teach
 - **select name, title
from instructor natural join teaches natural join course;**

the natural join of *instructor* and *teaches* contains the attributes (*ID*, *name*, *dept name*, *salary*, *course id*, *sec id*, *sem*, *year*),

while the *course* relation contains the attributes (*course id*, *title*, *dept name*, *credits*).

As a result, the natural join of these two would require that the *dept name* attribute values from the two inputs be the same, in addition to requiring that the *course id* values be the same. This query would then omit all (instructor name, course title) pairs where the instructor

teaches a course in a department other than the instructor's own department.

- $(ID, name, dept\ name, salary, course\ id, sec\ id, sem, year),$
- $(course\ id, title, dept\ name, credits).$
- This query would then omit all (instructor name, course title) pairs where the instructor teaches a course in a department other than the instructor's own department

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

- correct version

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

- The operation **join . . . using** requires a list of attribute names to be specified.
- No other common attribute even if present will be matched, other than the ones indicated in using clause..

- *Display id of students who have taken the course offered by physics department*
- student(SID, name, dept_name, tot_cred)
- takes(SID, course_id, sec_id, semester, year, grade)
- course(course_id, title, dept_name, credits)
-
-
- Select student.SID from student, course, takes where student.SID=takes.SID and course.course_id =takes.course_id and course.dept_name="physics" ;
- **Select student.SID from (student natural join takes) join course using course_id where course.dept_name="physics"**

- Display the student' details who scored A or A+ grade in “DBS” course.
- student(SID, name, dept_name, tot_cred)
- takes(SID, course_id, sec_id, semester, year, **grade**)
- course(course_id, **title**, dept_name, credits)

Additional Basic Operations

The Rename Operation

The names of the attributes in the result are derived from the names of the attributes in the relations in the **from** clause.

We may need to change this for the following reasons:

- two relations in the **from** clause may have attributes with the same name, in which case an attribute name is duplicated in the result.
 - if we used an arithmetic expression in the **select** clause, the resultant attribute does not have a name.
 - even if an attribute name can be derived from the base relations as in the preceding example, we may want to change the attribute name in the result.
- The SQL allows renaming relations and attributes using the **as** clause:

old-name as new-name

- E.g.
- **select** *ID, name, salary/12 as monthly_salary* **from** *instructor*

- One reason to rename a relation is to replace a long relation name with a shortened version that is more convenient to use elsewhere in the query.
- **select** *T.name, S.course id*
- **from** *instructor as T, teaches as S*
- **where** *T.ID= S.ID;*

Another reason to rename a relation is a case where we wish to compare tuples in the same relation. We then need to take the Cartesian product of a relation with itself and, without renaming, it becomes impossible to distinguish one tuple from the other.

- 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.’**

An identifier, such as T and S , that is used to rename a relation is referred to as a **correlation name** in the SQL standard, but is also commonly referred to as a **table alias**, or a **correlation variable**, or a **tuple variable**.

String Operations

Note: The SQL standard specifies that the equality operation on strings is case sensitive; as a result the expression “comp. sci.” = ‘Comp. Sci.’” evaluates to **false**. However, some database systems, such as **MySQL** and **SQL Server**, do not distinguish uppercase from lowercase when matching strings; as a result “comp. sci.”= ‘Comp. Sci.’” would evaluate to **true** on these databases. This

- 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%'

■ Patters are **case sensitive**.

■ Pattern matching examples:

- ‘Intro%’ matches any string **beginning with “Intro”**.
- ‘%Comp%’ matches any string **containing “Comp” as a substring**.
- ‘____’ matches any string of **exactly three characters**.
- ‘___ %’ matches any string of at **least three characters**.

exercises • display all the records from
the CUSTOMERS table, where
the SALARY starts with 200.

```
SELECT * FROM table_name WHERE column LIKE 'X%'
```

```
SELECT * FROM table_name WHERE column LIKE '%X%'
```

```
SELECT * FROM table_name WHERE column LIKE 'X_'
```

```
SELECT * FROM table_name WHERE column LIKE '_X'
```

```
SELECT * FROM table_name WHERE column LIKE '_X_'
```

String Operations (Cont.)

- Match the string “100 %” // Usage of % in a string
- ‘\’ is used to specify the special character such as % or \.

(eg. Retrieve whose attendance status is 100%)

Select names from student where attendance like ‘100\%’

- 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.

Eg.: Select **SUBSTR**(‘ABCDEF’, 2, 3) as substring from tablename ; o/p: BCD

select **length**(name) **as** length_val from dual ;

select **upper**(‘hi’) **as** Upper_val from dual ;

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**

- *General Form:*

- SELECT distinct name**
 - FROM instructor**
 - WHERE Dept_name='CS'**
 - ORDER BY 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'*);
- All SQL platforms may not support this syntax

Set Operations

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

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

union

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

- Find courses that ran in Fall 2009 and in Spring 2010
- **(select course_id from section where sem = 'Fall' and year = 2009)**
intersect
(select course_id from section where sem = 'Spring' and year = 2010)

- Find courses that ran in Fall 2009 but not in Spring 2010
- **(select course_id from section where sem = 'Fall' and year = 2009)**
except
- **(select course_id from section where sem = 'Spring' and year = 2010)**

Set Operations

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

Set Operations (Cont.)

- Find the salaries of all instructors that are less than the largest salary ✓

```
select distinct T.salary  
from instructor as T, instructor as S  
where T.salary < S.salary
```

- Find all the salaries of all instructors ✓

```
select distinct salary  
from instructor
```

- Find the largest salary of all instructors

```
(select "second query" )  
except  
(select "first query")
```

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;
```

| ID | Student | Email1 | Email2 | Phone | Father |
|-----|-----------------|------------------------------|-----------------------|---------------|-------------------|
| 1 | Stan Marsh | NULL | smarsh@gmail.com | 740-097-0951 | Randy Marsh |
| 2 | Butters Stotch | bstotch@spelementary.com | bstotch@gmail.com | NULL | Stephen Stotch |
| 3 | Kenny McCormick | NULL | NULL | | Stuart McCormick |
| 4 | Kyle Broflovski | kbroflovski@spelementary.com | kbroflovski@gmail.com | 619-722-2491 | Gerald Broflovski |
| 666 | Eric Cartman | ecartman@spelementary.com | ecartman@gmail.com | 740-6733-5671 | NULL |

| ID | Student | Email1 | Email2 |
|----|-----------------|--------|--------|
| 3 | Kenny McCormick | NULL | NULL |

- **SELECT ID, Student, Email1, Email2**
FROM tblSouthPark WHERE Email1 IS NULL AND Email2 IS NULL;

Null Values and Three Valued Logic

- Any comparison with *null* returns *unknown*
 - Example: $5 < \text{null}$ or $\text{null} < > \text{null}$ or $\text{null} = \text{null}$
- Three-valued logic using the truth value *unknown*:
 - OR: (*unknown or true*) = *true*,
 (*unknown or false*) = *unknown*
 (*unknown or unknown*) = *unknown*
 - AND: (*true and unknown*) = *unknown*,
 (*false and unknown*) = *false*,
 (*unknown and unknown*) = *unknown*
 - NOT: (**not unknown**) = *unknown*
 - “**P is unknown**” evaluates to true if P is not known

Aggregate Functions

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

avg: average value

min: minimum value

max: maximum value

sum: sum of values

count: number of values

Aggregate Functions (Cont.)

- Find the average salary of instructors in the Computer Science department
 - **select avg (salary)**
from instructor
where dept_name= 'Comp. Sci.';
- Find the total number of instructors who teach a course in the Spring 2010 semester
 - **select count (distinct ID)** [Distinct: Since same teacher teaching more than one subject in Spring 10]
from teaches
where semester= 'Spring' and year = 2010
- Find the number of tuples in the *course* relation
 - **select count (*)**
from course;

“group by” clause

- There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this wish in SQL using the **group by** clause.
- The attribute or attributes given in the **group by** clause are used to form groups.
- Tuples with the same value on all attributes in the **group by** clause are placed in one group.

Aggregate Functions – Group By

- Find the average salary of instructors in each department
 - `select dept_name, avg (salary) from instructor
group by dept_name;`
 - Note: departments with no instructor will not appear in result

| <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 |

- Find the number of instructors in each department who teach a course in the Spring 2
- **select *dept name*, count (distinct *ID*) as *instr_count***
- **from *instructor* natural**
- **where *semester* = 'Sprin**
- **group by *dept name*; 0**

| <i>dept_name</i> | <i>instr_count</i> |
|------------------|--------------------|
| Comp. Sci. | 3 |
| Finance | 1 |
| History | 1 |
| Music | 1 |

Aggregation (Cont.)

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list.

In other words, any attribute that is not present in the **group by** clause must appear only inside an aggregate function if it appears in the **select** clause, otherwise the query is treated as erroneous.

- /* erroneous query */
select *dept_name*, *ID*, **avg** (*salary*)
from *instructor*
group by *dept_name*;

Each instructor in a particular group (defined by *dept name*) can have a different *ID*, and since only one tuple is output for each group, there is no unique way of choosing which *ID* value to output. As a result, such cases are disallowed by SQL.

exercise

| ord_no | purch_amt | ord_date | customer_id | salesman_id |
|--------|-----------|------------|-------------|-------------|
| 70001 | 150.5 | 2012-10-05 | 3005 | 5002 |
| 70009 | 270.65 | 2012-09-10 | 3001 | 5005 |
| 70002 | 65.26 | 2012-10-05 | 3002 | 5001 |
| 70004 | 110.5 | 2012-08-17 | 3009 | 5003 |
| 70007 | 948.5 | 2012-09-10 | 3005 | 5002 |
| 70005 | 2400.6 | 2012-07-27 | 3007 | 5001 |
| 70008 | 5760 | 2012-09-10 | 3002 | 5001 |

- Write a SQL statement to find the total purchase amount of all orders
Select sum(purch_amt) from orders;
- Write a SQL statement to find the number of salesmen currently listing for all of their customers.
Select count(distinct salesman_id) from orders;

| ord_no | purch_amt | ord_date | customer_id | salesman_id |
|--------|-----------|------------|-------------|-------------|
| 70001 | 150.5 | 2012-10-05 | 3005 | 5002 |
| 70009 | 270.65 | 2012-09-10 | 3001 | 5005 |
| 70002 | 65.26 | 2012-10-05 | 3002 | 5001 |
| 70004 | 110.5 | 2012-08-17 | 3009 | 5003 |
| 70007 | 948.5 | 2012-09-10 | 3005 | 5002 |
| 70005 | 2400.6 | 2012-07-27 | 3007 | 5001 |
| 70008 | 5760 | 2012-09-10 | 3002 | 5001 |

```
1 | SELECT customer_id,ord_date,MAX(purch_amt)
2 | FROM orders
3 | GROUP BY customer_id,ord_date;
```

Output of the Query:

| customer_id | ord_date | max |
|-------------|------------|---------|
| 3002 | 2012-10-05 | 65.26 |
| 3003 | 2012-08-17 | 75.29 |
| 3005 | 2012-10-05 | 150.50 |
| 3007 | 2012-07-27 | 2400.60 |
| 3009 | 2012-08-17 | 110.50 |
| 3001 | 2012-09-10 | 270.65 |
| 3002 | 2012-09-10 | 5760.00 |
| 3005 | 2012-09-10 | 948.50 |
| 3009 | 2012-10-10 | 2480.40 |
| 3008 | 2012-06-27 | 250.45 |
| 3004 | 2012-10-10 | 1983.43 |
| 3002 | 2012-04-25 | 3045.60 |

| customer_id | cust_name | city | grade | salesman_id |
|-------------|----------------|------------|-------|-------------|
| 3002 | Nick Rimando | New York | 100 | 5001 |
| 3007 | Brad Davis | New York | 200 | 5001 |
| 3005 | Graham Zusi | California | 200 | 5002 |
| 3008 | Julian Green | London | 300 | 5002 |
| 3004 | Fabian Johnson | Paris | 300 | 5006 |
| 3009 | Geoff Cameron | Berlin | 100 | 5003 |
| 3003 | Jozv Altidor | Moscow | 200 | 5007 |

- Write a SQL statement to know how many customer have listed their names.
- find the number of customers who gets at least a gradation for his/her performance.
- find the highest purchase amount by the each customer with their ID and highest purchase amount
- **SELECT customer_id, MAX(purch_amt) FROM orders GROUP BY customer_id;**

“Group by... having “ clause

- At times, it is useful to state a condition that applies to groups rather than to tuples.
- For example, we might be interested in only those departments where the average salary of the instructors is more than \$42,000.
- This condition does not apply to a single tuple; rather, it applies to each group constructed by the **group by** clause.
- To express such a query, we use the **having** clause of SQL. SQL applies predicates in the **having** clause after groups have been formed, so aggregate functions may be used

Aggregate Functions – Having Clause

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

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

| dept_name | avg(avg_salary) |
|------------|-----------------|
| Physics | 91000 |
| Elec. Eng. | 80000 |
| Finance | 85000 |
| Comp. Sci. | 77333 |
| Biology | 72000 |
| History | 61000 |

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

```
SELECT commission, COUNT (*)
FROM agents
GROUP BY commission
HAVING COUNT (*) > 3;
```

agents

| AGENT_NAME | COMMISSION |
|------------|------------|
| Alex | .13 |
| Subbarao | .14 |
| Benjamin | .11 |
| Ramasundar | .15 |
| Alford | .12 |
| Ravi Kumar | .15 |
| Santakumar | .14 |
| Lucida | .12 |
| Anderson | .13 |
| Mukesh | .11 |
| McDen | .15 |
| Ivan | .15 |

GROUP BY commission

| COMMISSION | COUNT(*) |
|------------|----------|
| .15 | 4 |
| .11 | 2 |
| .14 | 2 |
| .13 | 2 |
| .12 | 2 |

HAVING COUNT (*) > 3;

| COMMISSION | COUNT(*) |
|------------|----------|
| .15 | 4 |

- To illustrate the use of both a **having** clause and a **where** clause in the same query, we consider the query “For each course section offered in 2009, find the average total credits (*tot cred*) of all students enrolled in the section, if the section had at least 2 students.”
- **select** *course_id, semester, year, sec_id, avg (tot cred)*
- **from** *takes* **natural join** *student*
- **where** *year = 2009*
- **group by** *course id, semester, year, sec_id*
- **having count (ID) >= 2;**

Null Values and Aggregates

- Total all salaries

```
select sum (salary )  
from instructor
```

- Above statement ignores null amounts

- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes

- What if collection has only null values?

- count returns 0
- all other aggregates return null



Table1:

| Field1 | Field2 | Field3 |
|--------|--------|--------|
| 1 | 1 | 1 |
| NULL | NULL | NULL |
| 2 | 2 | NULL |
| 1 | 3 | 1 |

Then

```
SELECT COUNT(*), COUNT(Field1), COUNT(Field2), COUNT(DISTINCT Field3)
FROM Table1
```

Output Is:

```
COUNT(*) = 4; -- count all rows even null/duplicates

-- count only rows without null values on that field
COUNT(Field1) = COUNT(Field2) = 3

COUNT(Field3) = 2
COUNT(DISTINCT Field3) = 1 -- Ignore duplicates
```

Basic sql-part 4

Aggregate and null value to scalar sub-query

Null Values and Aggregates

- Total all salaries

```
select sum (salary )  
from instructor
```

- Above statement ignores null amounts

- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes

- What if collection has only null values?

- count returns 0
- all other aggregates return null



```
SELECT 2 + NULL + 2 + 2 + 2 + 5 + 5 + 2 + 5
```

This **SELECT** statement will return NULL, since NULL plus anything always returns **NULL**.

If we use the **SUM()** function on the table, however, we can rest assured that NULLs are eliminated.

Result=25

Table1:

| Field1 | Field2 | Field3 |
|--------|--------|--------|
| 1 | 1 | 1 |
| NULL | NULL | NULL |
| 2 | 2 | NULL |
| 1 | 3 | 1 |

Then

```
SELECT COUNT(*), COUNT(Field1), COUNT(Field2), COUNT(DISTINCT Field3)
FROM Table1
```

Output Is:

```
COUNT(*) = 4; -- count all rows even null/duplicates

-- count only rows without null values on that field
COUNT(Field1) = COUNT(Field2) = 3

COUNT(Field3) = 2
COUNT(DISTINCT Field3) = 1 -- Ignore duplicates
```

Nested Subqueries

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

- Set membership
- SQL allows testing tuples for membership in a relation. The **in** connective tests for set membership, where the set is a collection of values produced by a **select** clause.
- The **not in** connective tests for the absence of set membership.
- “Find all the courses taught in both the Fall 2009 and Spring 2010 semesters.
- Question hint “by checking for membership....”

Set Membership (Operates: in, not in)

- Find courses offered in Fall 2009 **and** in Spring 2010

```
select distinct course_id
from section
where semester= 'Fall' and year= 2009 and
      course_id in (select course_id
                        from section
                        where semester= 'Spring' and year= 2010);
```

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

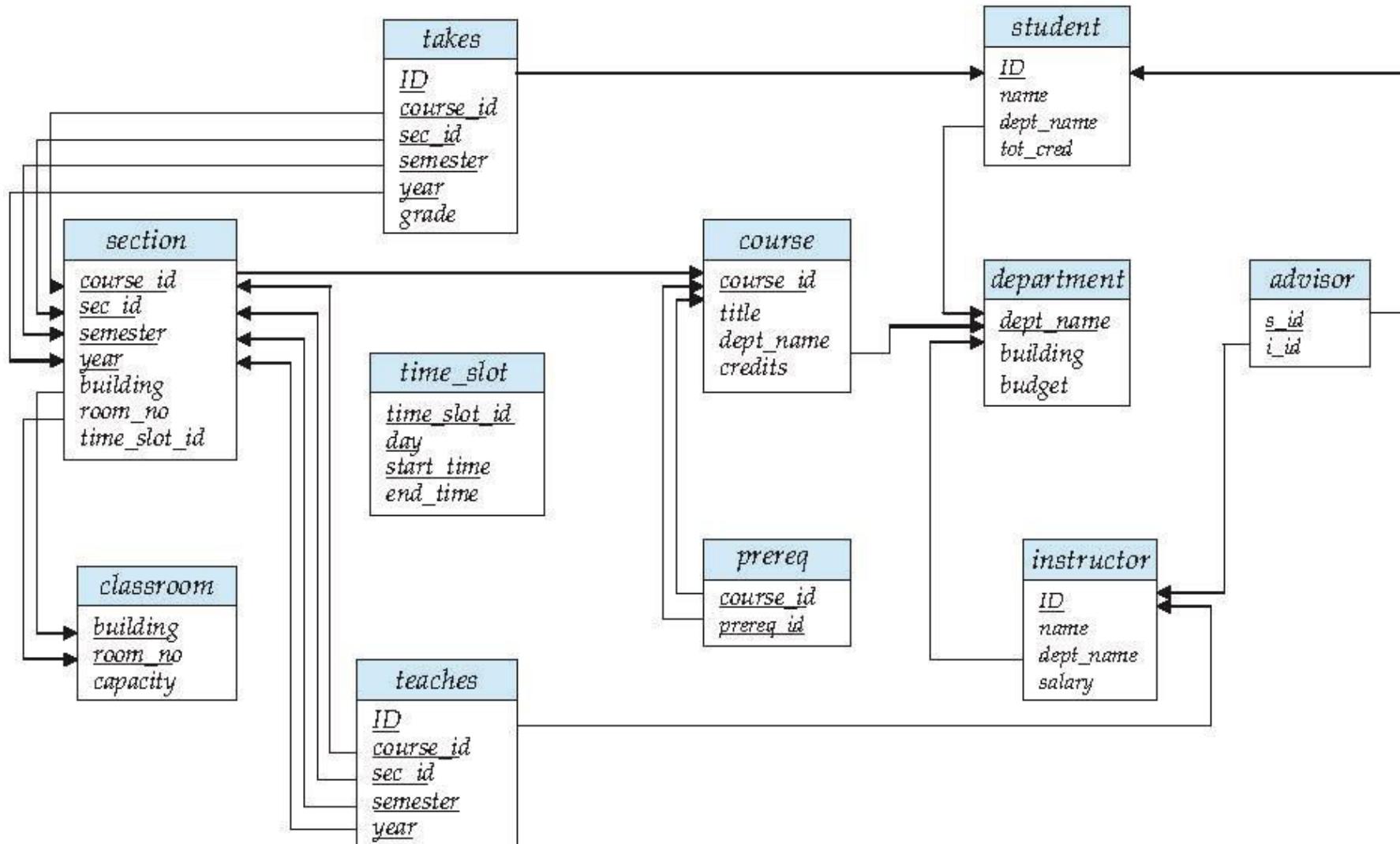
```
select distinct course_id
from section
where semester= 'Fall' and year= 2009 and
      course_id not in (select course_id
                        from section
                        where semester= 'Spring' and year= 2010);
```

Example Query

- 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);
```

- The **in** and **not in** operators can also be used on **enumerated sets**.
The following query selects the names of instructors whose names are neither “Mozart” nor “Einstein”.
 - **select distinct name**
 - **from instructor where name not in ('Mozart', 'Einstein');**



Set Comparison (operators: some, all)

- 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');
```

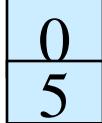
- The **> some** comparison in the **where** clause of the outer **select** is true if the *salary* value of the tuple is greater than at least one member of the set of all salary values for instructors in Biology.
- SQL also allows **< some**, **<= some**, **>= some**, **= some**, and **<> some** comparison
- **= some** is identical to **in**,
- whereas **<> some** is *not* the same as **not in**

Definition of Some Clause

- $F <\text{comp}> \text{some } r \Leftrightarrow \exists t \in r \text{ such that } (F <\text{comp}> t)$

Where comp can be: $<$, \leq , $>$, $=$, \neq

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

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

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

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

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

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

Example Query

- 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');
```

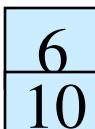
As it does for **some**, SQL also allows **< all**, **<= all**, **>= all**, **= all**, and **<> all** comparisons.

As an exercise, verify that **<> all** is identical to **not in**, whereas **= all** is *not* the same as **in**.

Definition of all Clause

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

(5 < all ) = false

(5 < all ) = true

(5 = all ) = false

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

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

However, $(= \mathbf{all}) \not\equiv \mathbf{in}$

Test for Empty Relations/ set cardinality

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$
- **Ex:** Get the list of courses which do not have pre-requisite courses.

Using SET operation

Select course_id from Course **except** select cours_id from Prereq

Using Nested Query : NOT exists

Select course_id from Coursre s where not exists (select pre_id from Prereq P where P.course_id=S.course_id)

Correlation Variables

- Yet another way of specifying the query “Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester”

```
select course_id  
from section as S  
where semester= 'Fall' and year= 2009 and  
exists (select *  
        from section as T  
        where semester= 'Spring' and year= 2010  
          and S.course_id= T.course_id);
```

- Correlated subquery
- Correlation name or correlation variable
-



- The above query also illustrates a feature of SQL where a correlation name from an outer query (*S* in the above query), can be used in a subquery in the **where** clause.
- What is **correlated subquery**?
- A subquery that uses a correlation name from an outer query is
- called a **correlated subquery**
- In queries that contain subqueries, a scoping rule applies for correlation names. In a subquery, according to the rule, it is legal to use only correlation names defined in the subquery itself or in any query that contains the subquery.
- If a correlation name is defined both locally in a subquery and globally in a containing query, the local definition applies

```
mysql> SELECT * FROM EMPLOYEE;
```

| SSN | DNO | SUPERSSN | FNAME | LNAME | ADDRESS | SEX | SALARY |
|----------|-----|----------|----------|-------|-----------|-----|--------|
| RNSACC01 | 1 | RNSACC02 | AHANA | K | MANGALORE | F | 350000 |
| RNSACC02 | 1 | NULL | SANTHOSH | KUMAR | MANGALORE | M | 300000 |
| RNSCSE01 | 5 | RNSCSE02 | JAMES | SMITH | BANGALORE | M | 500000 |
| RNSCSE02 | 5 | RNSCSE03 | HEARN | BAKER | BANGALORE | M | 700000 |
| RNSCSE03 | 5 | RNSCSE04 | EDWARD | SCOTT | mysore | M | 500000 |
| RNSCSE04 | 5 | RNSCSE05 | PAVAN | HEGDE | MANGALORE | M | 650000 |
| RNSCSE05 | 5 | RNSCSE06 | GIRISH | MALYA | mysore | M | 450000 |
| RNSCSE06 | 5 | NULL | NEHA | SN | BANGALORE | F | 800000 |
| RNSECE01 | 3 | NULL | JOHN | SCOTT | BANGALORE | M | 450000 |
| RNSISE01 | 4 | NULL | VEENA | M | mysore | M | 600000 |
| RNSIT01 | 2 | NULL | NAGESH | HR | BANGALORE | M | 500000 |

```
1 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM PROJECT;
```

| PNO | PNAME | DNO | PLOCATION |
|-----|-------------------|-----|-----------|
| 100 | IOT | 5 | BANGALORE |
| 101 | CLOUD | 5 | BANGALORE |
| 102 | BIGDATA | 5 | BANGALORE |
| 103 | SENSORS | 3 | BANGALORE |
| 104 | BANK MANAGEMENT | 1 | BANGALORE |
| 105 | SALARY MANAGEMENT | 1 | BANGALORE |
| 106 | OPENSTACK | 4 | BANGALORE |
| 107 | SMART CITY | 2 | BANGALORE |

```
8 rows in set (0.00 sec)
```

```
mysql> SELECT PNO FROM PROJECT WHERE DNO='8';
Empty set (0.00 sec)
```

```
mysql> SELECT E.FNAME, E.LNAME FROM EMPLOYEE E WHERE EXISTS(SELECT PNO FROM PROJECT WHERE DNO='8');
Empty set (0.00 sec)
```

```
mysql> SELECT PNO FROM PROJECT WHERE DNO='5';
+----+
| PNO |
+----+
| 100 |
| 101 |
| 102 |
+----+
3 rows in set (0.00 sec)
```

```
mysql> SELECT E.FNAME, E.LNAME FROM EMPLOYEE E WHERE EXISTS(SELECT PNO FROM PROJECT WHERE DNO='5');
+-----+-----+
| FNAME | LNAME |
+-----+-----+
| AHANA | K      |
| SANTHOSH | KUMAR |
| JAMES | SMITH |
| HEARN | BAKER |
| EDWARD | SCOTT |
| PAVAN | HEGDE |
| GIRISH | MALYA |
| NEHA | SN     |
| JOHN | SCOTT |
| VEENA | M      |
| NAGESH | HR    |
+-----+-----+
11 rows in set (0.00 sec)
```

Not Exists

We can test for the nonexistence of tuples in a subquery by using the **not exists**

construct

-
-

```
mysql> SELECT E.FNAME, E.LNAME FROM EMPLOYEE E WHERE NOT EXISTS(SELECT PNO FROM PROJECT WHERE DNO='8');
+-----+-----+
| FNAME | LNAME |
+-----+-----+
| AHANA | K      |
| SANTHOSH | KUMAR |
| JAMES | SMITH |
| HEARN | BAKER |
| EDWARD | SCOTT |
| PAVAN | HEGDE |
| GIRISH | MALYA |
| NEHA | SN     |
| JOHN | SCOTT |
| VEENA | M      |
| NAGESH | HR    |
+-----+-----+
11 rows in set (0.00 sec)
```

```
mysql> SELECT E.FNAME, E.LNAME FROM EMPLOYEE E WHERE NOT EXISTS(SELECT PNO FROM PROJECT WHERE DNO='5');
Empty set (0.00 sec)
```

- Find the 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 T.ID = S.ID));
```

```
(select course_id from course  
      where dept_name = 'Biology')
```

- BIO101
- BIO102

```
select T.course_id from takes as T  
      where T.ID = S.ID));
```

- CS101
- CS102
- BIO101
- EXCEPT: BIO102(there is a biology course not taken by student)

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a **subquery has any duplicate tuples** in its result.
 - (Evaluates to “true” on an empty set)
- Find all courses that were offered **at most once** in 2009

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

Q1: Get the students list who have taken **at most** one subject.

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

Subqueries in the From Clause (Cont.)

- To display instructor, his salary along with his department's average salary.

And yet another way to write it: **lateral** clause

```
select name, salary, avg_salary
from instructor I1,
     lateral (select avg(salary) as avg_salary
               from instructor I2
              where I2.dept_name= I1.dept_name);
```

- Lateral clause permits later part of the **from** clause (after the **lateral keyword**) to access correlation variables from the earlier part.
- Note: lateral is part of the SQL standard, but is **not supported on many database systems**; some databases such as SQL Server offer alternative syntax

With Clause

- The **with** clause provides a way of defining **a temporary view/relation/table** 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 budget  
from department, max_budget  
where department.budget = max_budget.value;
```



Scalar Subquery

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

- List out the number of instructors in each department.

- E.g.

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

- Runtime error if subquery returns more than one result tuple



Modification of the Database

part5

Modification of the Database

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

Insert into ..values

- syntax
- `INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);`
- `INSERT INTO table_name
VALUES (value1, value2, value3, ...);`

Modification of the Database – 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);
```

Example table

- CREATE TABLE Persons (
 Customerid int NOT NULL **AUTO_INCREMENT**,
 CustomerName varchar(20),
 ContactName varchar(20),
 Address, City varchar(20),
 PostalCode int(10),
 Country varchar(20),
 PRIMARY KEY (Customerid)
);

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|------------|----------------------|-----------------|--------------------------------|-----------|------------|---------|
| 89 | White Clover Markets | Karl Jablonski | 305 - 14th Ave. S. Suite 3B | Seattle | 98128 | USA |
| 90 | Wilman Kala | Matti Karttunen | Keskuskatu 45 | Helsinki | 21240 | Finland |
| 91 | Wolski | Zbyszek | ul. Filtrowa 68 | Walla | 01-012 | Poland |
| 92 | Cardinal | Tom B. Erichsen | Skagen 21 | Stavanger | 4006 | Norway |

- INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)
VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen , 21', 'Stavanger', '4006', 'Norway');

- Insert Data Only in Specified Columns
- **INSERT INTO Customers (CustomerName, City, Country)**
VALUES

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|------------|----------------------|-----------------|--------------------------------|-----------|------------|---------|
| 89 | White Clover Markets | Karl Jablonski | 305 - 14th Ave. S. Suite 3B | Seattle | 98128 | USA |
| 90 | Wilman Kala | Matti Karttunen | Keskuskatu 45 | Helsinki | 21240 | Finland |
| 91 | Wolski | Zbyszek | ul. Filtrowa 68 | Walla | 01-012 | Poland |
| 92 | Cardinal | null | null | Stavanger | null | Norway |



Insertion (Cont.)

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

student(SID, name, dept_name, tot_cred)

instructor(ID, name, dept_name, salary)

insert into student

select ID, name, dept_name, 0
from instructor

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

insert into student select * from student

might insert **an infinite number of tuples**, if the primary key constraint on *student* were absent.)

deletion

- `DELETE FROM table_name WHERE condition;`

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|------------|----------------------|-----------------|--------------------------------|-----------|------------|---------|
| 89 | White Clover Markets | Karl Jablonski | 305 - 14th Ave. S. Suite 3B | Seattle | 98128 | USA |
| 90 | Wilman Kala | Matti Karttunen | Keskuskatu 45 | Helsinki | 21240 | Finland |
| 91 | Wolski | Zbyszek | ul. Filtrowa 68 | Walla | 01-012 | Poland |
| 92 | Cardinal | Tom B. Erichsen | Skagen 21 | Stavanger | 4006 | Norway |

- `DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';`

Modification of the Database – Deletion

- Delete all instructors

```
delete from instructor
```

- Delete all instructors from the Finance department

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

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

```
delete from instructor
```

```
where dept_name in (select dept_name  
from department  
where building = 'Watson');
```

Deletion (Cont.)

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

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

updation

- The UPDATE statement is used to modify the existing records in a table.
- `UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;`
- `UPDATE Customers SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'
WHERE CustomerID = 1;`
- `UPDATE Customers SET ContactName='Juan';`

Modification of the Database – Updates

- Increase salaries of instructors whose salary is over \$100,000 by 3%, and all others receive a 5% raise

- Write two **update** statements:

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

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

- The order is important
 - Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- Same query as before but with case statement

```
update instructor  
    set salary = case  
        when salary <= 100000 then salary * 1.05  
        else salary * 1.03  
    end
```

The general form of the case statement is as follows.

```
CASE  
    WHEN condition1 THEN result1  
    WHEN condition2 THEN result2  
    WHEN conditionN THEN resultN  
    ELSE result  
END;
```

- The following SQL will order the customers by City. However, if City is NULL, then order by Country:
- ```
SELECT CustomerName, City, Country
FROM Customers
ORDER BY
(CASE
 WHEN City IS NULL THEN Country
 ELSE City
END);
```

# Updates with Scalar Subqueries

- Recompute and **update tot\_creds value** for all students  
set the *tot cred* attribute of each *student* tuple to the **sum of the credits** of courses **successfully completed** by the student

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

**End of Chapter 3**



# Chapter 4: Intermediate SQL

## part1- join to views

# Chapter 4: Intermediate SQL

## • Join Expressions

- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- 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

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

# Join operations – Example

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

- Relation *prereq*

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

- Observe that
  - prereq information is missing for CS-315 and
  - course information is missing for CS-437

# Natural or conditional Joins

- course **natural join** prereq

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

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



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

# Outer Join

- More generally, some tuples in either or both of the relations being joined may be “lost” in this way.
- An extension of the join operation that **avoids loss of information**.
- The **outer join** operation works in a manner similar to the join operations we have already studied, but preserve those tuples that would be lost in a join, by creating **tuples in the result containing null values**.
- 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.

- There are in fact three forms of outer join:
- The **left outer join** preserves tuples only in the relation named before (to the left of) the **left outer join** operation
- The **right outer join** preserves tuples only in the relation named after (to the right of) the **right outer join** operation.
- The **full outer join** preserves tuples in both relations

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

| <i>Join Conditions</i>         |
|--------------------------------|
| natural                        |
| on <predicate>                 |
| using $(A_1, A_1, \dots, A_n)$ |

# Left Outer Join

- course **natural left outer join** prereq

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

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

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

# Right Outer Join

- course **natural right outer join** prereq

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

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

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

# Full Outer Join

- course **natural full outer join** prereq

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

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

| 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    |

- “Display a list of all students in the Comp. Sci. department, along with the course sections, if any, that they have taken in Spring 2009; all course sections from Spring 2009 must be displayed, even if no student from the Comp. Sci. department has taken the course section.” This query can be written as:
- **select \* from (select \* from student where dept name= 'Comp. Sci') natural full outer join (select \* from takes where semester = 'Spring' and year = 2009);**

# Join...using

- To provide the benefit of natural join while avoiding the danger of equating attributes erroneously, SQL provides a form of the natural join construct that allows you to specify exactly which columns should be equated. This feature is illustrated by the following query:
- ***select name, title from (instructor natural join teaches) join course using (course id);***
- The operation **join . . . using** requires a list of attribute names to be specified. Both inputs must have attributes with the specified names.
- Consider the operation ***r1 join r2 using(A1, A2).***

# Join...on

- The **on** condition allows a general predicate over the relations being joined. This predicate is written like a **where** clause predicate except for the use of the keyword **on** rather than **where**. Like the **using** condition, the **on** condition appears at the end of the join expression.
- Consider the following query, which has a join expression containing the **on** condition.
- **select \*from student join takes on student.ID= takes.ID;**
- The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal.

- The following SQL statement will select all customers, and any orders they might have:
- ```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
LEFT OUTER JOIN Orders ON Customers.CustomerID =
Orders.CustomerID;
```

Inner join

- To distinguish normal joins from outer joins, normal joins are called **inner joins** in SQL. A join clause can thus specify **inner join** instead of **outer join** to specify that a normal join is to be used. The keyword **inner** is, however, optional.
- The default join type, when the **join** clause is used without the **outer** prefix is the **inner join**.

select * from student join takes using (ID);

is equivalent to:

select * from student inner join takes using (ID);

- The INNER JOIN keyword selects records that have matching values in both tables.
- ```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

# Joined Relations – Examples

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

| <i>course_id</i> | <i>title</i> | <i>dept_name</i> | <i>credits</i> | <i>prereq_id</i> | <i>course_id</i> |
|------------------|--------------|------------------|----------------|------------------|------------------|
| 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?
- Notice that we do not repeat those attributes that appear in the schemas of both relations; rather they appear only once.
- Notice also the order in which the attributes are listed: first the attributes common to the schemas of both relations, second those attributes unique to the schema of the first relation, and finally, those attributes unique to the schema of the second relation.

# 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    |

# 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.

# Does view occupy memory????

- Actually view is a stored select statement and its code is stored in data dictionary.
- It doesn't occupy space in the memory.
- whenever view is called, the stored select statement executes and fetches the data from the base table.
- If base table is deleted, view becomes invalid.

# View Definition

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

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

where <query expression> is any legal SQL expression. The view name is represented by v.

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

# Example Views

A view of instructors without their salary

```
create view faculty as
```

```
 select ID, name, dept_name
 from instructor
```

- Find all instructors in the Biology department

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

- **The attribute names of a view can be specified explicitly as follows:**

- To Create a view of department salary totals

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

# Views Defined Using Other Views

■ **create view** *physics\_fall\_2009* **as**  
    **select** *course.course\_id, sec\_id, building, room\_number*  
    **from** *course, section*  
    **where** *course.course\_id = section.course\_id*  
        **and** *course.dept\_name = 'Physics'*  
        **and** *section.semester = 'Fall'*  
        **and** *section.year = '2009';*

■ **create view** *physics\_fall\_2009\_watson* **as**  
    **select** *course\_id, room\_number*  
    **from** *physics\_fall\_2009*  
    **where** *building = 'Watson';*

# View Expansion

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

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

# Updating a view

- The difficulty is that a modification to the database expressed in terms of a view must be translated to a modification to the actual relations in the logical model of the database.
- **create view** *faculty* **as**  
**select** *ID, name, dept\_name*  
**from** *instructor*
- If a tuple is added as follows:
- **insert into** *faculty* **values** ('30765', 'Green', 'Music');
- This insertion must be represented by an **insertion into the relation** *instructor*, since *instructor* is the actual relation from which the database system constructs the view *faculty*. However, to insert a tuple into *instructor*, we must have some value for *salary*.

- There are two reasonable approaches to dealing with this insertion:
  - Reject the insertion, and return an error message to the user.
  - Insert a tuple ('30765', 'Green', 'Music', *null*) into the *instructor* relation.

# Update of a View

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

```
insert into faculty values ('30765', 'Green', 'Music');
```

This insertion must be represented by the insertion of the tuple

```
insert into instructor values ('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', 'AB5');`
  - If id does not exists
  - what if no department is in AB5?
- **Suppose we Insert into instructor the (69987, 'White', null, null)**
- *Insert in department(null, AB5, null)*
- *In the absence of common dept name this view update will fail*

# Updating main table through views

The screenshot shows the Oracle SQL Developer interface. The top menu bar has standard icons for file, edit, search, and database operations. Below it, a toolbar with various icons for different functions like execute, refresh, and save. The main window has two tabs: 'Worksheet' and 'Query Builder'. The 'Worksheet' tab is active and contains the following SQL query:

```
select * from dept;
```

Below the query is a 'Query Result' tab showing the output of the query. It has a toolbar with icons for execute, refresh, and save. The results are displayed in a table:

| DEPTNO | DNAME      | LOC      |
|--------|------------|----------|
| 1      | ACCOUNTING | NEW YORK |
| 2      | RESEARCH   | DALLAS   |
| 3      | SALES      | CHICAGO  |
| 4      | OPERATIONS | BOSTON   |

At the bottom of the result pane, it says 'All Rows Fetched: 4 in C'.

The screenshot shows the Oracle SQL Developer interface. The top menu bar has standard icons for file, edit, search, and database operations. Below it, a toolbar with various icons for different functions like execute, refresh, and save. The main window has two tabs: 'Worksheet' and 'Query Builder'. The 'Worksheet' tab is active and contains the following SQL script:

```
create view v_dept
as
select dname, loc from DEPT;
```

Below the script is a 'Script Output' tab showing the execution message:

view V\_DEPT created.

At the bottom of the output pane, it says 'Task completed in 0.727 seconds'.

Worksheet    Query Builder

```
select * from v_dept;
```

Script Output    Query Result

All Rows Fetched

| DNAME        | LOC      |
|--------------|----------|
| 1 ACCOUNTING | NEW YORK |
| 2 RESEARCH   | DALLAS   |
| 3 SALES      | CHICAGO  |
| 4 OPERATIONS | BOSTON   |

Start Page    SCOTT

Worksheet    Query Builder

```
update v_dept
set loc = 'USA'
where dname = 'ACCOUNTING';
```

Script Output    Query Result

Task completed in 0.002 seconds

1 rows updated.

Worksheet    Query Builder

```
select * from v_dept;
```

Query Result

All Rows Fetched:

| DNAME        | LOC     |
|--------------|---------|
| 1 ACCOUNTING | USA     |
| 2 RESEARCH   | DALLAS  |
| 3 SALES      | CHICAGO |
| 4 OPERATIONS | BOSTON  |

The screenshot shows the Oracle SQL Developer interface. At the top, there's a toolbar with various icons. Below it is a tab bar with 'Worksheet' and 'Query Builder' tabs, with 'Worksheet' selected. In the main area, a query is written in the worksheet:

```
select * from DEPT;
```

Below the worksheet, there are two tabs: 'Script Output' and 'Query Result'. The 'Query Result' tab is selected. It displays the results of the executed query:

|   | DEPTNO | DNAME      | LOC     |
|---|--------|------------|---------|
| 1 | 10     | ACCOUNTING | USA     |
| 2 | 20     | RESEARCH   | DALLAS  |
| 3 | 30     | SALES      | CHICAGO |
| 4 | 40     | OPERATIONS | BOSTON  |

At the bottom of the result pane, there are several icons: a red cursor, a printer, a refresh, a red X, and the word 'SQL'. To the right of these icons, it says 'All Rows Fetched: 4 in 0.00'.

- an SQL view is said to be **updatable** (that is, inserts, updates or deletes can be applied on the view) if the following conditions are all satisfied by the query defining the view:
  - The **from clause has only one** database **relation**.
  - The **select clause** contains only attribute names of the relation, and **does not have any expressions, aggregates, or distinct specification**.
  - Any attribute not listed in the **select clause** can be set to null
  - The **query does not have a group by or having clause**.

| <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         |
| 69987     | White       | <i>null</i>      | <i>null</i>   |

*instructor*

| <i>dept_name</i> | <i>building</i> | <i>budget</i> |
|------------------|-----------------|---------------|
| Biology          | Watson          | 90000         |
| Comp. Sci.       | Taylor          | 1000000       |
| Elec. Eng.       | Taylor          | 85000         |
| Finance          | Painter         | 120000        |
| History          | Painter         | 50000         |
| Music            | Packard         | 80000         |
| Physics          | Watson          | 70000         |
| <i>null</i>      | Taylor          | <i>null</i>   |

*department*

# with check option

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

- By default, SQL would allow the above update to proceed.
- However, views can be defined with a **with check option** clause at the end of the view definition; then, if a tuple inserted into the view does not satisfy the view's **where** clause condition, the insertion is rejected by the database system.
- Updates are similarly rejected if the new value does not satisfy the **where** clause conditions.

```
1 CREATE [REDACTED] VIEW vps AS
2 SELECT
3 employeeNumber,
4 lastName,
5 firstName,
6 jobTitle,
7 extension,
8 email,
9 officeCode,
10 reportsTo
11 FROM
12 employees
13 WHERE
14 jobTitle LIKE '%VP%'
15 WITH CHECK OPTION;
```

```
1 INSERT INTO vps(employeeNumber,firstname,lastname,jobtitle,extension,email,office
Code,reportsTo)
2 VALUES(1704,'John','Smith','IT Staff','x9112','johnsmith@classicmodelcars.com',1,
1703);
```

This time, MySQL rejected the insert and issued the following error message:

```
1 Error Code: 1369. CHECK OPTION failed 'classicmodels.vps'
```

# Materialized Views

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

Intermediate- sql-2

# Materialized Views

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

# Syntax of MV

```
Create materialized view View_Name
Build [Immediate/Deffered]
Refresh [Fast/Complete/Force]
on [Commit/Demand]
as Select
```

# example

- Create or Replace Materialized view MV\_Employee
- as
- Select E.Employee\_num,E.Employee\_name,D.Department\_Name
- from Employee E , Department D where E.Dept\_no=D.Dept\_no
- Refresh auto on commit select \* from Department;

## Difference Between Materialized View And View :

| View                                                                                                    | Materialized Views(Snapshots)                                                                               |
|---------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| 1.View is nothing but the logical structure of the table which will retrieve data from 1 or more table. | 1.Materialized views(Snapshots) are also logical structure but data is physically stored in database.       |
| 2.You need to have Create view privileges to create simple or complex view                              | 2.You need to have create materialized view 's privileges to create Materialized views                      |
| 3.Data access is not as fast as materialized views                                                      | 3.Data retrieval is fast as compare to simple view because data is accessed from directly physical location |
| 4.There are 2 types of views:<br><br>1.Simple View<br><br>2.Complex view                                | 4.There are following types of Materialized views:<br><br>1.Refresh on Auto<br><br>2.Refresh on demand      |
| 5.In Application level views are used to restrict data from database                                    | 5.Materialized Views are used in Data Warehousing.                                                          |

# Basics of Transactions

- A **transaction** consists of a sequence of query and/or update statements. The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed.
- Unit of work
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Transactions begin implicitly
- Ended by **commit work**: commits the current transaction; that is, it makes the updates performed by the transaction become permanent in the database
- **rollback work**: causes the current transaction to be rolled back; that is, it undoes all the updates performed by the SQL statements in the transaction.
- **(note: detail discussion on transaction will revisit later)**

# 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.
- FOR EXAMPLE:
  - 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 are usually identified as part of the database schema design process, and declared as part of the **create table** command used to create relations.
- However, integrity constraints can also be added to an existing relation by using the command **alter table *table-name* add *constraint***, where *constraint* can be any constraint on the relation.
- When such a command is executed, the system first ensures that the relation satisfies the specified constraint.
- If it does, the constraint is added to the relation; if not, the command is rejected.

## Integrity Constraints on a Single Relation

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

# Not Null and Unique Constraints

- **Primary Key Constraint:** When this constraint is associated with the column of a table it will not allow NULL values into the column and it will maintain unique values as part of the table.
- **not null**
  - Declare *name* and *budget* to be **not null**

*name varchar(20) not null*  
*budget numeric(12,2) not null*

- **unique** ( $A_1, A_2, \dots, A_m$ )
- Unique Constraint: When this constraint is associated with the column(s) of a table it will not allow us to store a repetition of data/values in the column, but Unique constraint allows ONE NULL value. More than one NULL value is also considered as repetition, hence it does not store a repetition of NULL values also.
  - The unique specification states that the attributes  $A_1, A_2, \dots, A_m$  form a **candidate key**.
  - **Candidate keys** are **permitted to be null** (in contrast to primary keys).
- A constraint that is added immediately next to the column definition is known as a **Column-level constraint**. All constraints that are added after the columns definition of the table is completed are known as **Table-level constraints**.

- CREATE TABLE employee
  - ( id number(5) PRIMARY KEY,
  - name char(20),
  - dept char(10),
  - age number(2),
  - salary number(10),
  - location char(10) UNIQUE
- A constraint that is added immediately next to the column definition is known as a Column-level constraint. All constraints that are added after the columns definition of the table is completed are known as Table-level constraints.

# The check clause

- When applied to a relation declaration, the clause **check( $P$ )** specifies a predicate  $P$  that must be satisfied by every tuple in a relation.
- **check ( $P$ ),** where  $P$  is a predicate

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

```
create table section (
 course_id varchar(8),
 sec_id varchar(8),
 semester varchar(6),
 year numeric(4,0),
 building varchar(15),
 room_number varchar(7),
 time_slot_id varchar(4),
 primary key (course_id, sec_id, semester, year),
 check (semester in ('Fall', 'Winter', 'Spring', 'Summer'))
);
```

# Referential Integrity

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

- When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation (that is, the transaction performing the update action is rolled back).
- However, a **foreign key** clause can specify that if a delete or update action on the referenced relation violates the constraint, then, instead of rejecting the action, the system must take steps to change the tuple in the referencing relation to restore the constraint

## Cascading Actions in Referential Integrity

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

- Because of the clause **on delete cascade** associated with the foreign-key declaration, **if a delete of a tuple in *department*** results in this referential-integrity constraint being violated, the system does not reject the delete.
- Instead, the **delete “cascades”** to the *course* relation, deleting the tuple that refers to the department that was deleted.
- Similarly, the system does not reject **an update** to a field referenced by the constraint if it violates the constraint; instead, the system updates the field *dept name* in the referencing tuples in *course* to the new value as well.
- alternative actions to cascade: **set null, set default**

```
CREATE TABLE buildings (
 building_no INT PRIMARY KEY AUTO_INCREMENT,
 building_name VARCHAR(255) NOT NULL,
 address VARCHAR(255) NOT NULL
);
```

**Step 2.** Create the `rooms` table:

```
CREATE TABLE rooms (
 room_no INT PRIMARY KEY AUTO_INCREMENT,
 room_name VARCHAR(255) NOT NULL,
 building_no INT NOT NULL,
 FOREIGN KEY (building_no)
 REFERENCES buildings (building_no)
 ON DELETE CASCADE
);
```

```
SELECT * FROM buildings;
```

|   | building_no | building_name    | address                        |
|---|-------------|------------------|--------------------------------|
| ▶ | 1           | ACME Headquaters | 3950 North 1st Street CA 95134 |
|   | 2           | ACME Sales       | 5000 North 1st Street CA 95134 |

```
SELECT * FROM rooms;
```

|   | room_no | room_name     | building_no |
|---|---------|---------------|-------------|
| ▶ | 1       | Amazon        | 1           |
|   | 2       | War Room      | 1           |
|   | 3       | Office of CEO | 1           |
|   | 4       | Marketing     | 2           |
|   | 5       | Showroom      | 2           |

```
DELETE FROM buildings
WHERE building_no = 2;
```

**Step 8.** Query data from rooms table:

```
SELECT * FROM rooms;
```

|   | room_no | room_name     | building_no |
|---|---------|---------------|-------------|
| ▶ | 1       | Amazon        | 1           |
|   | 2       | War Room      | 1           |
|   | 3       | Office of CEO | 1           |

# Integrity Constraint Violation During Transactions

- E.g.

```
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 (next slide)

# Complex check clause

- As defined by the SQL standard, the predicate in the **check** clause can be an arbitrary predicate, which can include a subquery.
- **check** (*time slot id* in (select *time slot id* from *time slot*))
- The **check** condition verifies 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.
- Thus, 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 (in this case, when a tuple is deleted or modified in relation *time slot*)

# Complex Check Clauses

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- Domain constraints and referential-integrity constraints are special forms of assertions
- **create assertion** <assertion-name> **check** <predicate>;

- *This SQL statement creates an assertion to demand that there's no more than a single president among the employees:*

```
create assertion AT_MOST_ONE_PRESIDENT as CHECK
((select count(*)
 from EMP e
 where e.JOB = 'PRESIDENT') <= 1
)
```

- *This SQL statement creates an assertion to demand that Boston based departments do not employ trainers:*

```
create assertion NO_TRAINERS_IN_BOSTON as CHECK
 (not exists
 (select 'trainer in Boston'
 from EMP e, DEPT d
 where e.DEPTNO = d.DEPTNO
 and e.JOB = 'TRAINER'
 and d.LOC = 'BOSTON')
)
```

- *This SQL statement creates an assertion to demand that vacation records cannot be outside of one's employment period:*

```
create assertion VACATION_DURING_EMPLOYMENT as CHECK
(not exists
```

```
 (select 'vacation outside employment'
 from EMP e
 ,EMP_VACATION ev
 where e.EMPNO = ev.EMPNO
 and (ev.FIRST_DATE < e.HIRE_DATE or
 ev.LAST_DATE > e.TERMINATION_DATE))
)
```

- Make sure that supervisee's salary is less than his Supervisor's salary

Create assertion [Emp\\_sal\\_chek](#)

```
check(not exists (select * from Employee E1, Employee E2
 where E1.Mgr_Id=E2.Emp_Id and
 E1.salary>E2.salary));
```

- **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.**

**create assertion credits earned constraint check**

```
(not exists (select ID from student
where tot cred <> (select sum(credits)
from takes natural join course
where student.ID=takes.ID
and grade is not null and grade<>'F'))
```

# 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

# User-Defined Types

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

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

- **create table** *department*  
*(dept\_name varchar (20),*  
*building varchar (15),*  
*budget Dollars);*

# Domains

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

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

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
- **create domain** *degree\_level* **varchar**(10)  
**constraint** *degree\_level\_test*  
**check** (**value in** ('Bachelors', 'Masters', 'Doctorate'));

# Large-Object Types

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

# Intermediate sql

Authorization

# Authorization

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

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

Forms of authorization to modify the **database schema**

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

# Authorization Specification in SQL

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

**grant** <privilege list>

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

- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

# Privileges in SQL

- **select**: allows **read access** to relation, or the ability to query using the view

- Example: grant users  $U_1$ ,  $U_2$ , and  $U_3$  **select** authorization on the *instructor* relation:

```
grant select on instructor to U_1 , U_2 , U_3
```

- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

# Revoking Authorization in SQL

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

**revoke** <privilege list>

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

- Example:

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

- <privilege-list> may be **all** to revoke all privileges the 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 grantors, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

# Roles

- **create role** *instructor*;
- **grant** *instructor* **to** Amit;
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles
  - **create role** *teaching\_assistant*
  - **grant** *teaching\_assistant* **to** *instructor*;
    - *Instructor* inherits all privileges of *teaching\_assistant*
- Chain of roles
  - **create role** *dean*;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;

# Authorization on Views

- **create view geo\_instructor as**  
**(select \***  
**from instructor**  
**where dept\_name = 'Geology');**
- **grant select on geo\_instructor to geo\_staff**
- Suppose that a *geo\_staff* member issues
  - **select \***  
**from geo\_instructor;**

Ex:

```
REVOKE privilege_name
ON object_name
FROM {user_name | PUBLIC | role_name}
```

# 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**;
- Etc. read Section 4.6 for more details we have omitted here.

# ADVANCED SQL

- PL/SQL
- PROCEDURES
- FUNCTIONS
- TRIGGERS
- cursors

## PL/SQL

- PL/SQL is Oracle's *procedural* language extension to SQL, the non-procedural relational database language.
- With PL/SQL, you can use SQL statements to **manipulate ORACLE data and the flow of control statements to process the data**. Moreover, you can declare constants and variables, define subprograms (procedures and functions), and trap runtime errors.
- Thus, PL/SQL combines the data manipulating power of SQL with the data processing power of procedural languages.

## DIFFERENCE BETWEEN PL/SQL AND SQL

- When a SQL statement is issued on the client computer, the request is made to the database on the server, and the result set is sent back to the client.
- As a result, a single SQL statement causes two trips on the network. If multiple SELECT statements are issued, the network traffic increase significantly very fast. For example, four SELECT statements cause eight network trips.
- If these statements are part of the PL/SQL block, they are sent to the server as a single unit. The SQL statements in this PL/SQL program are executed at the server and the result set is sent back as a single unit. There is still only one network trip made as is in case of a single SELECT statement.

## PL/SQL BLOCKS

- PL/SQL blocks can be divided into two groups:
  1. Named
  2. Anonymous.
- Named blocks are used when creating subroutines. These subroutines are procedures, functions, and packages.
- The subroutines can be stored in the database and referenced by their names later on.
- In addition, subroutines can be defined within the anonymous PL/SQL block.
- Anonymous PL/SQL blocks do not have names. As a result, they cannot be stored in the database and referenced later.

# PL/SQL BLOCK STRUCTURE

- PL/SQL blocks contain three sections
  1. Declare section
  2. Executable section and
  3. Exception-handling section.
- The executable section is the only mandatory section of the block.
- Both the declaration and exception-handling sections are optional.

# PL/SQL BLOCK STRUCTURE

- PL/SQL block has the following structure:

DECLARE

    Declaration statements

BEGIN

    Executable statements

EXCEPTION

    Exception-handling statements

END ;

## DECLARATION SECTION

- The *declaration section* is the first section of the PL/SQL block.
- It contains definitions of PL/SQL identifiers such as variables, constants, cursors and so on.
- Example

```
DECLARE
 v_first_name VARCHAR2(35);
 v_last_name VARCHAR2(35);
 v_counter NUMBER := 0 ;
```

## EXECUTABLE SECTION

- The executable section is the next section of the PL/SQL block.
- This section contains executable statements that allow you to manipulate the variables that have been declared in the declaration section.

```
BEGIN
 SELECT first_name, last_name
 FROM student
 WHERE student_id = 123 ;
 DBMS_OUTPUT.PUT_LINE
 ('Student name : ' || first_name || ' ' || last_name);
END;
```

# EXCEPTION-HANDLING SECTION

- The *exception-handling section* is the last section of the PL/SQL block.
- This section contains statements that are executed when a runtime error occurs within a block.
- Runtime errors occur while the program is running and cannot be detected by the PL/SQL compiler.

**EXCEPTION**

**WHEN NO\_DATA\_FOUND THEN**

**DBMS\_OUTPUT.PUT\_LINE**

**(‘ There is no student with student id 123 ’);**

**END;**

```
DECLARE
-- variable declaration
message varchar2(20) := 'Hello, World!';
BEGIN
/* * PL/SQL executable statement(s) */
dbms_output.put_line(message);

END; /
```

Hello World  
PL/SQL procedure successfully  
completed.

# PL/SQL EXAMPLE

DECLARE

```
v_first_name VARCHAR2(35);
v_last_name VARCHAR2(35);
```

BEGIN

```
SELECT first_name, last_name
INTO v_first_name, v_last_name
FROM student
WHERE student_id = 123;
DBMS_OUTPUT.PUT_LINE
('Student name: '||v_first_name||' '||v_last_name);
```

EXCEPTION

```
WHEN NO_DATA_FOUND THEN
 DBMS_OUTPUT.PUT_LINE
('There is no student with student id 123');
```

END;

# PL/SQL Program Units

- A PL/SQL unit is any one of the following –
- PL/SQL block
- Function
- Package
- Package body
- Procedure
- Trigger
- Type
- Type body

# Programmatic Control Constructs

- **If-Then-ElseIf-then-Else-End If :**

```
IF < condition> THEN
 < action >

ELSIF <condition> THEN
 < action >

ELSE
 < action >
END IF;
```

```
1 DECLARE
2 n_sales NUMBER := 300000;
3 n_commission NUMBER(10, 2) := 0;
4 BEGIN
5 IF n_sales > 200000 THEN
6 n_commission := n_sales * 0.1;
7 ELSE
8 n_commission := n_sales * 0.05;
9 END IF;
10 END;
```

```
1 DECLARE
2 n_sales NUMBER := 300000;
3 n_commission NUMBER(10, 2) := 0;
4 BEGIN
5 IF n_sales > 200000 THEN
6 n_commission := n_sales * 0.1;
7 ELSIF n_sales <= 200000 AND n_sales > 100000 THEN
8 n_commission := n_sales * 0.05;
9 ELSIF n_sales <= 100000 AND n_sales > 50000 THEN
10 n_commission := n_sales * 0.03;
11 ELSE
12 n_commission := n_sales * 0.02;
13 END IF;
14 END;
```

| S.No | Loop Type & Description                                                                                                                                                                                                                                    |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | <u>PL/SQL Basic LOOP</u> In this loop structure, sequence of statements is enclosed between the <b>LOOP</b> and the <b>END LOOP</b> statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop. |
| 2    | <u>PL/SQL WHILE LOOP</u> Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.                                                                                                |
| 3    | <u>PL/SQL FOR LOOP</u> Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.                                                                                                                            |
| 4    | <u>Nested loops in PL/SQL</u> You can use one or more loop inside any another basic loop, while, or for loop.                                                                                                                                              |

# BASIC LOOPS IN PL/SQL

- sequence of statements is enclosed between the LOOP and the END LOOP statements.
- At each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

Declare

Begin

Loop

    Statements

End loop

End;/

- <https://www.youtube.com/watch?v=AFx6QYcY1CU>
- Controlling the loops
  - EXIT
  - EXIT WHEN

# While and For

## **While Loop:**

```
WHILE <condition>
LOOP
 <Action>
END LOOP;
```

## Syntax

```
FOR counter IN initial_value .. final_value LOOP
 sequence_of_statements;
END LOOP;
```

```
DECLARE
 a number(2) := 10;
BEGIN
 WHILE a < 20 LOOP
 dbms_output.put_line('value of a: ' || a);
 a := a + 1;
 END LOOP;
END;
/
```

When the above code is executed at the SQL prompt, it produces

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

```
PL/SQL procedure successfully completed.
```

```
DECLARE
 a number(2);
BEGIN
 FOR a in 10 .. 20 LOOP
 dbms_output.put_line("value of a: " || a);
 END LOOP;
END;
/
```

When the above code is executed at the SQL prompt, it

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20
```

PL/SQL procedure successfully completed.

# Reverse FOR LOOP Statement

```
DECLARE
 a number(2) ;
BEGIN
 FOR a IN REVERSE 10 .. 20 LOOP
 dbms_output.put_line('value of a: ' || a);
 END LOOP;
END;
/
```

When the above code is executed at the SQL prompt,

```
value of a: 20
value of a: 19
value of a: 18
value of a: 17
value of a: 16
value of a: 15
value of a: 14
value of a: 13
value of a: 12
value of a: 11
value of a: 10
```

# Case Statement

```
CASE [TRUE | selector]
 WHEN expression1 THEN
 sequence_of_statements1;
 WHEN expression2 THEN
 sequence_of_statements2;
 ...
 WHEN expressionN THEN
 sequence_of_statementsN;
 [ELSE sequence_of_statementsN+1;]
END CASE [label_name];
```

```
// Case statement
```

```
DECLARE
```

```
 grade CHAR(1);
```

```
BEGIN
```

```
 grade := 'B';
```

```
CASE grade
```

```
 WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
```

```
 WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
```

```
 WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
```

```
 ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
```

```
END CASE;
```

```

DECLARE
 n_pct employees.commission_pct%TYPE;
 v_eval varchar2(10);
 n_emp_id employees.employee_id%TYPE := 145;
BEGIN
 -- get commission percentage
 SELECT commission_pct
 INTO n_pct
 FROM employees
 WHERE employee_id = n_emp_id;

 -- evaluate commission percentage
 CASE n_pct
 WHEN 0 THEN
 v_eval := 'N/A';
 WHEN 0.1 THEN
 v_eval := 'Low';
 WHEN 0.4 THEN
 v_eval := 'High';
 ELSE
 v_eval := 'Fair';
 END CASE;
 -- print commission evaluation
 DBMS_OUTPUT.PUT_LINE('Employee ' || n_emp_id ||
 ' commission ' || TO_CHAR(n_pct) ||
 ' which is ' || v_eval);
END;

```

# Procedures and Functions in PL/SQL

- PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms –
- **Functions** – These subprograms return a single value; mainly used to compute and return a value.
- **Procedures** – These subprograms do not return a value directly; mainly used to perform an action.

# Procedure

- A Procedure is a subprogram unit that consists of a group of PL/SQL statements. Each procedure in Oracle has its own unique name by which it can be referred. This subprogram unit is stored as a database object.
- A stored procedure or in simple term, a proc is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages.

| S.No | Parts & Description                                                                                                                                                                                                                                                                                                                                             |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | <p><b>Declarative Part</b></p> <p>It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.</p> |
| 2    | <p><b>Executable Part</b></p> <p>This is a mandatory part and contains statements that perform the designated action.</p>                                                                                                                                                                                                                                       |
| 3    | <p><b>Exception-handling</b></p> <p>This is again an optional part. It contains the code that handles run-time errors.</p>                                                                                                                                                                                                                                      |

# PROCEDURES

- The syntax for creating a procedure is as follows:

**CREATE OR REPLACE PROCEDURE**

name

(<parameter| IN/OUT <datatype>)

[AS | IS]

[local declarations]

**BEGIN**

executable statements

**[EXCEPTION**

exception handlers]

**END [name];**

- CREATE PROCEDURE instructs the compiler to create new procedure. Keyword 'OR REPLACE' instructs the compiler to replace the existing procedure (if any) with the current one.
- Procedure name should be unique.
- Keyword 'IS' will be used, when the procedure is nested into some other blocks. If the procedure is standalone then 'AS' will be used. Other than this coding standard, both have the same meaning

- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
dbms_output.line("Hello world");
END;
```

---

When the above code is executed using the SQL prompt, it will produce the following result – PROCEDURE CREATED  
EXECUTE greetings;

The procedure can also be called from another PL/SQL block –

```
BEGIN
greetings;
END; /
```

## Deleting a Standalone Procedure

A standalone procedure is deleted with the **DROP PROCEDURE** statement.

Syntax for deleting a procedure is

```
DROP PROCEDURE procedure-name;
```

# Parameter

- The **parameter is variable or placeholder** of any valid PL/SQL datatype through which the PL/SQL subprogram exchange the values with the main code. This parameter allows to give input to the subprograms and to extract from these subprograms.
- These parameters should be **defined** along with the subprograms at the **time of creation**.
- These parameters are **included in the calling statement** of these subprograms to interact the values with the subprograms.
- The datatype of the parameter in the subprogram and the calling statement **should be same**.
- The **size of the datatype should not mention at the time of parameter declaration**, as the size is dynamic for this type.

# PARAMETERS

- Parameters are the means to pass values to and from the calling environment to the server.
- These are the values that will be processed or returned via the execution of the procedure.
  - There are three types of parameters:  
**IN, OUT, and IN OUT.**
- IN passes value into the procedure, OUT passes back from the procedure and INOUT does both.

# Types of Parameters

| Mode   | Description                                | Usage                                                                                                                        |
|--------|--------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| IN     | Passes a value into the program            | Read only value<br>Constants, literals, expressions<br>Cannot be changed within program<br>Default mode                      |
| OUT    | Passes a value back from the program       | Write only value<br>Cannot assign default values<br>Has to be a variable<br>Value assigned only if the program is successful |
| IN OUT | Passes values in and also send values back | Has to be a variable<br>Value will be read and then written                                                                  |

| S.No | Parameter Mode & Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | <p><b>IN</b></p> <p>An IN parameter lets you pass a value to the subprogram. <b>It is a read-only parameter.</b> Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. <b>It is the default mode of parameter passing. Parameters are passed by reference.</b></p> |
| 2    | <p><b>OUT</b></p> <p>An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. <b>The actual parameter must be variable and it is passed by value.</b></p>                                                                                                                                                                                                                     |
| 3    | <p><b>IN OUT</b></p> <p>An <b>IN OUT</b> parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read.</p> <p>The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. <b>Actual parameter is passed by value.</b></p>                                                                                             |

create table named emp have two column id and salary with number datatype.

```
CREATE OR REPLACE PROCEDURE emp_insert(id IN NUMBER, sal IN NUMBER) AS
BEGIN
 INSERT INTO emp VALUES(id, sal);
 DBMS_OUTPUT.PUT_LINE('VALUE INSERTED.');
END;
/
```

Procedure created.

```
SET SERVEROUTPUT ON;
EXECUTE emp_insert(101,2000);
procedure successfully completed.
```

// Insert a person information into a PERSON table provided his information does not exist already.

```
CREATE OR REPLACE PROCEDURE insertPerson (id IN VARCHAR, DOB IN DATE,
fname IN VARCHAR, lname IN VARCHAR)
```

IS

```
counter INTEGER; --declaration part
```

```
BEGIN
```

```
SELECT COUNT(*) INTO counter FROM person p WHERE p.pid = id;
```

```
IF (counter > 0) THEN
```

```
-- person with the given pid already exists
```

```
DBMS_OUTPUT.PUT_LINE('WARNING Inserting person: person with pid ' ||
id || 'already exists!');
```

```
ELSE
```

```
INSERT INTO person VALUES (id, DOB, fname, lname);
```

```
DBMS_OUTPUT.PUT_LINE('Person with pid ' || id || ' is inserted.');
```

```
END IF;
```

```
END;
```

```
/
```

// Procedure calls another procedure

```
CREATE OR REPLACE PROCEDURE insertFaculty (pid IN VARCHAR, DOB IN DATE, fname IN VARCHAR, lname IN VARCHAR, rank IN VARCHAR, dept IN VARCHAR) IS
```

```
BEGIN
```

```
 insertPerson (pid, DOB, fname, lname); // User defined Procedure
```

```
 insert into facultyEDB values(pid, rank, dept);
```

```
 DBMS_OUTPUT.PUT_LINE('Faculty with pid ' || pid || ' is inserted.');
```

```
END insertFaculty;
```

```
EXECUTE insertFaculty('121-11-1111', '21-OCT-1961', 'Susan', 'Urban', 'Emeritus', 'CSE');
```

```
-- from sql prompt
```

# Example

- In order to **execute a procedure** use the following syntax:

EXECUTE Procedure\_name;

Or

EXEC Procedure\_name;

SQL> EXECUTE insertPerson ('p1', '10-10-2000', 'John', 'Smith');

- To see the o/p on the screen use following command

**SET SERVEROUTPUT ON**

# FUNCTIONS

- Functions are a type of stored code and are very similar to procedures.
- The significant difference is that a function is a PL/SQL **block that *returns* a single value.**
- Functions can accept one, many, or no parameters, but a function **must have a return clause** in the executable section of the function.
- The datatype of the return value must be declared in the header of the function.
- A function is not a stand-alone executable in the way that a procedure is: **It must be used in some context.**
- A function has output that needs to be assigned to a variable, or it can be used in a **SELECT** statement.

# FUNCTIONS

- The function does not necessarily have to have any parameters, but it must have a RETURN value declared in the header, and it must return values for all the varying possible execution streams.
- The RETURN statement does not have to appear as the last line of the main execution section, and there **may be more than one RETURN statement** (there should be a RETURN statement for each exception).

# FUNCTIONS

- The syntax for creating a function is as follows:

```
CREATE [OR REPLACE] FUNCTION function_name
 (parameter list)
RETURN datatype
IS
BEGIN
 <body>
 RETURN (return_value);
END;
```

```
Select * from customers;
```

| ID | NAME     | AGE | ADDRESS   | SALARY  |
|----|----------|-----|-----------|---------|
| 1  | Ramesh   | 32  | Ahmedabad | 2000.00 |
| 2  | Khilan   | 25  | Delhi     | 1500.00 |
| 3  | kaushik  | 23  | Kota      | 2000.00 |
| 4  | Chaitali | 25  | Mumbai    | 6500.00 |
| 5  | Hardik   | 27  | Bhopal    | 8500.00 |
| 6  | Komal    | 22  | MP        | 4500.00 |

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
 total number(2) := 0;
BEGIN
 SELECT count(*) into total
 FROM customers;

 RETURN total;
END;
/
```

```
DECLARE
 c number(2);
BEGIN
 c := totalCustomers();
 dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Total no. of Customers: 6
```

```
PL/SQL procedure successfully completed.
```

```
1. CREATE OR REPLACE FUNCTION welcome_msg_func (p_name IN VARCHAR2)
2. RETURN VARCHAR2
3. IS
4. BEGIN
5. RETURN ('Welcome '|| p_name);
6. END;
7. /
```

**Output:**

Function created

*Function created*

```
8. DECLARE
9. lv_msg VARCHAR2(250);
10. BEGIN
11. lv_msg := welcome_msg_func ('Guru99');
12. dbms_output.put_line(lv_msg);
13. END;
```

*Calling function with  
'Guru99' as parameter*

**Output:**

Welcome Guru99

## Example

```
CREATE OR REPLACE FUNCTION show_description (i_course_no IN number)
RETURN varchar2
IS
 v_description varchar2(50);
BEGIN
 SELECT description INTO v_description
 FROM course WHERE course_no = i_course_no;
 RETURN v_description;
EXCEPTION
 WHEN NO_DATA_FOUND THEN
 RETURN('The Course is not in the database');
 WHEN OTHERS THEN
 RETURN('Error in running show_description');
END;
```

# Making Use Of Functions

- In a **anonymous block**

```
DECLARE
 v_description VARCHAR2(50);
BEGIN
 v_description := show_description(&sv_cnumber);
 DBMS_OUTPUT.PUT_LINE(v_description);
END;
```

- In a **SQL statement**

```
SELECT course_no, show_description(course_no)
 FROM course;
```

- SET SERVEROUTPUT ON

# TRIGGERS & CURSORS

# Triggers

- An SQL trigger is a mechanism that automatically executes a specified PL/SQL block when a triggering event occurs on a table.
- The triggering event may be one of insert, delete, or update.
- The trigger is associated with a database table and is fired when the triggering event takes place on the table.

## TRIGGERS

- You can associate up to 12 database triggers with a given table.
- A database trigger has **three parts**:  
a **triggering event**, an **optional trigger constraint**,  
and a **trigger action**.
- When an event occurs, a database trigger is fired,  
and an predefined PL/SQL block will perform the  
necessary action.

# Triggers

```
create [or replace] trigger trigger-name
{before | after}
{delete | insert | update [of column [, column] ...]}
ON table-name
[[referencing {old [as] <old> [new [as] <new>]
| new [as] <new> [old [as] <old>]]
for each row
[when (condition)]]
pl/sql_block
```

# TRIGGERS

The trigger\_name references the name of the trigger.

BEFORE or AFTER specify when the trigger is fired (before or after the triggering event).

The triggering\_event references a DML statement issued against the table (e.g., INSERT, DELETE, UPDATE).

The table\_name is the name of the table associated with the trigger.

The clause, FOR EACH ROW, specifies a trigger is a row trigger and fires once for each modified row.

Bear in mind that if you drop a table, all the associated triggers for the table are dropped as well.

# Triggers

- **referencing** specifies correlation names that can be used to refer to the **old and new** values of the row components that are being affected by the trigger
- **for each row** designates the trigger to be a row trigger, i.e., the trigger is fired once for each row that is affected by the triggering event and meets the optional trigger constraint defined in the when clause.
- **when** specifies the trigger restriction.

## example

```
mysql> desc Student;
```

| Field | Type        | Null | Key | Default |
|-------|-------------|------|-----|---------|
| tid   | int(4)      | NO   | PRI | NULL    |
| name  | varchar(30) | YES  |     | NULL    |
| subj1 | int(2)      | YES  |     | NULL    |
| subj2 | int(2)      | YES  |     | NULL    |
| subj3 | int(2)      | YES  |     | NULL    |
| total | int(3)      | YES  |     | NULL    |
| per   | int(3)      | YES  |     | NULL    |

```
create trigger stud_marks
before INSERT
on
Student
for each row
set Student.total = Student.subj1 + Student.subj2 + Student.subj3, Student.p
```

```
mysql> insert into Student values(0, "ABCDE", 20, 20, 20, 0, 0);
Query OK, 1 row affected (0.09 sec)

mysql> select * from Student;
+----+-----+-----+-----+-----+-----+
| tid | name | subj1 | subj2 | subj3 | total | per |
+----+-----+-----+-----+-----+-----+
| 100 | ABCDE | 20 | 20 | 20 | 60 | 36 |
+----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

## TYPES OF TRIGGERS

- A trigger may be a **ROW** or **STATEMENT** type.
- If the statement **FOR EACH ROW** is present in the **CREATE TRIGGER** clause of a trigger, the trigger is a row trigger. A **row trigger** is fired for each row affected by an triggering statement.
- A **statement trigger**, however, is **fired only once** for the triggering **statement**, regardless of the number of rows affected by the triggering statement

- Suppose, you want to restrict users to update credit of customers from 28th to 31st of every month so that you can close the financial month.

To enforce this rule, you can use this statement-level trigger:

```
1 CREATE OR REPLACE TRIGGER customers_credit_trg
2 BEFORE UPDATE OF credit_limit
3 ON customers
4 DECLARE
5 l_day_of_month NUMBER;
6 BEGIN
7 -- determine the transaction type
8 l_day_of_month := EXTRACT(DAY FROM sysdate);
9
10 IF l_day_of_month BETWEEN 28 AND 31 THEN
11 raise_application_error(-20100,'Cannot update customer credit from 28th to 31s
t');
12 END IF;
13 END;
```

# TYPES OF TRIGGERS

## Example 2: statement trigger

```
CREATE OR REPLACE TRIGGER mytrig1
BEFORE DELETE OR INSERT OR UPDATE ON employee
BEGIN
IF (TO_CHAR(SYSDATE, 'day') IN ('sat', 'sun')) OR
(TO_CHAR(SYSDATE,'hh:mi') NOT BETWEEN '08:30' AND '18:30')

THEN RAISE_APPLICATION_ERROR(-20500, 'table is secured');

END IF;
END;
/
```

# Raise\_application\_error

- Used inside trigger
- Purpose:
  - output an error message and
  - immediately **stop** the event that fired the trigger
  - For example, data insertion
- Can include variables/trigger values, see previous slide
- E.g.  
`RAISE_APPLICATION_ERROR(-20000, 'trigger violated')`

↑  
label

↑  
Message text

- ROW LEVEL TRIGGER allows to track old and new values
- When a DML statement changes a column, **the old and new values are visible** to the executing code
- This is done by prefixing the table column with :old or :new
  - :new** is useful for **INSERT** and **UPDATE**
  - :old** is useful for **DELETE** and **UPDATE**

Eg    **:old.emp\_salary**        **:new.emp\_salary**

- triggers may fire other triggers in which case they are **CASCADING**. Try not to create too many interdependencies with triggers!

:OLD.column\_name

:NEW.column\_name

IF :NEW.credit\_limit > :OLD.credit\_limit THEN

--few statements

ENDIF

## Example: ROW Trigger

```
CREATE OR REPLACE TRIGGER mytrig2
AFTER DELETE OR INSERT OR UPDATE ON employee
FOR EACH ROW
BEGIN
IF DELETING THEN
 INSERT INTO xemployee(emp_ssn, emp_last_name, emp_first_name, del_date)
 VALUES (:old.emp_ssn, :old.emp_last_name, :old.emp_first_name, sysdate);

ELSIF INSERTING THEN
 INSERT INTO nemployee(emp_ssn, emp_last_name, emp_first_name, add_date)
 VALUES (:new.emp_ssn, :new.emp_last_name, :new.emp_first_name, sysdate);

ELSIF UPDATING('emp_salary') THEN
 INSERT INTO cemployee(emp_ssn, oldsalary, newsalary, up_date)
 VALUES (:old.emp_ssn, :old.emp_salary, :new.emp_salary, sysdate);

END IF;
END;
/
```

// If person's department changes and earlier he is the chairperson for that department, then for that department chair is set to NULL in department table.

CREATE OR REPLACE TRIGGER faculty\_after\_update\_row

AFTER UPDATE ON facultyEDB // Here facultyEDB has dept and pid attributes.

FOR EACH ROW

BEGIN

IF UPDATING ('dept') AND :old.dept <> :new.dept

THEN UPDATE department SET chair = NULL WHERE chair = :old.pid;

---

END IF;

END; /

**// When faculty is deleted from a faculty table , delete his  
information from Person table**

```
CREATE OR REPLACE TRIGGER faculty_after_delete_row
AFTER DELETE ON facultyEDB
FOR EACH ROW BEGIN
 DELETE FROM person WHERE pid = :old.pid;
END; /
```

# ENABLING, DISABLING, DROPPING TRIGGERS

SQL>ALTER TRIGGER trigger\_name **DISABLE**;

SQL>ALTER TABLE table\_name DISABLE ALL TRIGGERS;

**To enable a trigger, which is disabled, we can use the following syntax:**

SQL>ALTER TABLE table\_name **ENABLE** trigger\_name;

**All triggers can be enabled for a specific table by using the following command**

SQL> ALTER TABLE table\_name ENABLE ALL TRIGGERS;

SQL> **DROP** TRIGGER trigger\_name

# Referential integrity trigger example

```
UPDATE author SET aID='9' WHERE aID='5';
```

ERROR at line 1:

ORA-02292: integrity constraint  
(MCTPL.BOOK\_FK) violated - child record found

Without trigger -  
referential integrity prevents  
changes to aID

```
CREATE OR REPLACE TRIGGER author_trg
```

```
AFTER UPDATE OF aID ON author
```

```
FOR EACH ROW
```

```
BEGIN
```

```
UPDATE Book SET authID = :new.aID WHERE authID= :old.aID;
```

```
END;
```

Trigger  
automatically applies  
corresponding changes  
to aID in child table

```
UPDATE author SET aID='9' WHERE aID='5';
```

1 row updated.

With trigger -  
changes to aID are now allowed!

## Example trigger: Incorporating business rules

To raise a message if more than 1000 books from the same publisher has been added to the “Book” table.

```
CREATE OR REPLACE TRIGGER publish_trg
BEFORE INSERT OR UPDATE ON book
FOR EACH ROW

DECLARE
 how_many NUMBER;
BEGIN
 SELECT COUNT(*) INTO how_many FROM book
 WHERE publisher = :new.publisher;
 IF how_many >= 1000 then
 Raise_application_error(-20000, 'Publisher' ||
 :new.publisher || 'already has 1000 books');
 END IF;
END;
```

# Compilation errors

- When you create a trigger, Oracle responds
  - "Trigger created" or
  - "Warning: Trigger created with compilation errors."
- Type in the command

**SHOW ERRORS**

on its own to make the error messages visible

# **Database Access Using Cursors**

# Cursors

- A pointer to the context area
- Cursor Types:
  - Explicit: user-defined
  - Implicit: system-defined

# Types of Cursor

- **Implicit Cursors:**

Implicit Cursors are also known as Default Cursors of SQL SERVER. These Cursors are allocated by SQL SERVER when the user performs DML operations.

- **Explicit Cursors :**

Explicit Cursors are Created by Users whenever the user requires them. Explicit Cursors are used for Fetching data from Table in Row-By-Row Manner.

# Explicit Cursors

- To use explicit cursors...
  - Declare the cursor
  - Open the cursor
  - Fetch the results into PL/SQL variables
  - Close the cursor

- [https://www.youtube.com/watch?v= snAMqCBitg](https://www.youtube.com/watch?v=snAMqCBitg)

# Declaring Cursors

DECLARE

```
v_StudentID students.id%TYPE;
v_FirstName students.first_name%TYPE;
v_LastName students.last_name%TYPE;
```

**CURSOR c\_HistoryStudents IS**

```
SELECT id, first_name, last_name
FROM students
WHERE major = 'History';
```

BEGIN

-- open cursor, fetch records & then close cursor here

END;

/

# OPEN Cursor

DECLARE

```
v_StudentIDstudents.id%TYPE;
v_FirstNamestudents.first_name%TYPE;
v_LastNamestudents.last_name%TYPE;
```

**CURSOR** c\_HistoryStudents **IS**

```
SELECT id, first_name, last_name
FROM students
WHERE major = 'History';
```

BEGIN

OPEN c\_HistoryStudents;

-- fetch records & then close cursor here

END;

/

# FETCH Records

```
DECLARE
 v_StudentID students.id%TYPE;
 v_FirstName students.first_name%TYPE;
 v_LastName students.last_name%TYPE;
 CURSOR c_HistoryStudents IS
 SELECT id, first_name, last_name
 FROM students
 WHERE major = 'History';
BEGIN
 OPEN c_HistoryStudents;
 LOOP
 FETCH c_HistoryStudents INTO v_StudentID, v_FirstName, v_LastName;
 EXIT WHEN c_HistoryStudents%NOTFOUND;
 -- do something with the values that are now in the variables
 END LOOP;
 CLOSE c_HistoryStudents;
END;/
```

# Explicit Cursor Attributes

Obtain status information about a cursor.

| Attribute        | Type    | Description                                                                       |
|------------------|---------|-----------------------------------------------------------------------------------|
| <b>%ISOPEN</b>   | Boolean | Evaluates to TRUE if the cursor is open.                                          |
| <b>%NOTFOUND</b> | Boolean | Evaluates to TRUE if the most recent fetch does not return a row.                 |
| <b>%FOUND</b>    | Boolean | Evaluates to TRUE if the most recent fetch returns a row; complement of %NOTFOUND |
| <b>%ROWCOUNT</b> | Number  | Evaluates to the total number of rows returned so far.                            |

# Implicit cursor

```
DECLARE
 total_rows number(2);
BEGIN
 UPDATE customers
 SET salary = salary + 500;
 IF sql%notfound THEN
 dbms_output.put_line('no customers selected');
 ELSIF sql%found THEN
 total_rows := sql%rowcount;
 dbms_output.put_line(total_rows || ' customers selected ');
 END IF;
END;
/
```

# %type and % rowtype

- The %TYPE attribute, used in PL/SQL variable and parameter declarations, is supported by the data server. Use of this attribute ensures that type compatibility between table columns and PL/SQL variables is maintained.

## %TYPE

Table: Agents (aid, fname, lname, city, percent )

DECLARE

percent\_val      agents.percent%TYPE;

BEGIN

SELECT percent INTO percent\_val FROM agents WHERE aid = 'a02';

IF percent\_val > 50 THEN

    INSERT INTO agents (aid, fname, lname, city) VALUES (  
        'a07', 'John', 'Corpus');

END IF;

END; /

# %ROWTYPE

DECLARE

dept\_rec departments%ROWTYPE;

BEGIN

SELECT \* INTO dept\_rec -- Can access only one tuple

FROM departments

WHERE department\_id = 30 ;

DBMS\_OUTPUT.PUT\_LINE

(‘Dept infor: ‘ || dept\_rec.Name || ‘|| dept\_rec.Head\_Name) ;

END;

/

- Write a program in PL/SQL to display a cursor based detail information of employees from employees table.

```
DECLARE
CURSOR z_emp_info IS
SELECT employee_id, first_name, last_name, salary FROM employees;
r_emp_info z_emp_info%ROWTYPE;
BEGIN
OPEN z_emp_info;
LOOP FETCH z_emp_info
INTO r_emp_info;
EXIT WHEN z_emp_info%NOTFOUND;
dbms_output.Put_line('Employees Information:: ' ||' ID: '
||r_emp_info.employee_id ||' Name: ' ||r_emp_info.first_name ||'
||r_emp_info.last_name);
END LOOP;
dbms_output.Put_line('Total number of rows : '
||z_emp_info%rowcount); CLOSE z_emp_info;
END; /
```

- Write a program in PL/SQL to create an implicit cursor with for loop.

```
BEGIN
FOR emprec IN
(SELECT department_name, d.department_id, first_name, last_name,
job_id, salary FROM departments d join employees e ON
e.department_id = d.department_id WHERE job_id = 'ST_CLERK' AND
salary > 3000) LOOP
dbms_output.Put_line('Name: ' || emprec.first_name ||'
|| emprec.last_name || chr(9) || ' Department: '
|| emprec.department_name || chr(9) || ' Department ID: '
|| emprec.department_id || chr(9) || ' Job ID: ' || emprec.job_id || chr(9)
|| ' Salary: ' || emprec.salary);
Exit when sql%notfound
END LOOP;
END; /
```

Consider an employee database with two relations

*employee (employee name, street, city)*

*works (employee name, company name, salary)*

where the primary keys are underlined. Write a query to find companies whose employees earn a higher salary, on average, than the average salary at “First Bank Corporation”.



Das Bild kann zurzeit nicht angezeigt werden.

# Chapter 7: Entity-Relationship Model

Database System Concepts, 6<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

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



# Chapter 7: Entity-Relationship Model

- n Design Process
- n Modeling
- n Constraints
- n E-R Diagram
- n Weak Entity Sets
- n Extended E-R Features



# Modeling

- n A *database* can be modeled as:
  - | a collection of entities,
  - | relationship among entities.
- n An **entity** is an object that exists and is distinguishable from other objects.
  - | Example: person, company, event, plant, Course, Department
- n Entities have **attributes**
  - | Example: people have *names* and *addresses*
- n An **entity set** is a **set of entities of the same type** that share the same properties.
  - | Example: set of all persons, companies, trees, holidays



# Entity Sets *instructor* and *student*

instructor\_ID instructor\_name

|       |            |
|-------|------------|
| 76766 | Crick      |
| 45565 | Katz       |
| 10101 | Srinivasan |
| 98345 | Kim        |
| 76543 | Singh      |
| 22222 | Einstein   |

*instructor*

student-ID student\_name

|       |         |
|-------|---------|
| 98988 | Tanaka  |
| 12345 | Shankar |
| 00128 | Zhang   |
| 76543 | Brown   |
| 76653 | Aoi     |
| 23121 | Chavez  |
| 44553 | Peltier |

*student*



# Relationship Sets

- n A **relationship** is an association among several entities

Example:

44553 (Austin)  
*student entity*

*advisor*

relationship set

22222 (Einstein)  
*instructor entity*

- n A **relationship set** is a mathematical relation among  $n \geq 2$  entities, each taken from entity sets

$$\{(e_1, e_2, \dots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$

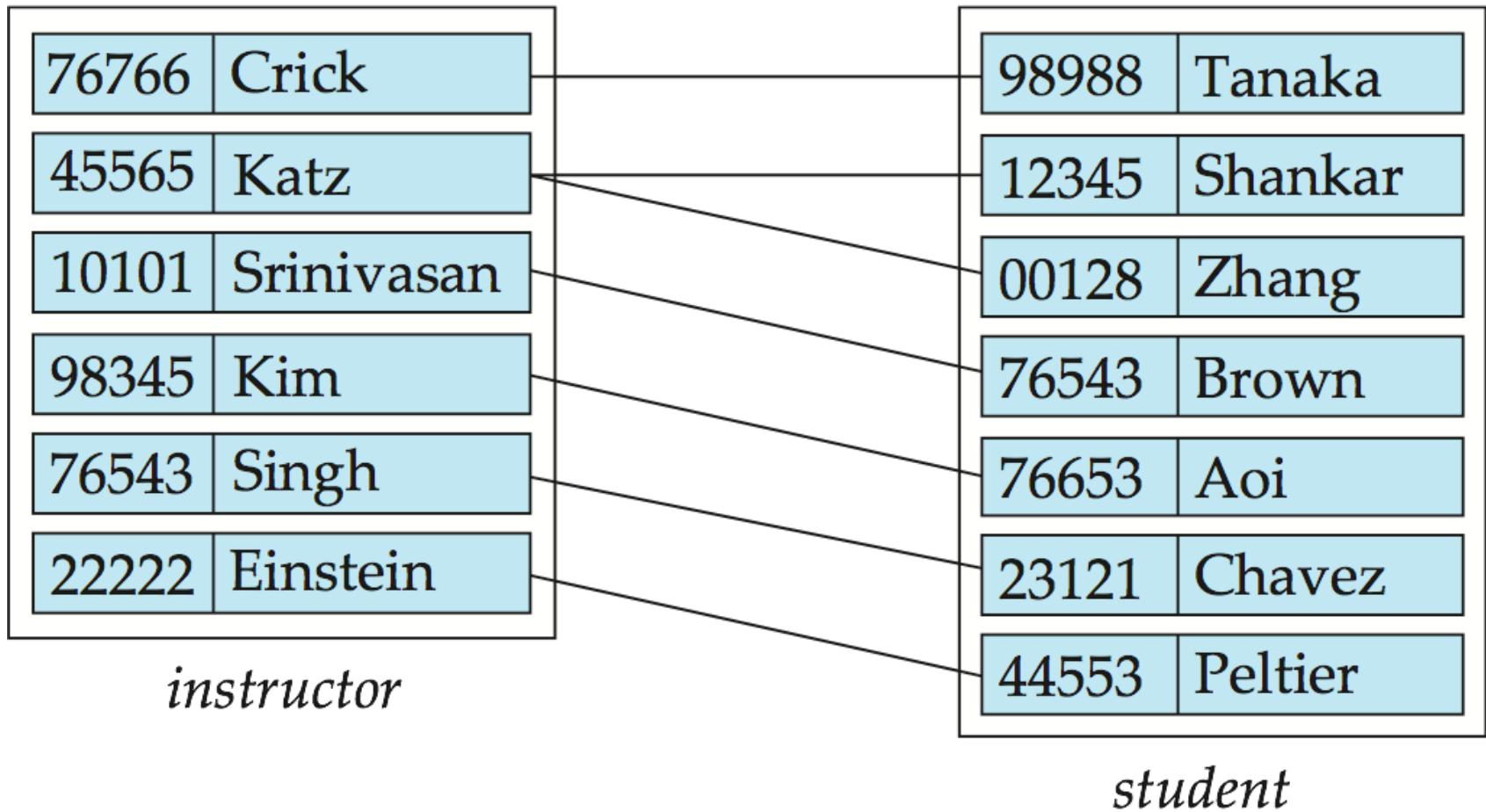
where  $(e_1, e_2, \dots, e_n)$  is a relationship

| Example:

$(44553, 22222) \in \text{advisor}$



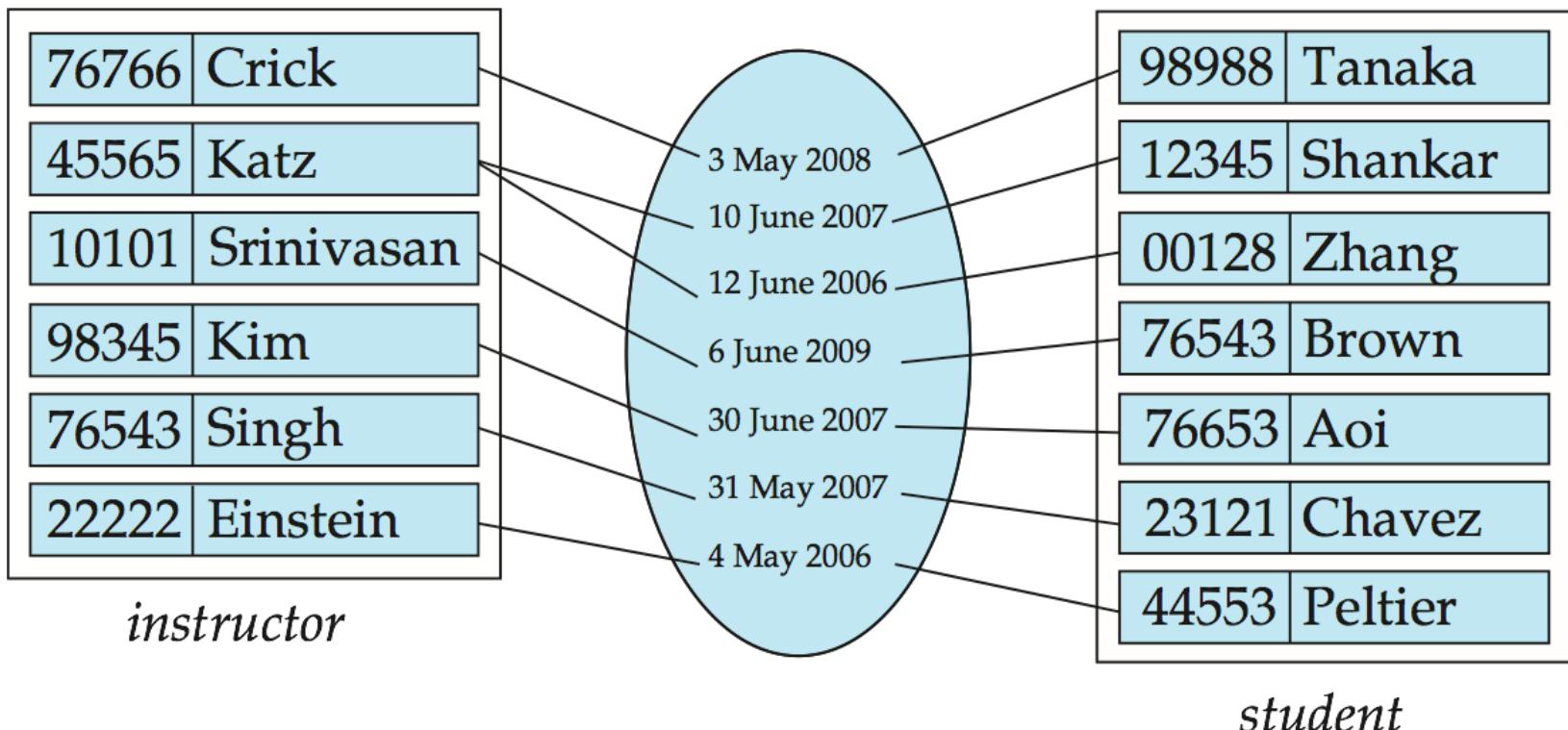
# Relationship Set *advisor*





# Relationship Sets (Cont.)

- n A **descriptive attribute** can also be **property** of a relationship set.
- n For instance, the *advisor* relationship set between entity sets *instructor* and *student* may have the **attribute date** which tracks when the student started being associated with the advisor





# Degree of a Relationship Set

- n **binary relationship**
  - | involve **two entity sets** (or degree two).
  - | most relationship sets in a database system are binary.
- n Relationships between more than two entity sets are rare. **Most relationships are binary**. (More on this later.)
  - ▶ Example: *students* work on research *projects* under the guidance of an *instructor*.
  - ▶ relationship *proj\_guide* is a **ternary relationship** between *instructor*, *student*, and *project*



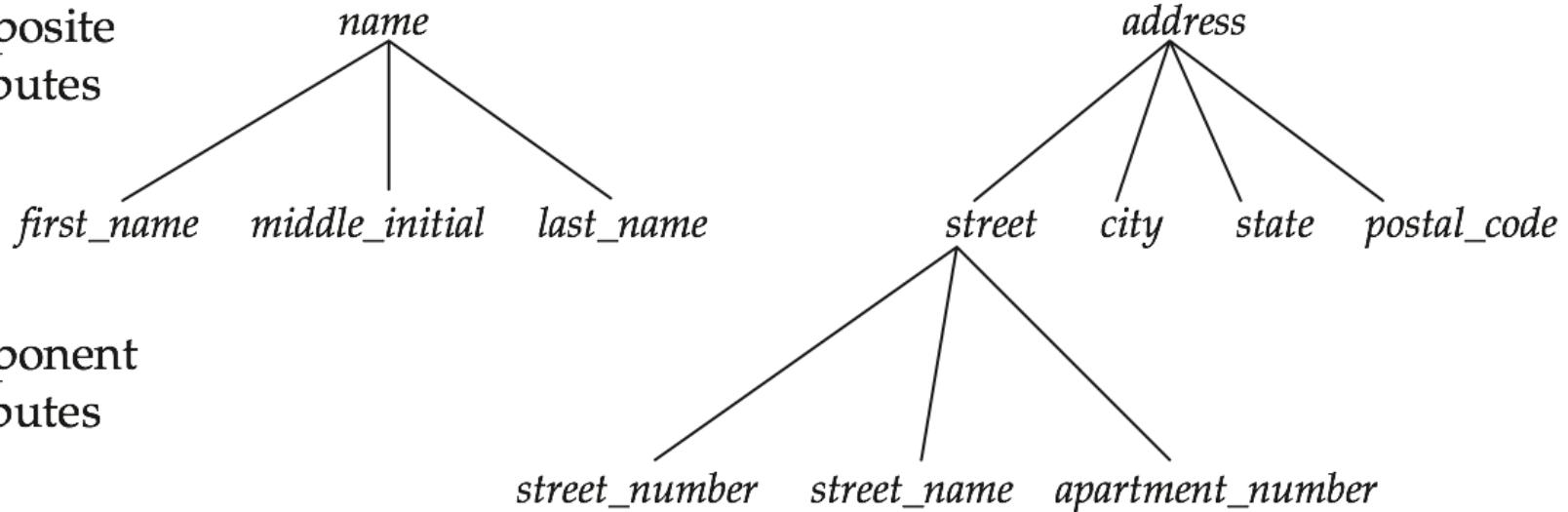
# Attributes

- n An entity is represented by a set of attributes, that is descriptive properties possessed by all members of an entity set.
  - | Example:  
 $\text{instructor} = (\text{ID}, \text{name}, \text{street}, \text{city}, \text{salary})$   
 $\text{course} = (\text{course\_id}, \text{title}, \text{credits})$
- n Domain – the set of permitted values for each attribute
- n Attribute types:
  - | Simple and composite attributes.
    - ▶ Example: Composite attribute: name with (Fname and Lname)
  - | Single-valued and multivalued attributes
    - ▶ Example: multivalued attribute: *phone\_numbers*
  - | Derived attributes
    - ▶ Can be computed from other attributes
    - ▶ Example: *age*, given *date\_of\_birth*



# Composite Attributes

composite  
attributes



component  
attributes

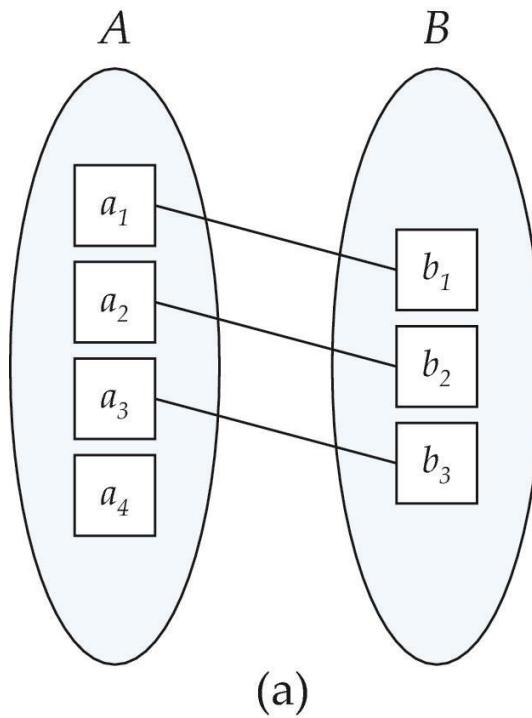


# Mapping Cardinality Constraints

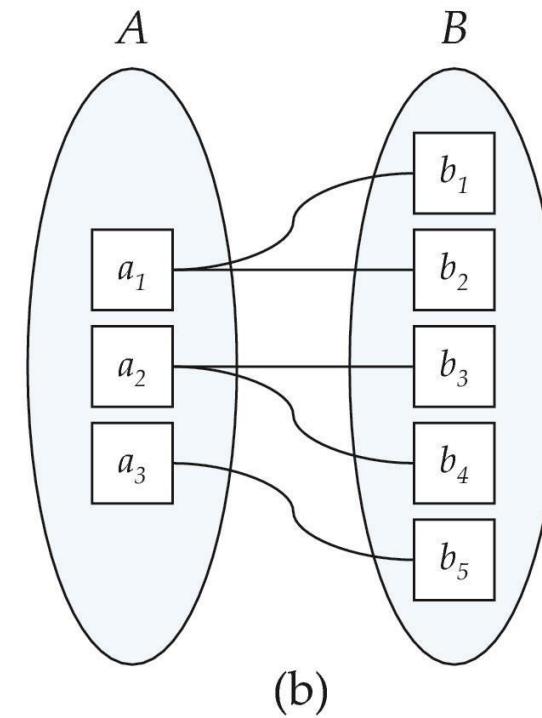
- n Express the number of entities to which another entity can be associated via **a relationship set**.
- n Most useful in describing binary relationship sets.
- n For a binary relationship set the **mapping cardinality** must be one of the following types:
  - | One to one
  - | One to many
  - | Many to one
  - | Many to many



# Mapping Cardinalities



One to one

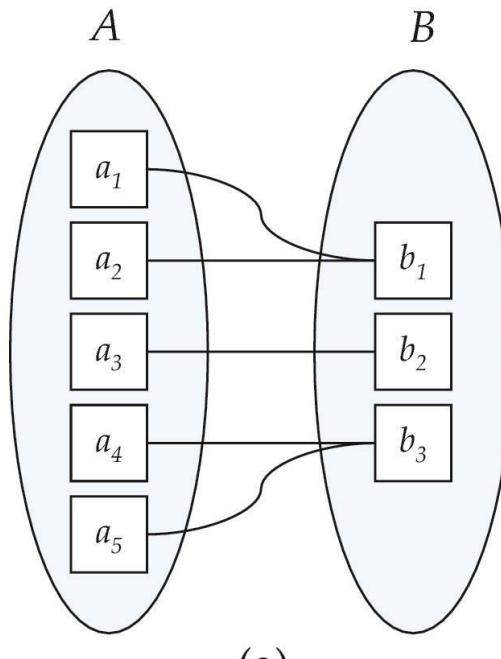


One to many

Note: Some elements in  $A$  and  $B$  may not be mapped to any elements in the other set

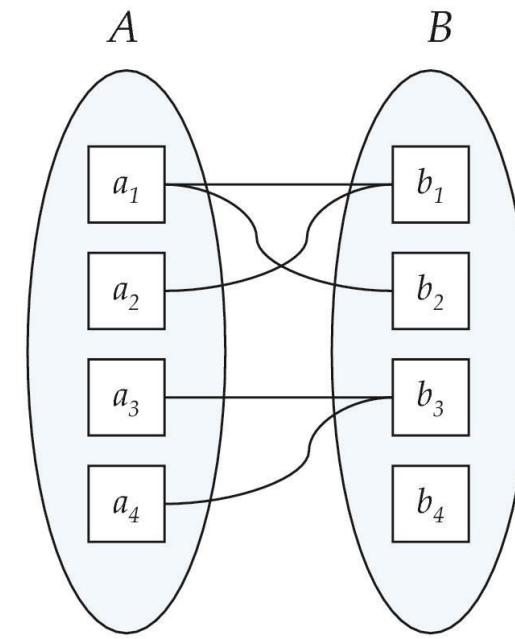


# Mapping Cardinalities



(a)

Many to  
one



(b)

Many to many

Note: Some elements in A and B may not be mapped to any elements in the other set



# Keys for Relationship Sets

- n The combination of primary keys of the participating entity sets forms a super key of a relationship set.
  - |  $(s\_id, i\_id)$  is the super key of *advisor*
  - | *NOTE: this means* a pair of entity sets can have at most one relationship in a particular relationship set.
- n Must consider the mapping cardinality of the relationship set when deciding what are the candidate keys
- n Suppose that the relationship set is many-to-many. Then the primary key of *advisor* consists of the union of the primary keys of *instructor* and *student*
- n If the relationship is many-to-one from *student* to *instructor*—that is, each student can have at most one advisor—then the primary key of *advisor* is simply the primary key of *student*
- n if an instructor can advise only one student—that is, if the *advisor* relationship is many-to-one from *instructor* to *student*—then the primary key of *advisor* is simply the primary key of *instructor*.
- n For one-to-one relationships either candidate key can be used as the primary key.

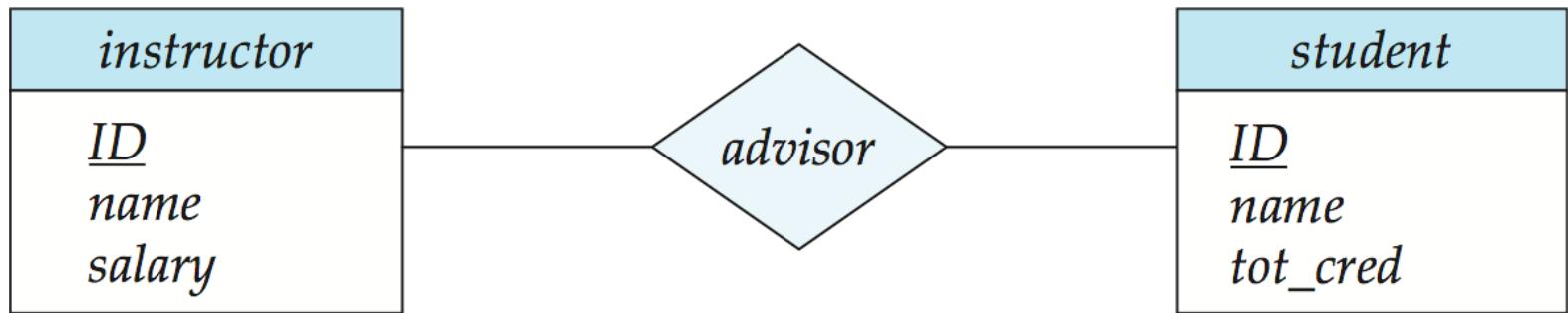


# Redundant Attributes

- n Suppose we have entity sets
  - | *instructor*, with attributes including *dept\_name*
  - | *Department*
- and a relationship
  - | *inst\_dept* relating *instructor* and *department*
- n Attribute *dept\_name* in entity *instructor* is redundant since there is an explicit relationship *inst\_dept* which relates instructors to departments
  - | The attribute replicates information present in the relationship, and should be removed from *instructor*
  - | BUT: when converting back to tables, in some cases the attribute gets reintroduced, as we will see.



# E-R Diagrams



- n **Rectangles** represent entity sets.
- n **Diamonds** represent relationship sets.
- n Attributes listed inside entity rectangle
- n **Underline** indicates **primary key** attributes

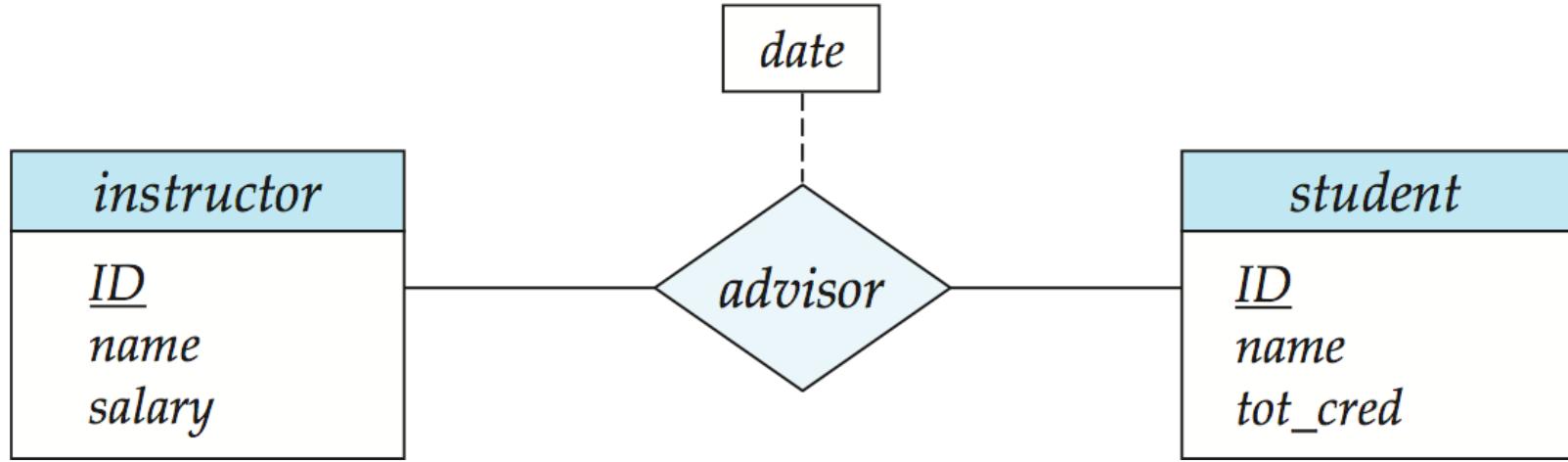


# Entity With Composite, Multivalued, and Derived Attributes

| <i>instructor</i>       |                       |
|-------------------------|-----------------------|
| <u><i>ID</i></u>        |                       |
| <i>name</i>             |                       |
|                         | <i>first_name</i>     |
|                         | <i>middle_initial</i> |
|                         | <i>last_name</i>      |
| <i>address</i>          |                       |
|                         | <i>street</i>         |
|                         | <i>street_number</i>  |
|                         | <i>street_name</i>    |
|                         | <i>apt_number</i>     |
| <i>city</i>             |                       |
| <i>state</i>            |                       |
| <i>zip</i>              |                       |
| { <i>phone_number</i> } |                       |
| <i>date_of_birth</i>    |                       |
| <i>age</i> ( )          |                       |



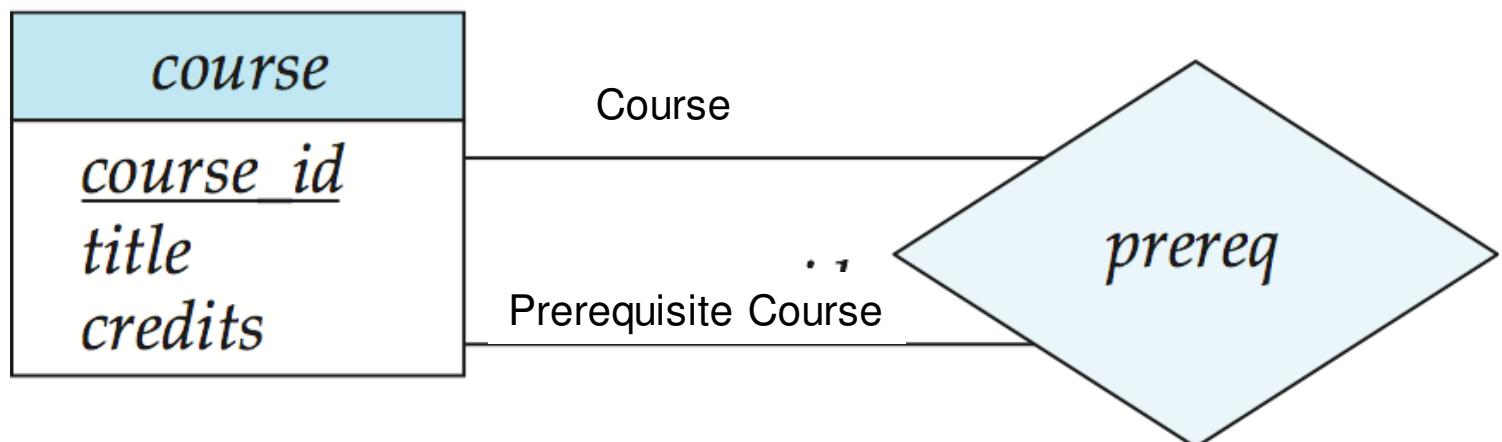
# Relationship Sets with Attributes





# Roles

- n Entity sets of a relationship need not be distinct
  - | Each occurrence of an entity set plays a “role” in the relationship
- n The labels “*course\_id*” and “*prereq\_id*” are called **roles**.





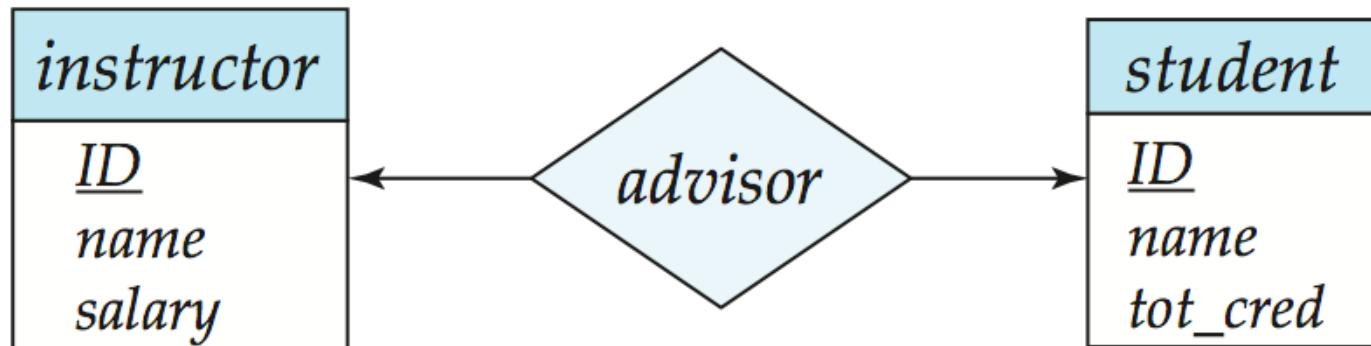
# Cardinality Constraints

- n We express cardinality constraints by drawing either a directed line ( $\rightarrow$ ), signifying “one,” or an undirected line ( $-$ ), signifying “many,” between the relationship set and the entity set.
- n One-to-one relationship:
  - | A student is associated with at most one *instructor* via the relationship *advisor*
  - | A *student* is associated with at most one *department* via *stud\_dept*



# One-to-One Relationship

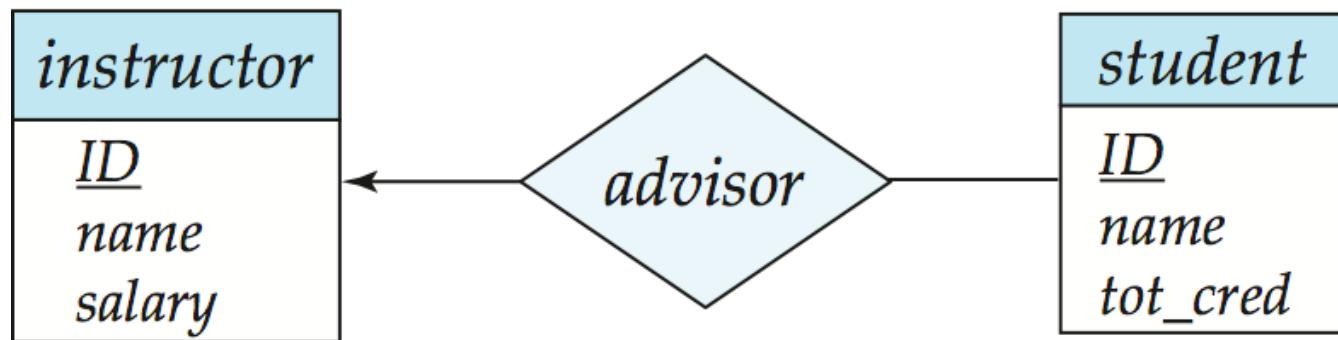
- n one-to-one relationship between an *instructor* and a *student*
  - | an instructor is associated with at most one student via *advisor*
  - | and a student is associated with at most one instructor via *advisor*





# One-to-Many Relationship

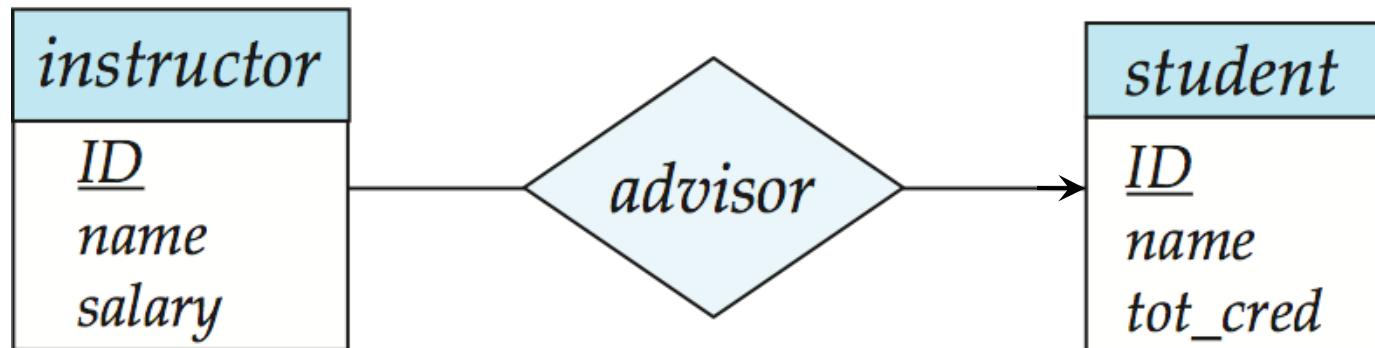
- n one-to-many relationship between an *instructor* and a *student*
  - | an instructor is associated with several (including 0) students via *advisor*
  - | a student is associated with at most one instructor via advisor,





# Many-to-One Relationships

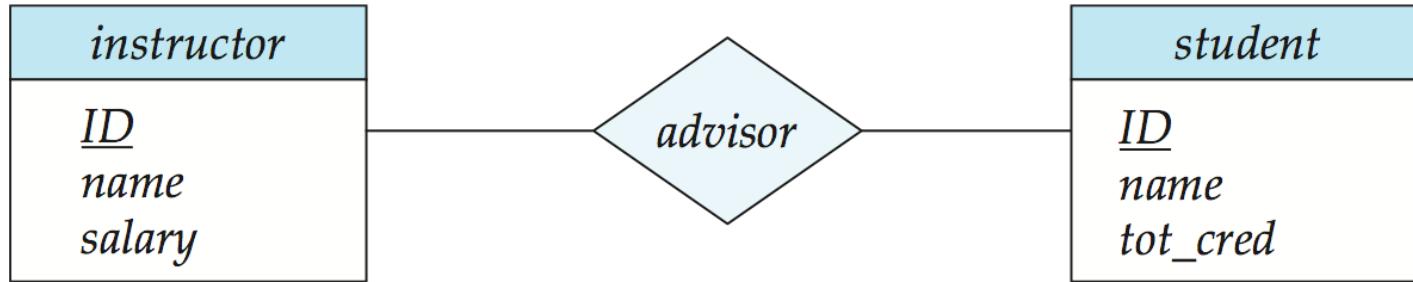
- n In a many-to-one relationship between an *instructor* and a *student*,
  - | an instructor is associated with **at most one student** via *advisor*,
  - | and a student is associated **with several (including 0) instructors** via *advisor*





# Many-to-Many Relationship

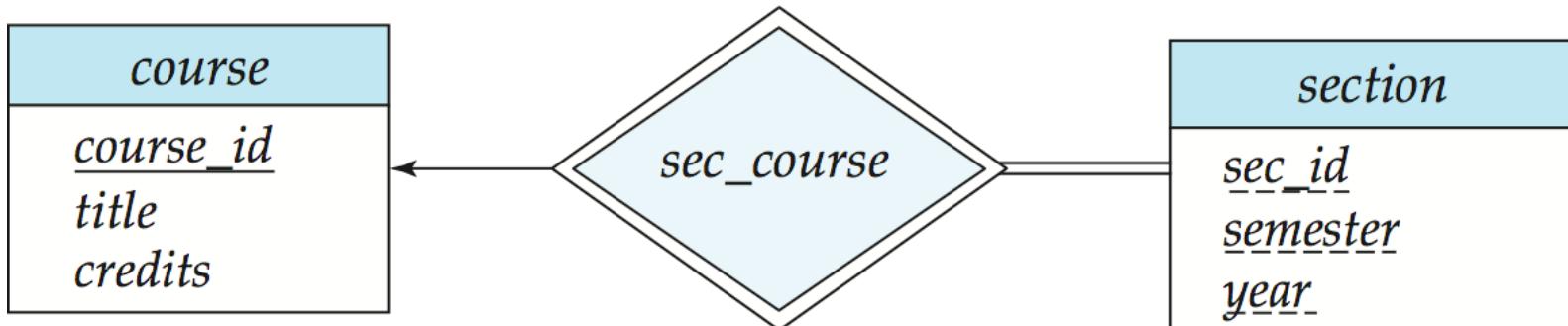
- n An instructor is associated with several (possibly 0) students via *advisor*
- n A student is associated with several (possibly 0) instructors via *advisor*





# Participation of an Entity Set in a Relationship Set

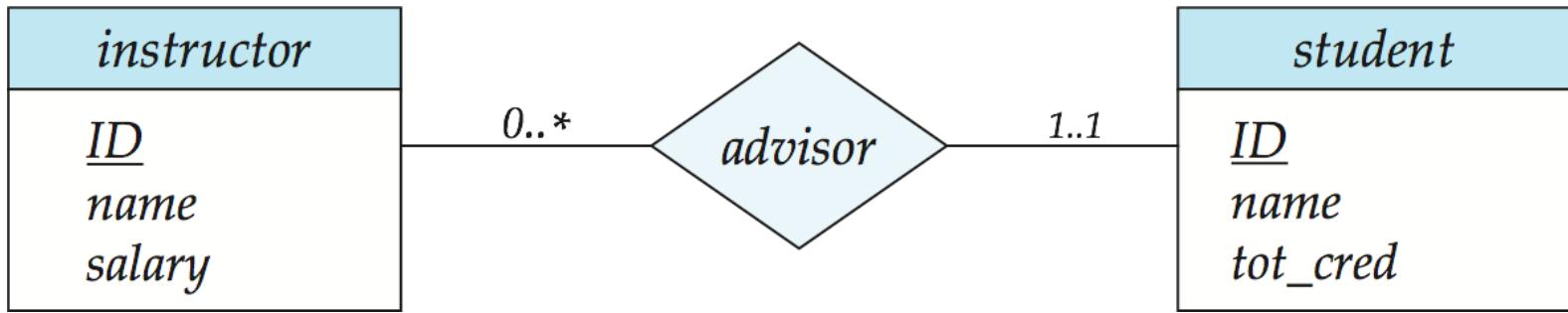
- n **Total participation** (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set
  - | E.g., participation of *section* in *sec\_course* is total
    - ▶ *every section must have an associated course*
- n **Partial participation**: some entities may not participate in any relationship in the relationship set
  - | Example: participation of *instructor* in *advisor* is partial





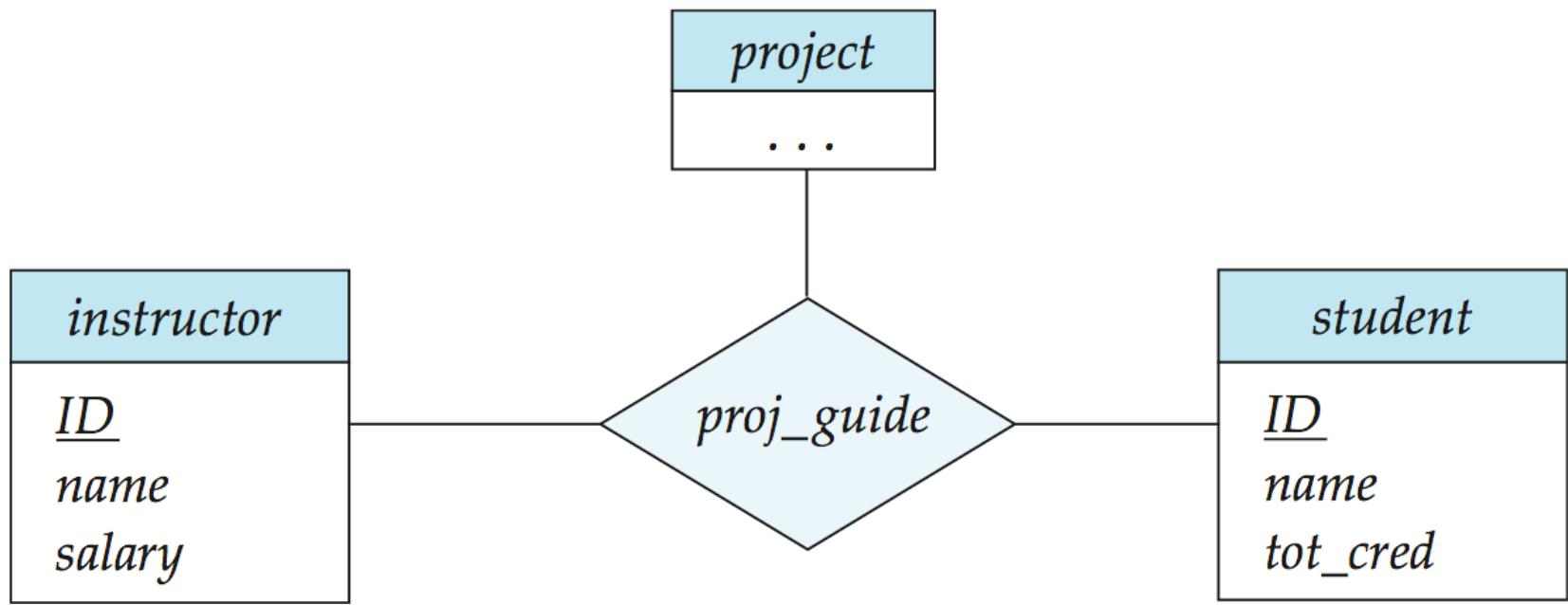
# Alternative Notation for Cardinality Limits

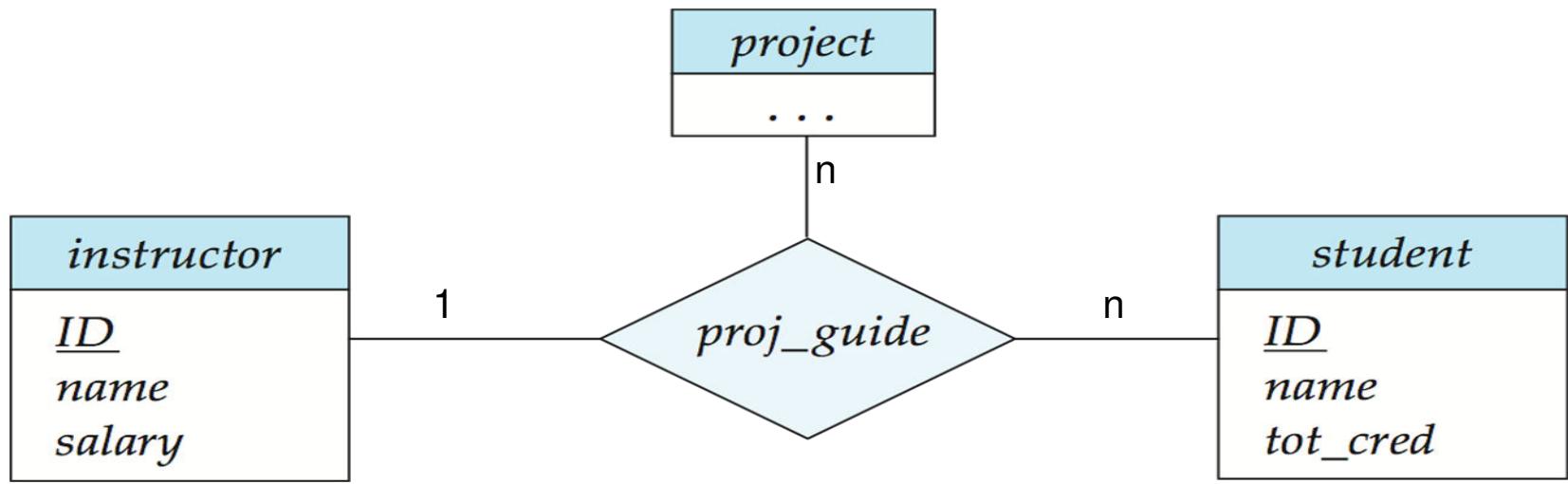
- Cardinality limits can also express participation constraints





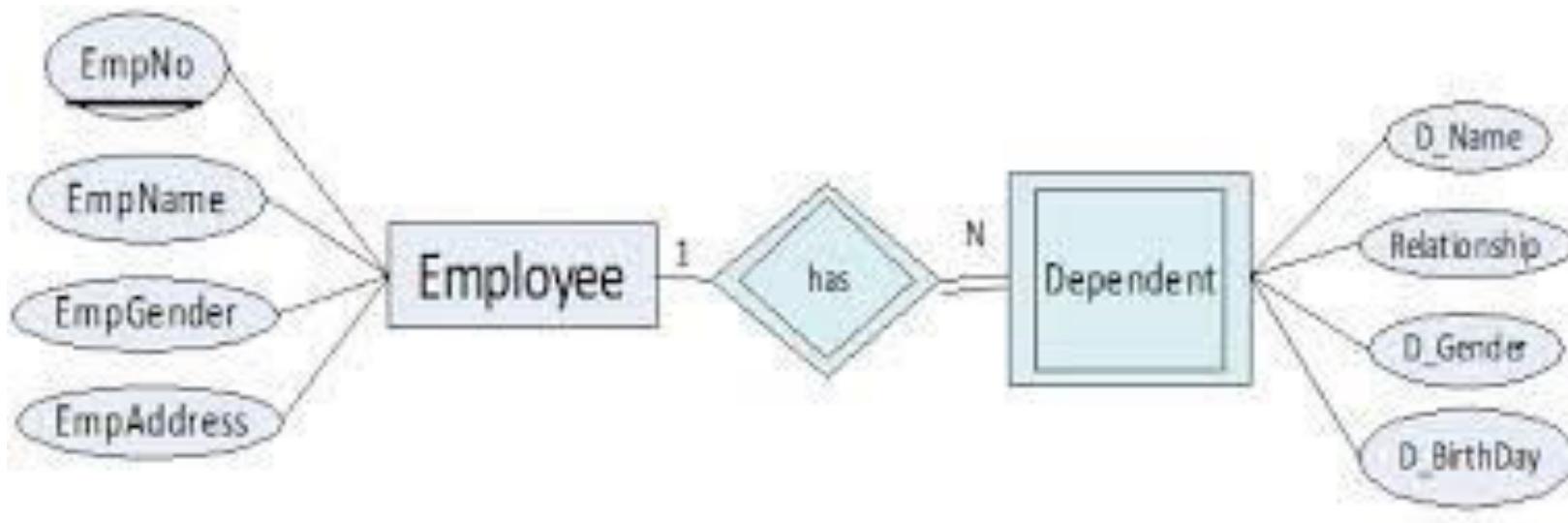
# E-R Diagram with a Ternary Relationship







# Example for weak entity set





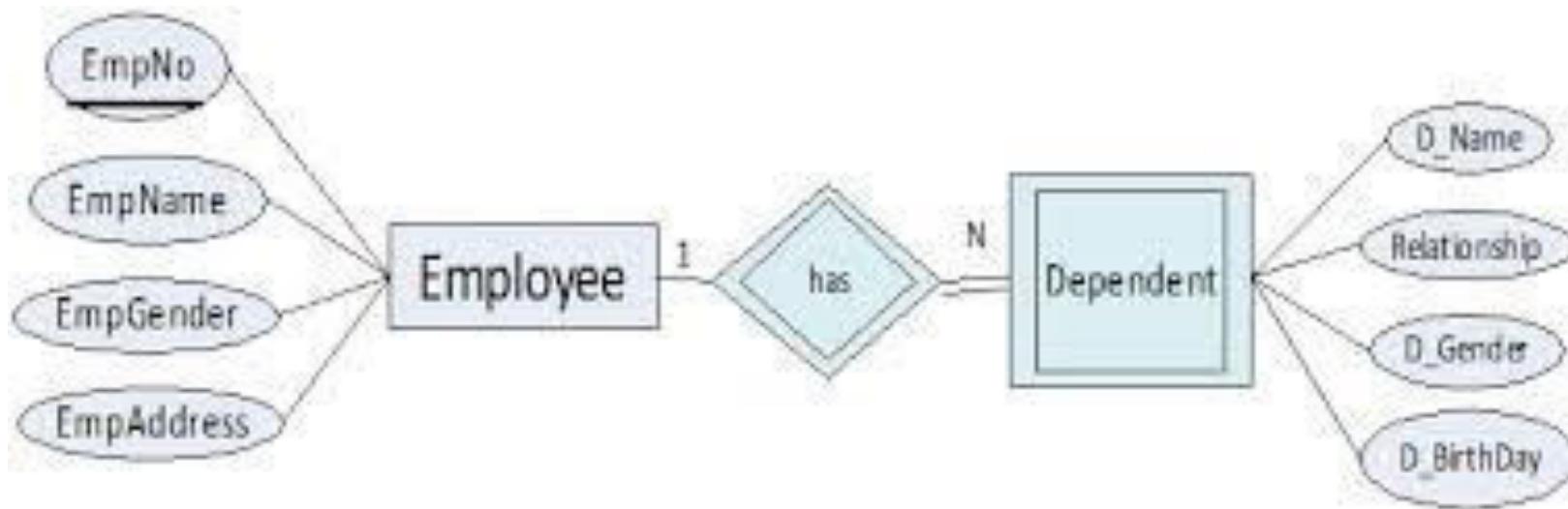
# Weak Entity Sets

- n An entity set that does not have a primary key is referred to as a **weak entity set**.
- n The existence of a weak entity set depends on the existence of a **identifying entity set**
  - | It must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set
  - | **Identifying relationship** depicted using a double diamond
- n The **discriminator (or partial key) of a weak entity set** is the set of attributes that distinguishes among all the entities of a weak entity set.
- n The **primary key of a weak entity** set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator.



# Weak Entity Sets (Cont.)

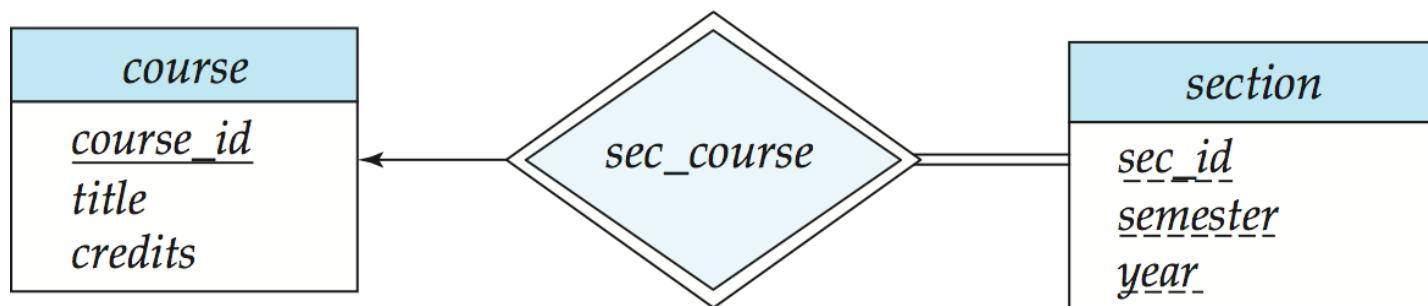
- We underline the **discriminator** of a weak entity set with a dashed line.
- We put the identifying relationship of a weak entity in a **double diamond**.





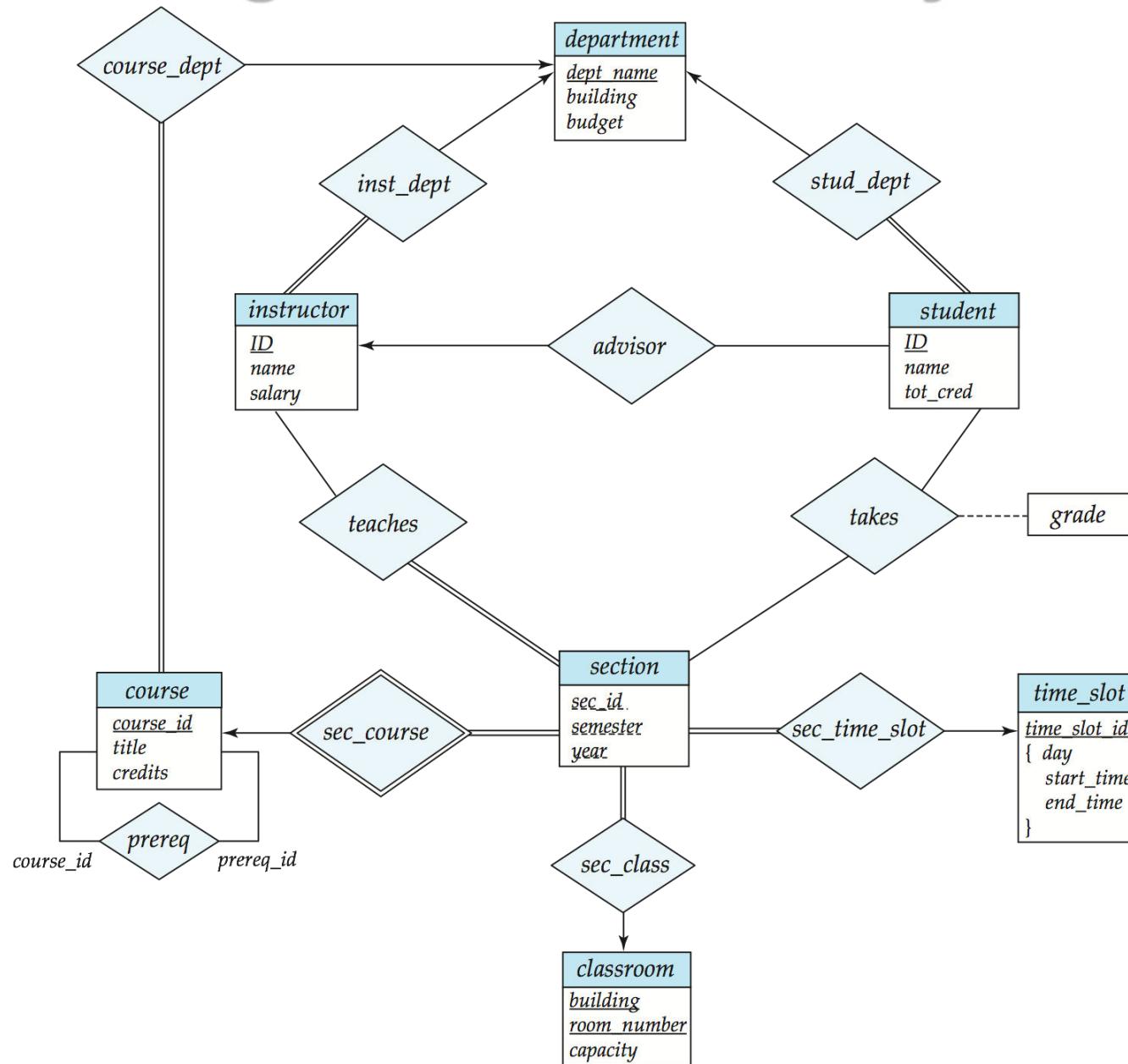
# Weak Entity Sets (Cont.)

- n Note: the primary key of the strong entity set is not explicitly stored with the weak entity set, since it is implicit in the identifying relationship.
- n If *course\_id* were explicitly stored, *section* could be made a strong entity, but then the relationship between *section* and *course* would be duplicated by an implicit relationship defined by the attribute *course\_id* common to *course* and *section*
- n Primary key for *section* – (*course\_id*, *sec\_id*, *semester*, *year*)





# E-R Diagram for a University Enterprise





# Reduction to Relational Schemas



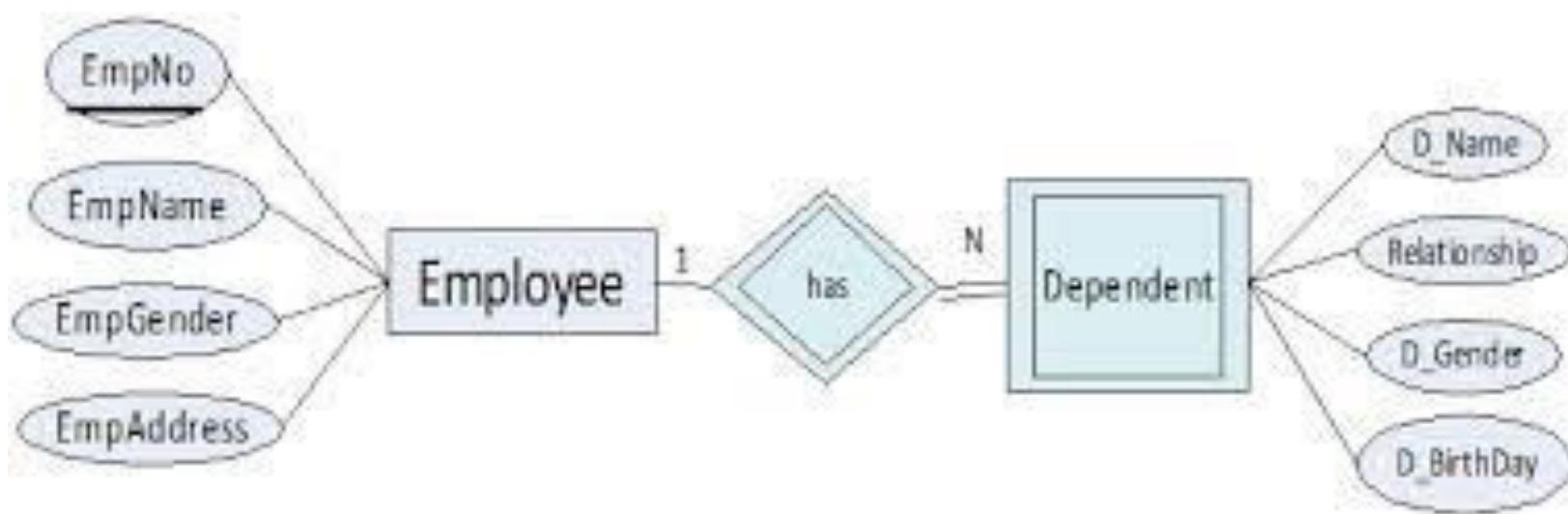
# Reduction to Relation Schemas

- n Entity sets and relationship sets can be expressed uniformly as *relation schemas* that represent the contents of the database.
- n A database which conforms to an E-R diagram can be represented by a collection of schemas.
- n For each entity set and relationship set there is a unique schema that is assigned the name of the corresponding entity set or relationship set.
- n Each schema has a number of columns (generally corresponding to attributes), which have unique names.



# Representing Entity Sets with Simple Attributes

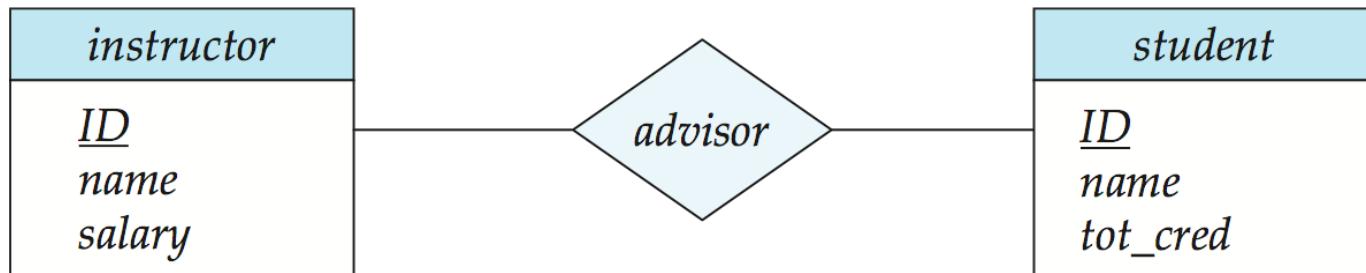
- n A strong entity set reduces to a schema with the same attributes *student(ID, name, tot\_cred)*
- n A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set  
Dependent ( *emp\_no,D\_name,relationship,D\_Gender,D\_Birthday*)





# Representing Relationship Sets

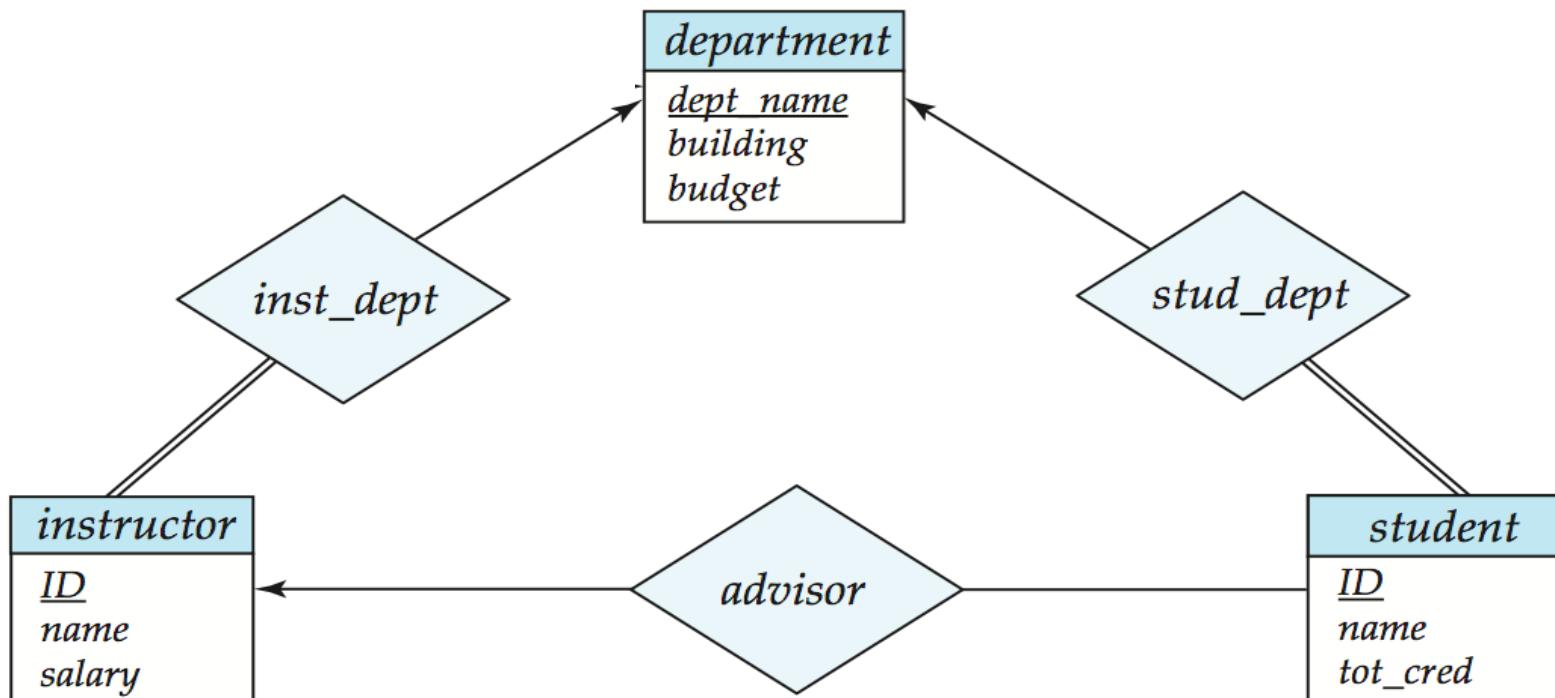
- n A many-to-many relationship set is represented as a schema with attributes for the primary keys of the two participating entity sets, and any descriptive attributes of the relationship set.
- n Example: schema for relationship set *advisor*  
 $\text{advisor} = (\underline{s\_id}, \underline{i\_id})$





# Redundancy of Schemas

- n Many-to-one and one-to-many relationship sets that are **total** on the many-side can be represented by **adding an extra attribute to the “many” side, containing the primary key of the “one” side**
- n If **Partial participation**: Separate table
- n Example: Instead of creating a schema for relationship set *inst\_dept*, add an attribute *dept\_name* to the schema arising from entity set *instructor*





# Redundancy of Schemas (Cont.)

- For **one-to-one** relationship sets, either side can be chosen to act as the “many” side
  - That is, **extra attribute** can be added to either of the tables corresponding to the two entity sets
- If participation is **partial** on the “many” side, replacing a schema by an extra attribute in the schema corresponding to the “many” side could result in null values
- The **schema corresponding** to a **relationship set** linking a **weak entity set** to its identifying **strong entity set** is **redundant**.
  - Example: The *section* schema already contains the attributes that would appear in the *sec\_course* schema



# Composite and Multivalued Attributes

| <i>instructor</i>       |
|-------------------------|
| <u>ID</u>               |
| <i>name</i>             |
| <i>first_name</i>       |
| <i>middle_initial</i>   |
| <i>last_name</i>        |
| <i>address</i>          |
| <i>street</i>           |
| <i>street_number</i>    |
| <i>street_name</i>      |
| <i>apt_number</i>       |
| <i>city</i>             |
| <i>state</i>            |
| <i>zip</i>              |
| { <i>phone_number</i> } |
| <i>date_of_birth</i>    |
| <i>age ()</i>           |

- Composite attributes are flattened out by creating a separate attribute for each component attribute
  - Example: given entity set *instructor* with composite attribute *name* with component attributes *first\_name* and *last\_name* the schema corresponding to the entity set has two attributes *name\_first\_name* and *name\_last\_name*
    - ▶ Prefix omitted if there is no ambiguity
- Ignoring multivalued attributes, extended instructor schema is
  - *instructor(ID, first\_name, middle\_initial, last\_name, street\_number, street\_name, apt\_number, city, state, zip\_code, date\_of\_birth)*



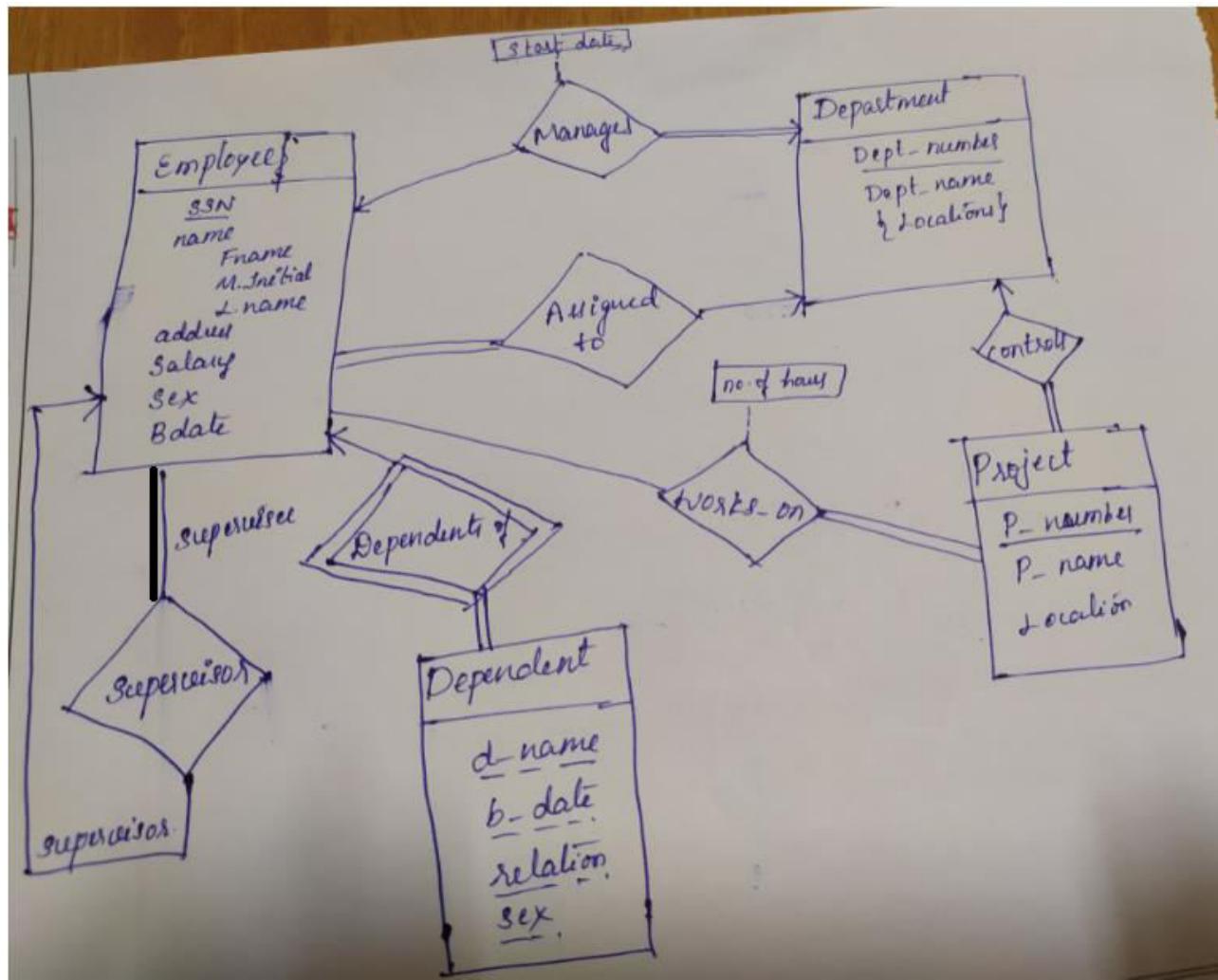
# Composite and Multivalued Attributes

- A multivalued attribute  $M$  of an entity  $E$  is represented by a separate schema  $EM$ 
  - Schema  $EM$  has attributes corresponding to the primary key of  $E$  and an attribute corresponding to multivalued attribute  $M$
  - Example: Multivalued attribute  $phone\_number$  of  $instructor$  is represented by a schema:  
 $inst\_phone = ( \underline{ID}, \underline{phone\_number} )$
  - Each value of the multivalued attribute maps to a separate tuple of the relation on schema  $EM$ 
    - ▶ For example, an  $instructor$  entity with primary key 22222 and phone numbers 456-7890 and 123-4567 maps to two tuples:  
 $(22222, 456-7890)$  and  $(22222, 123-4567)$



Construct an ER Diagram for Company having following details :

- Company organized into DEPARTMENT. Each department has unique name and a particular employee who manages the department. Start date for the manager is recorded. Department may have several locations.
- A department controls a number of PROJECT. Projects have a unique name, number and a single location.
- Company's EMPLOYEE name, ssno, address, salary, sex and birth date are recorded. An employee is assigned to one department, but may work for several projects (not necessarily controlled by her dept). Number of hours/week an employee works on each project is recorded; The immediate supervisor for the employee.
- Employee's DEPENDENT are tracked for health insurance purposes (dependent name, birthdate, relationship to employee).
- Further, categorize employees as Engineers and non-engineers.

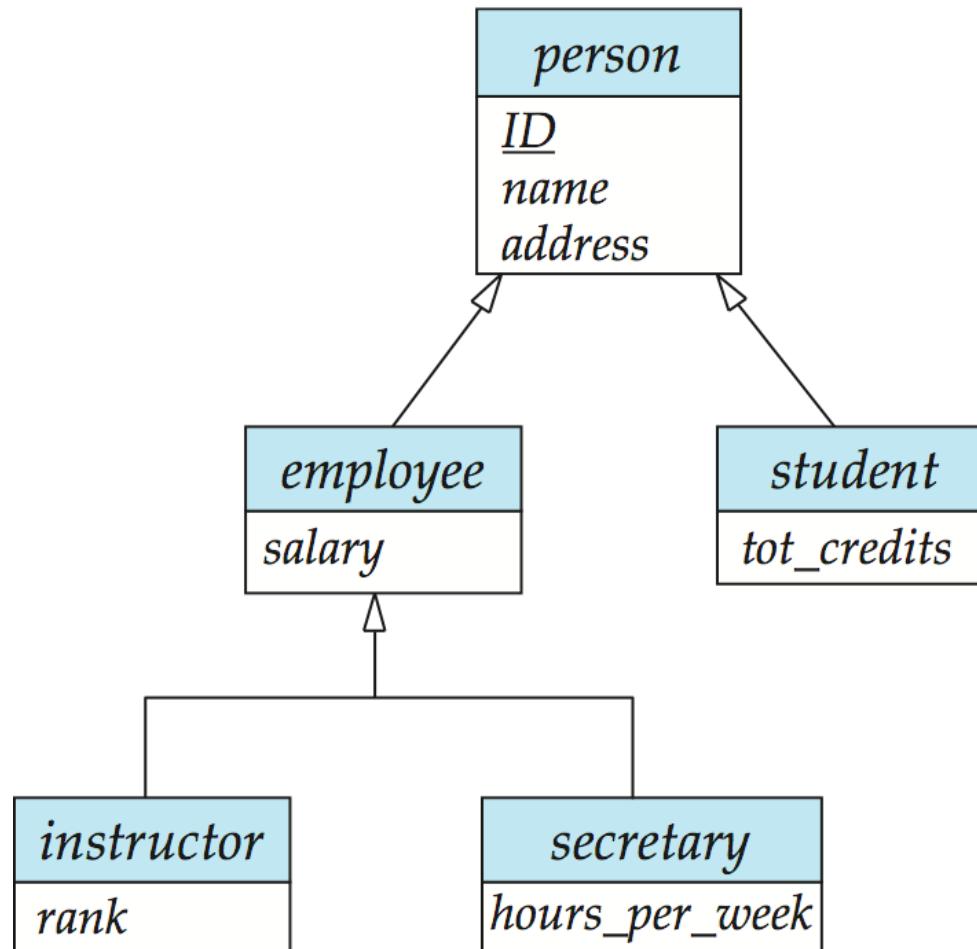




# Extended ER Features



# Specialization/Generalization Example



Advantage ?



# Extended E-R Features: Specialization

- Top-down design process; we designate subgroupings within an entity set that are distinctive from other entities in the set.
- These subgroupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.
- Depicted by a *triangle* component labeled ISA (E.g., *instructor* “is a” *person*).
- **Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.



# Extended ER Features: Generalization

- **A bottom-up design process** – combine a number of entity sets that share the same features into a higher-level entity set.
- Specialization and generalization are simple **inversions** of each other; they are represented in **an E-R diagram in the same way**.
- The terms specialization and generalization are used interchangeably.



# Specialization and Generalization (Cont.)

- Can have multiple specializations of an entity set based on different features.
- E.g., *permanent\_employee* vs. *temporary\_employee*, in addition to *instructor* vs. *secretary*
- Each particular employee would be
  - a member of one of *permanent\_employee* or *temporary\_employee*,
  - and also a member of one of *instructor*, *secretary*
- The **ISA relationship** also referred to as **superclass - subclass** relationship



# Design Constraints on a Specialization/Generalization

- Constraint on whether or not entities may belong to more than one lower-level entity set within a single generalization.
  - **Disjoint**
    - ▶ an entity can belong to only one lower-level entity set
    - ▶ Noted in E-R diagram by having multiple lower-level entity sets link to the same triangle
  - **Overlapping**
    - ▶ an entity can belong to more than one lower-level entity set



# Design Constraints on a Specialization/Generalization (Cont.)

- **Completeness constraint** -- specifies whether or not an entity in the higher-level entity set **must belong to at least one of the lower-level entity sets** within a generalization.
  - **total**: an entity **must belong to one of the lower-level entity sets**
  - **partial**: an entity **need not belong to one of the lower-level entity sets**



# Representing Specialization via Schemas

- Method 1:
  - Form a schema for the higher-level entity
  - Form a schema for each lower-level entity set, include primary key of higher-level entity set and local attributes

| schema          | attributes                    |
|-----------------|-------------------------------|
| <i>person</i>   | <i>ID, name, street, city</i> |
| <i>student</i>  | <i>ID, tot_cred</i>           |
| <i>employee</i> | <i>ID, salary</i>             |

- Drawback: getting information about, an *employee* requires accessing two relations, the one corresponding to the low-level schema and the one corresponding to the high-level schema



# Representing Specialization as Schemas (Cont.)

- Method 2:
  - Form a schema for each entity set with all local and inherited attributes

| schema          | attributes                              |
|-----------------|-----------------------------------------|
| <i>person</i>   | <i>ID, name, street, city</i>           |
| <i>student</i>  | <i>ID, name, street, city, tot_cred</i> |
| <i>employee</i> | <i>ID, name, street, city, salary</i>   |

- If specialization is **total**, the schema for the generalized entity set (*person*) not required to store information
  - ▶ Can be defined as a “view” relation containing **union of specialization relations**
- **Drawback:** *name, street* and *city* may be stored redundantly for people who are both **students** and **employees**( for Overlapping constraint case)



Das Bild kann zurzeit nicht angezeigt werden.

# How about doing another ER design interactively on the board?

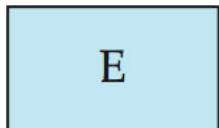
Database System Concepts, 6<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

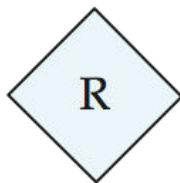
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Summary of Symbols Used in E-R Notation



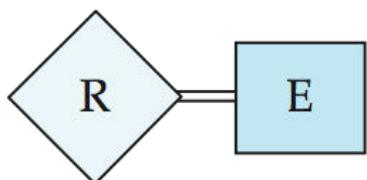
entity set



relationship set



identifying  
relationship set  
for weak entity set



total participation  
of entity set in  
relationship

| E    |
|------|
| A1   |
| A2   |
| A2.1 |
| A2.2 |
| {A3} |
| A40  |

attributes:  
simple (A1),  
composite (A2) and  
multivalued (A3)  
derived (A4)

| E         |
|-----------|
| <u>A1</u> |

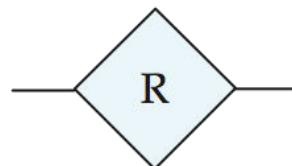
primary key

| E     |
|-------|
| A1    |
| ..... |

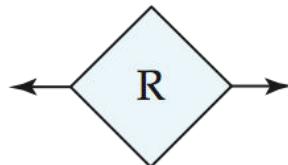
discriminating  
attribute of  
weak entity set



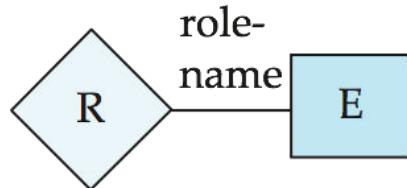
# Symbols Used in E-R Notation (Cont.)



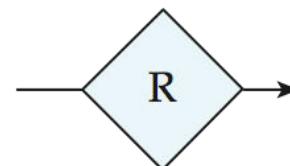
many-to-many  
relationship



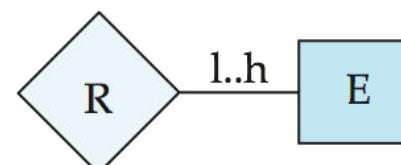
one-to-one  
relationship



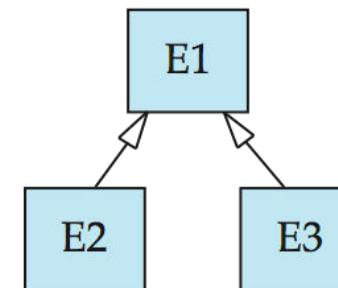
role indicator



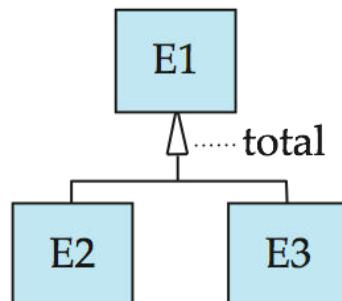
many-to-one  
relationship



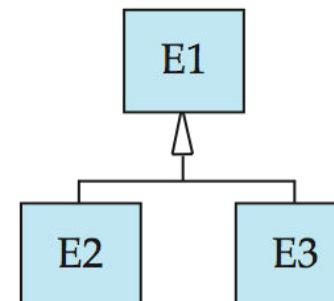
cardinality  
limits



ISA: generalization  
or specialization



total (disjoint)  
generalization



disjoint  
generalization



Das Bild kann zurzeit nicht angezeigt werden.

# End of Chapter 7

**Database System Concepts, 6<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan

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



# Figure 7.01

|       |            |
|-------|------------|
| 76766 | Crick      |
| 45565 | Katz       |
| 10101 | Srinivasan |
| 98345 | Kim        |
| 76543 | Singh      |
| 22222 | Einstein   |

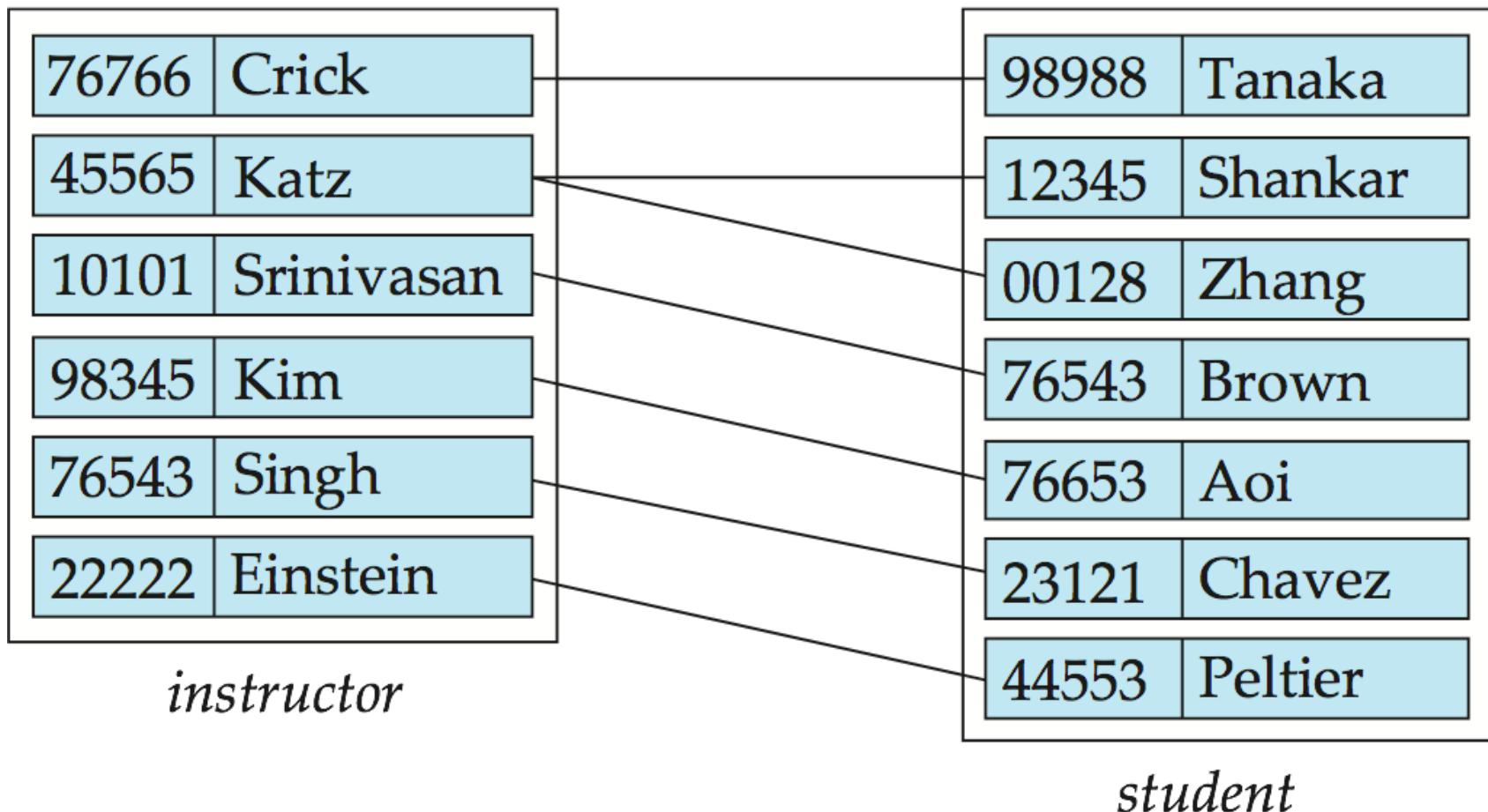
*instructor*

|       |         |
|-------|---------|
| 98988 | Tanaka  |
| 12345 | Shankar |
| 00128 | Zhang   |
| 76543 | Brown   |
| 76653 | Aoi     |
| 23121 | Chavez  |
| 44553 | Peltier |

*student*

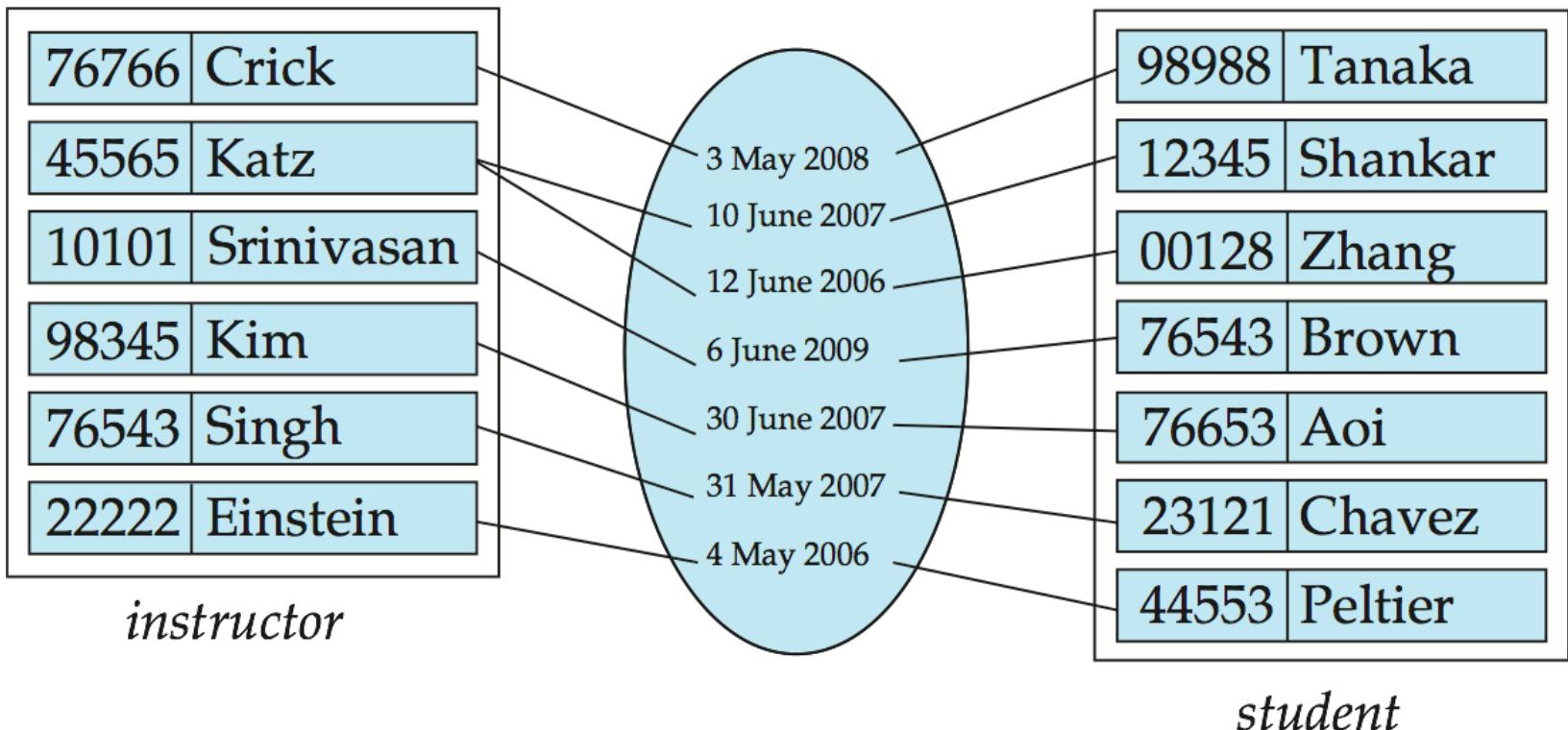


# Figure 7.02



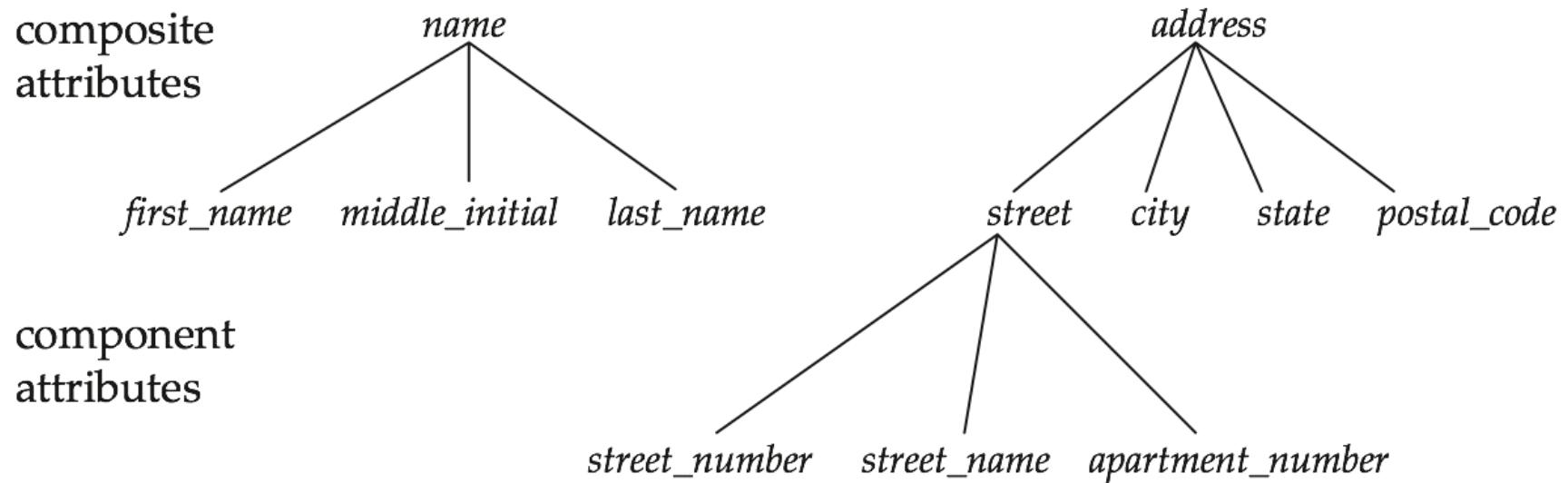


# Figure 7.03



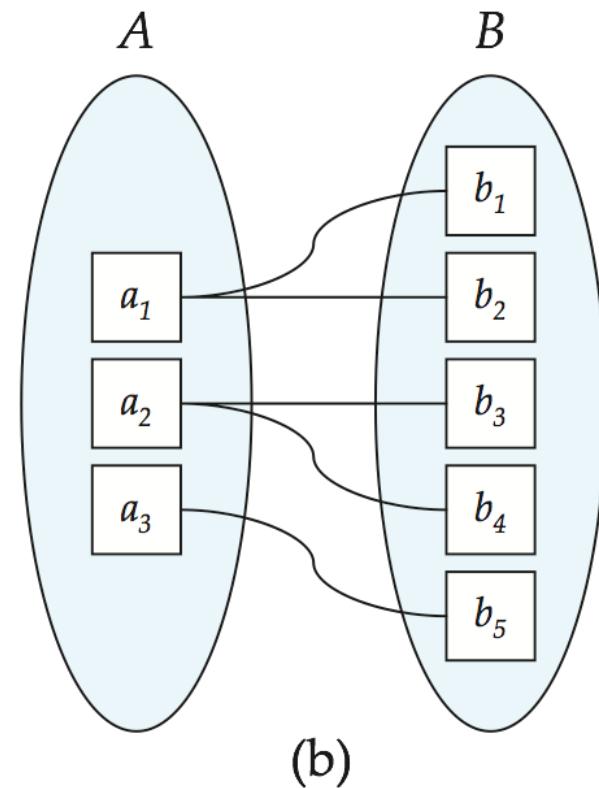
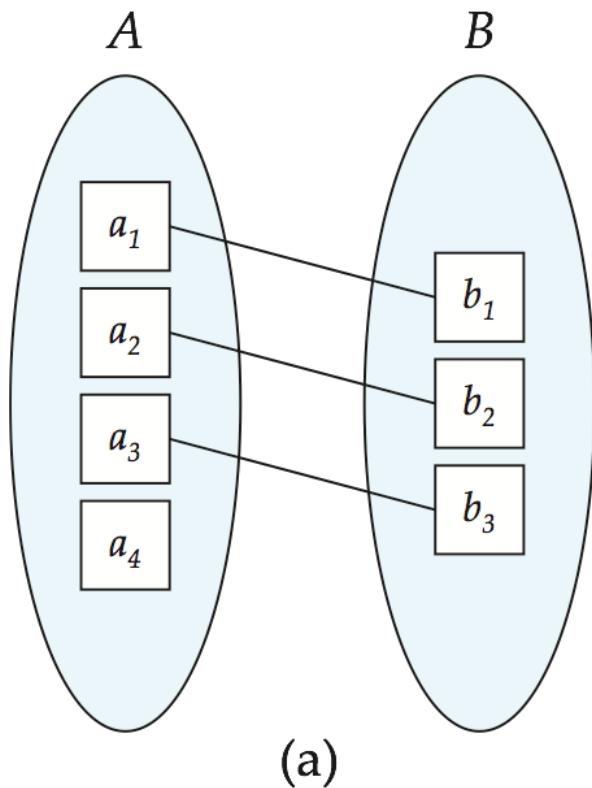


# Figure 7.04



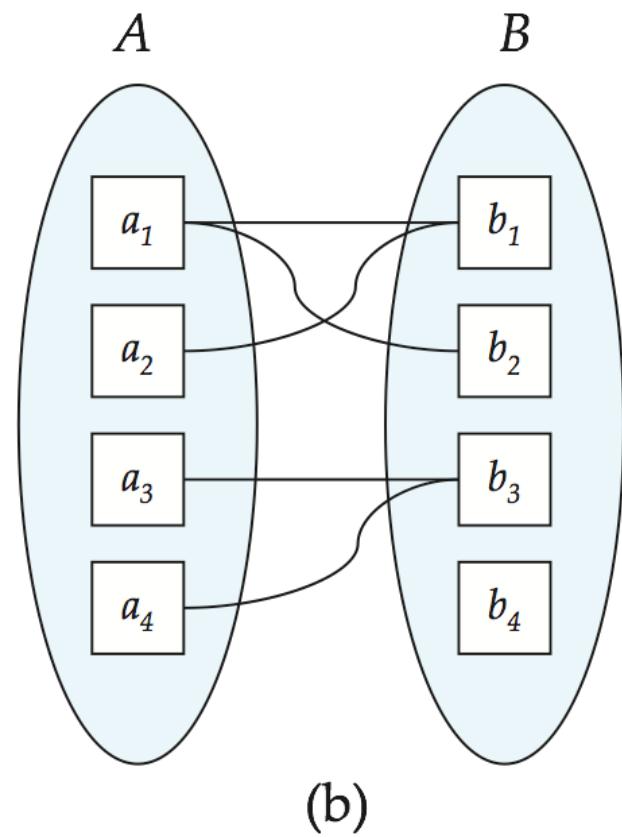
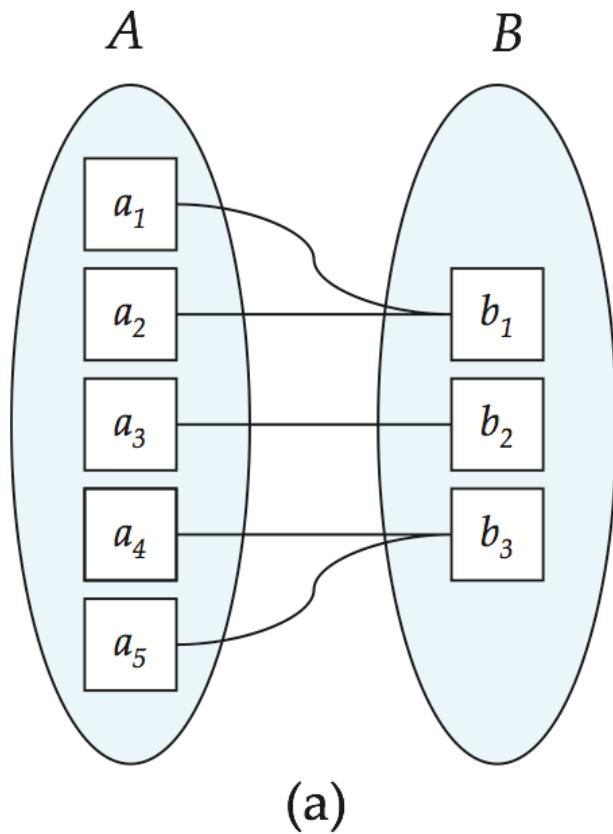


# Figure 7.05



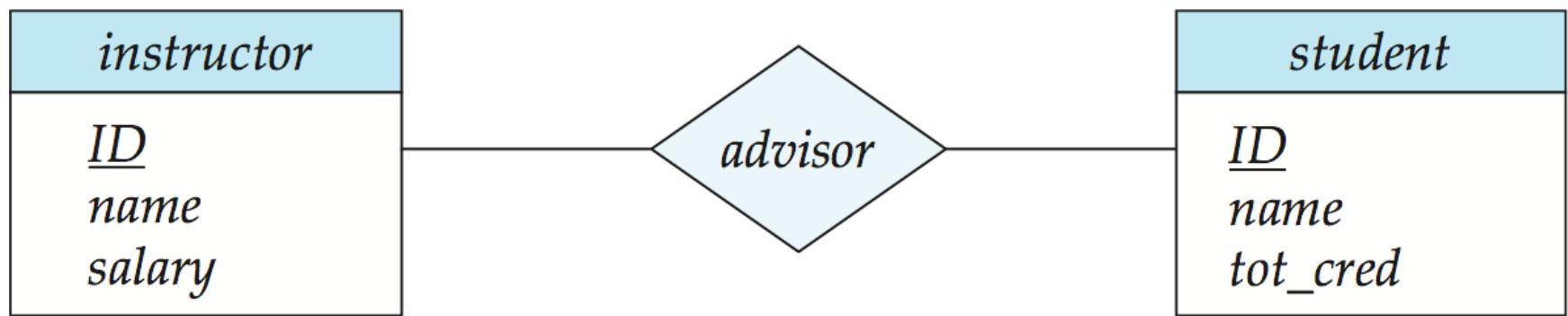


# Figure 7.06



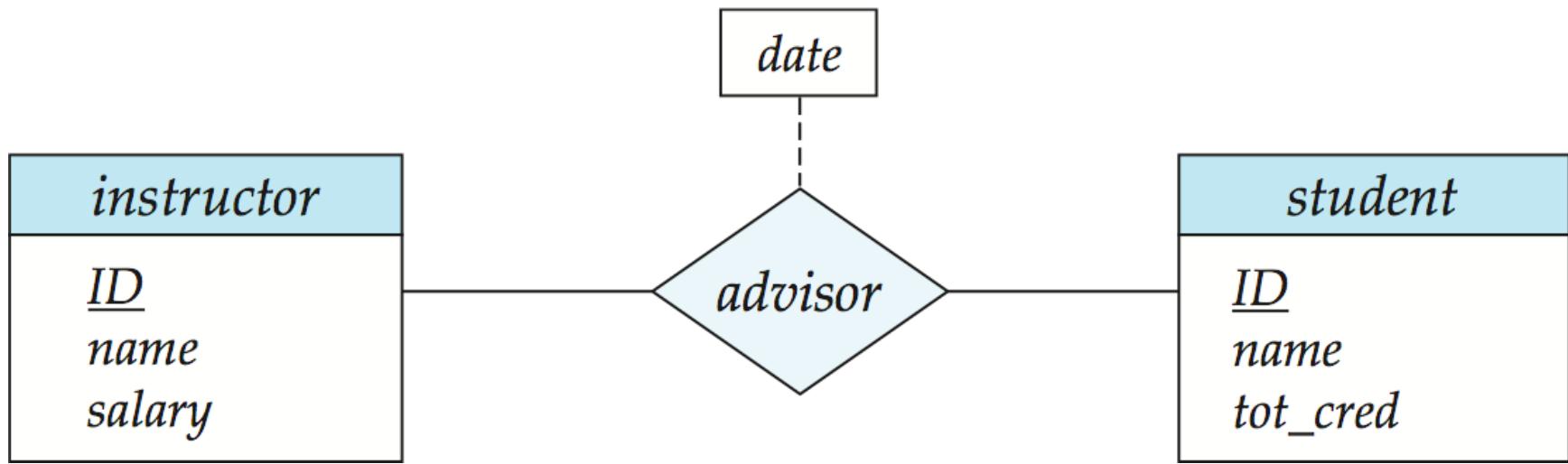


# Figure 7.07



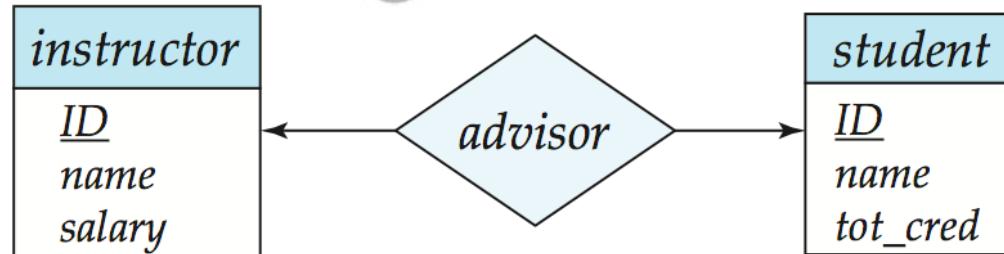


# Figure 7.08

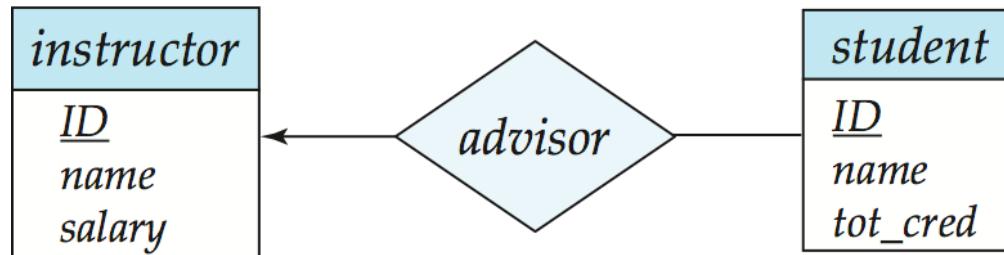




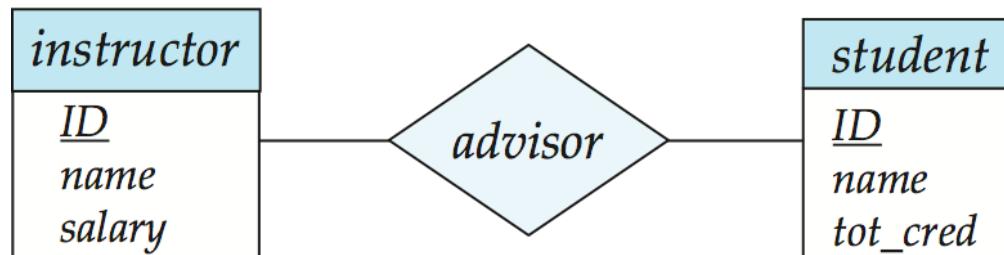
# Figure 7.09



(a)



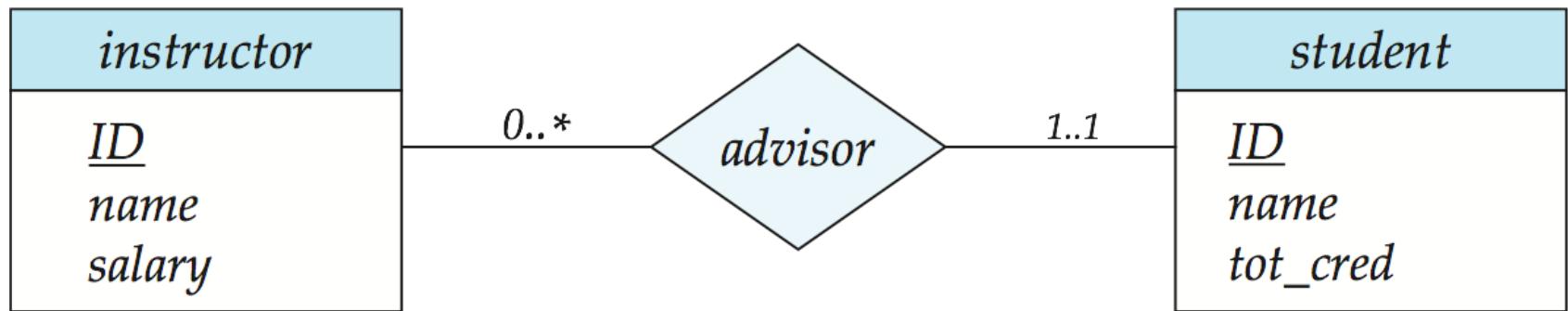
(b)



(c)



# Figure 7.10



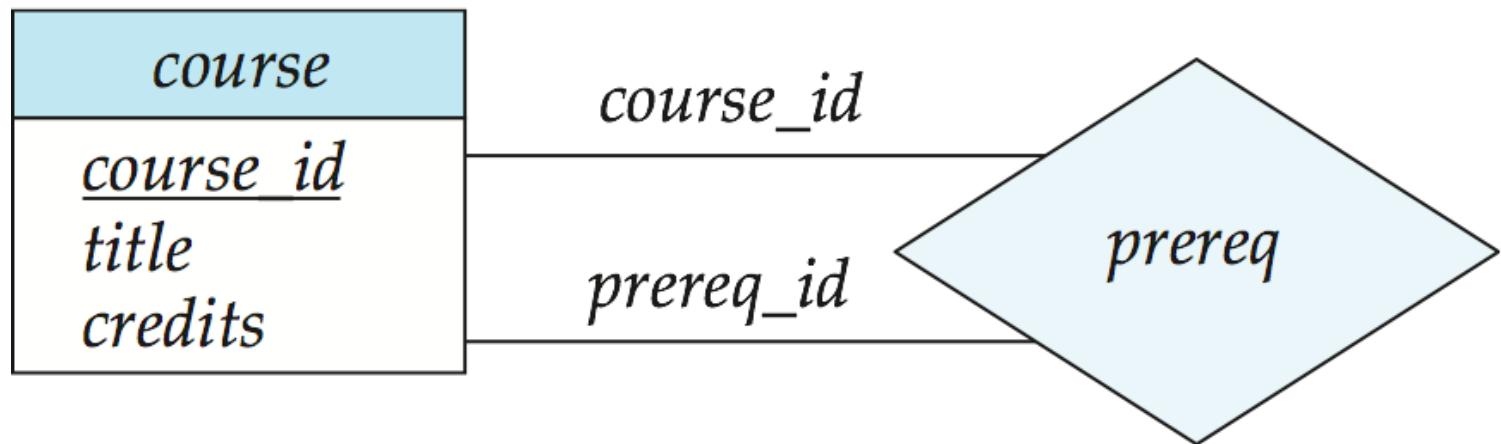


# Figure 7.11

| <i>instructor</i>       |
|-------------------------|
| <u>ID</u>               |
| <i>name</i>             |
| <i>first_name</i>       |
| <i>middle_initial</i>   |
| <i>last_name</i>        |
| <i>address</i>          |
| <i>street</i>           |
| <i>street_number</i>    |
| <i>street_name</i>      |
| <i>apt_number</i>       |
| <i>city</i>             |
| <i>state</i>            |
| <i>zip</i>              |
| { <i>phone_number</i> } |
| <i>date_of_birth</i>    |
| <i>age()</i>            |

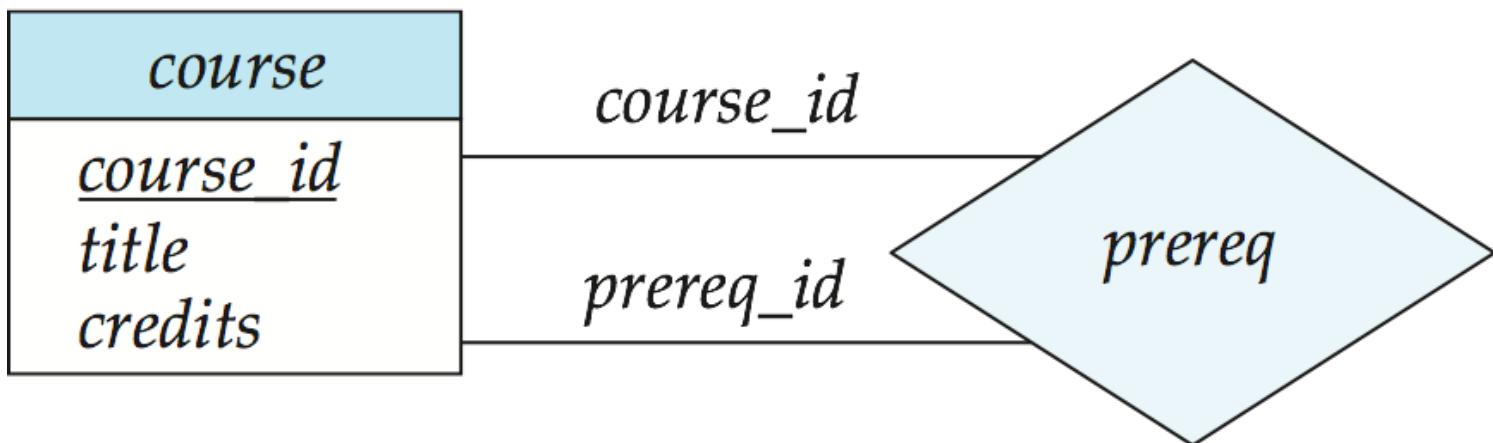


# Figure 7.12



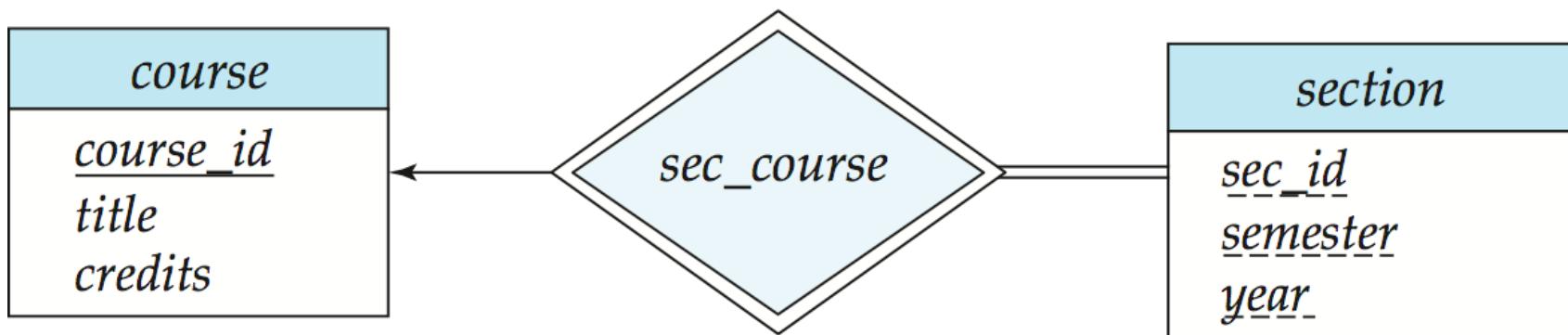


# Figure 7.13



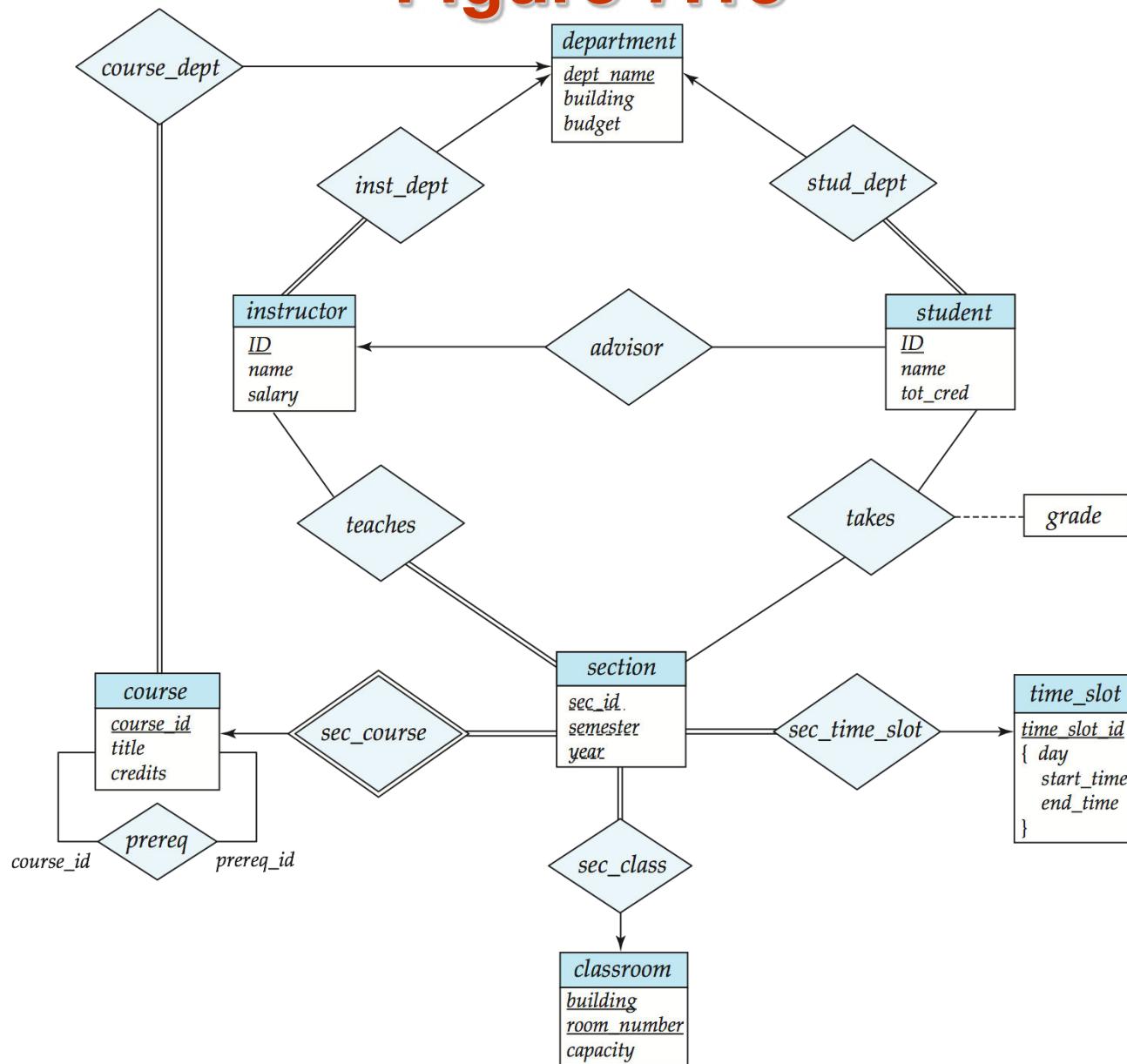


# Figure 7.14





# Figure 7.15

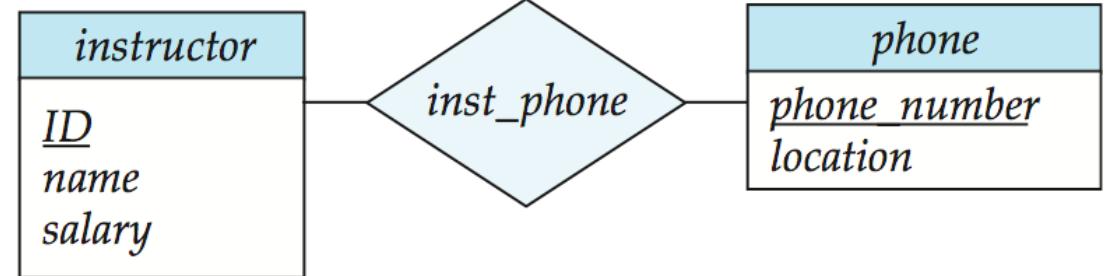




# Figure 7.17

| <i>instructor</i>   |
|---------------------|
| <u>ID</u>           |
| <i>name</i>         |
| <i>salary</i>       |
| <i>phone_number</i> |

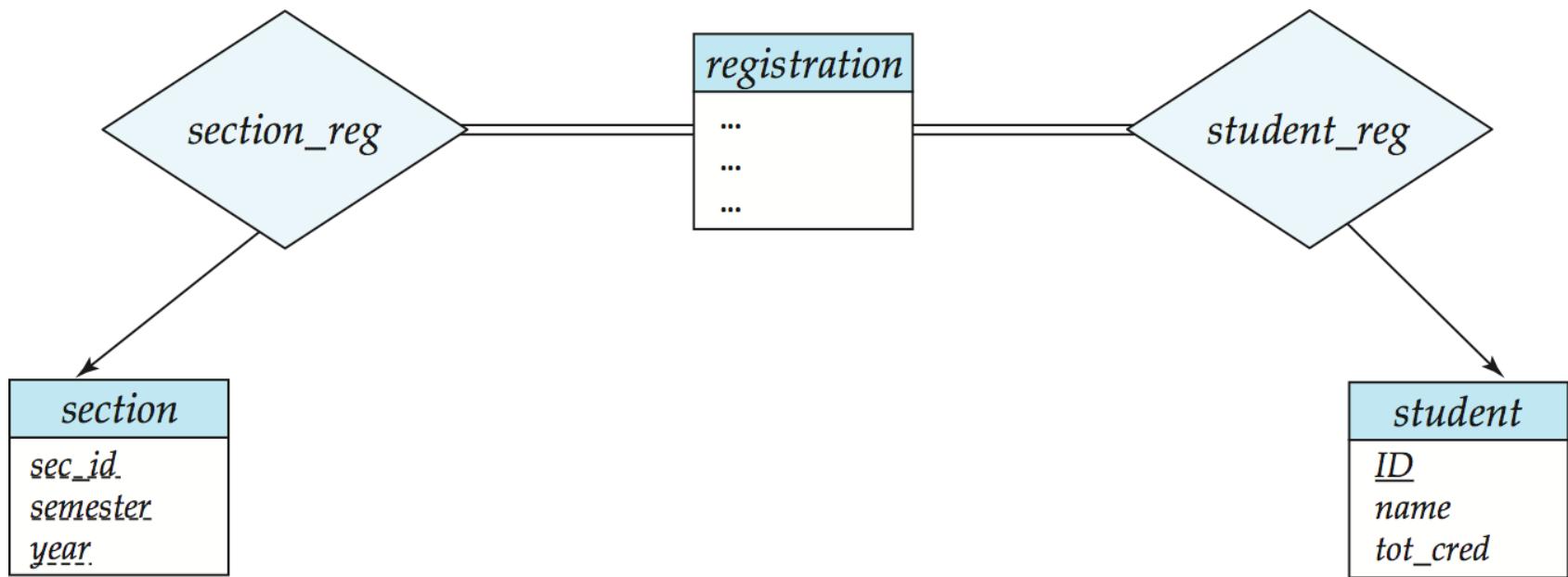
(a)



(b)

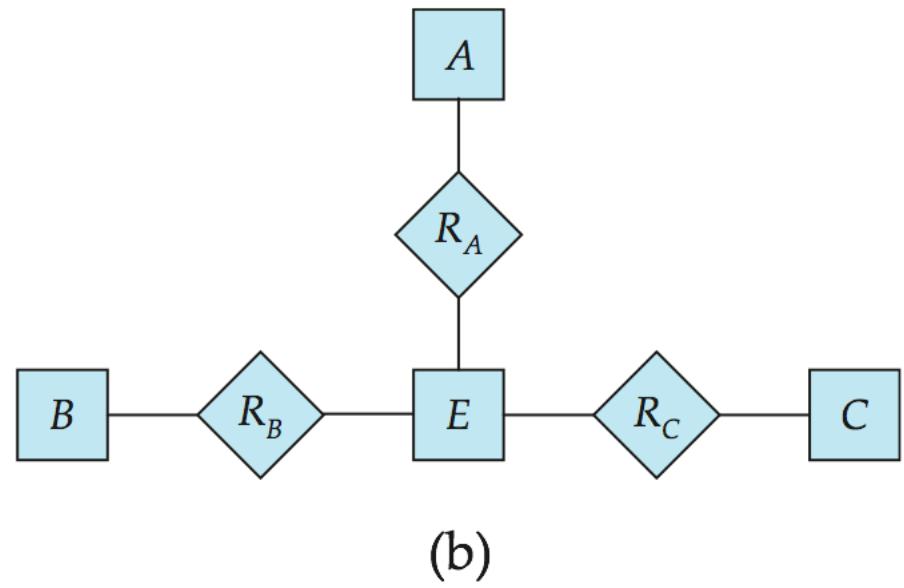
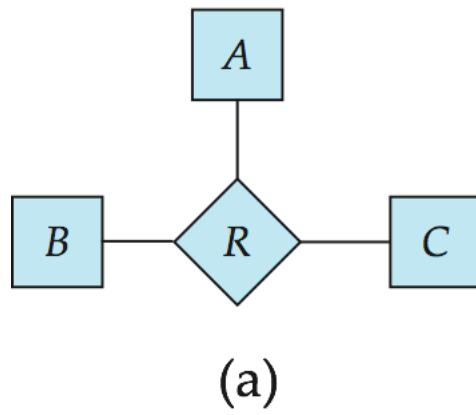


# Figure 7.18



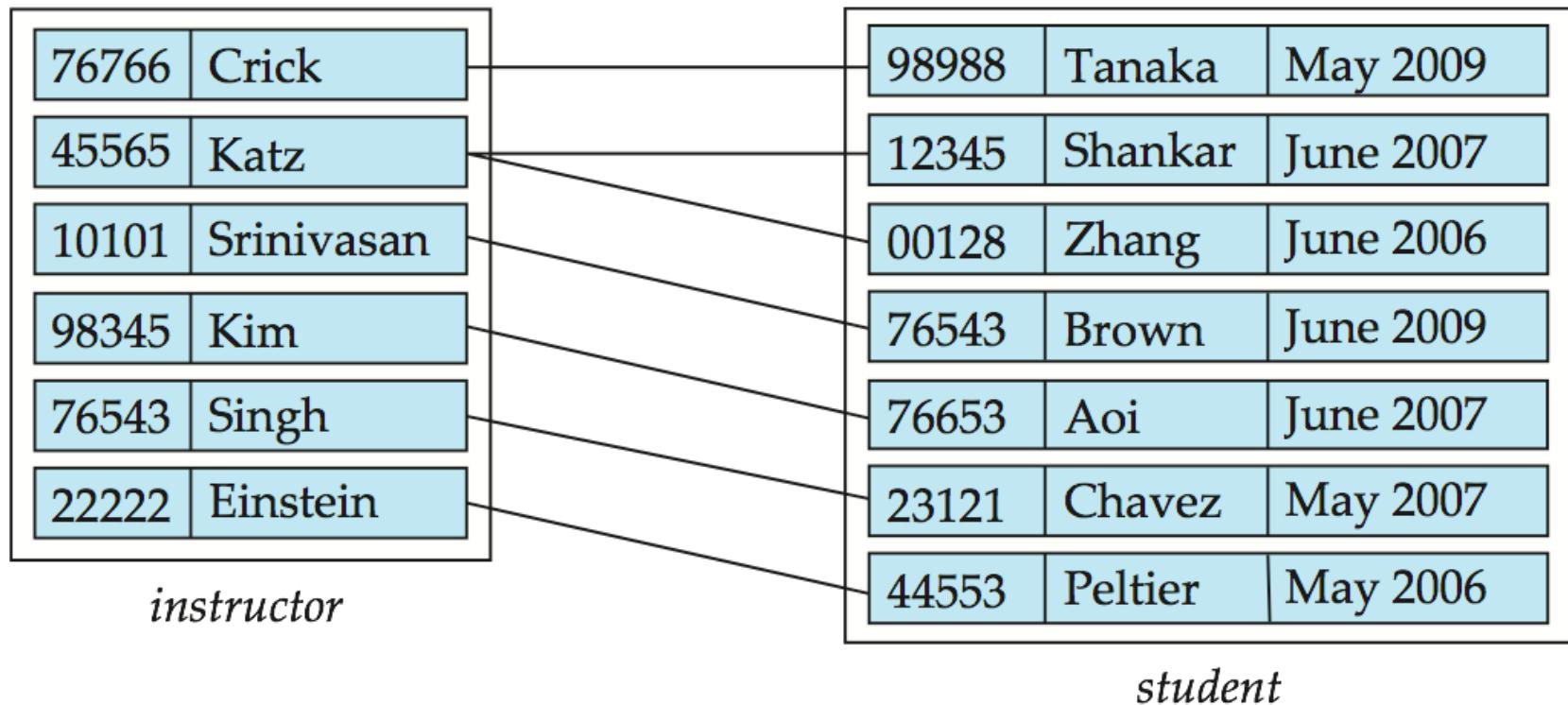


# Figure 7.19



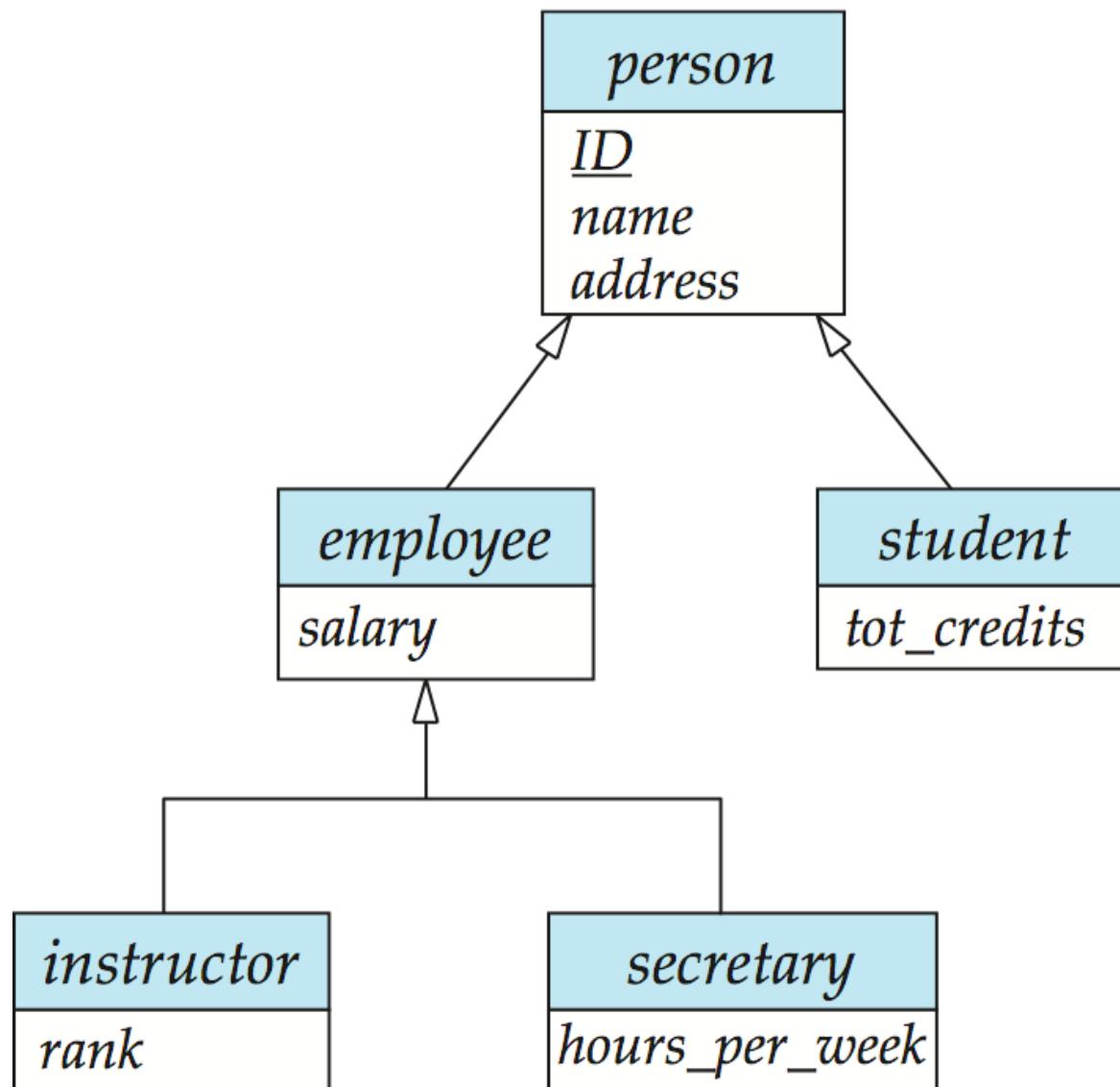


# Figure 7.20



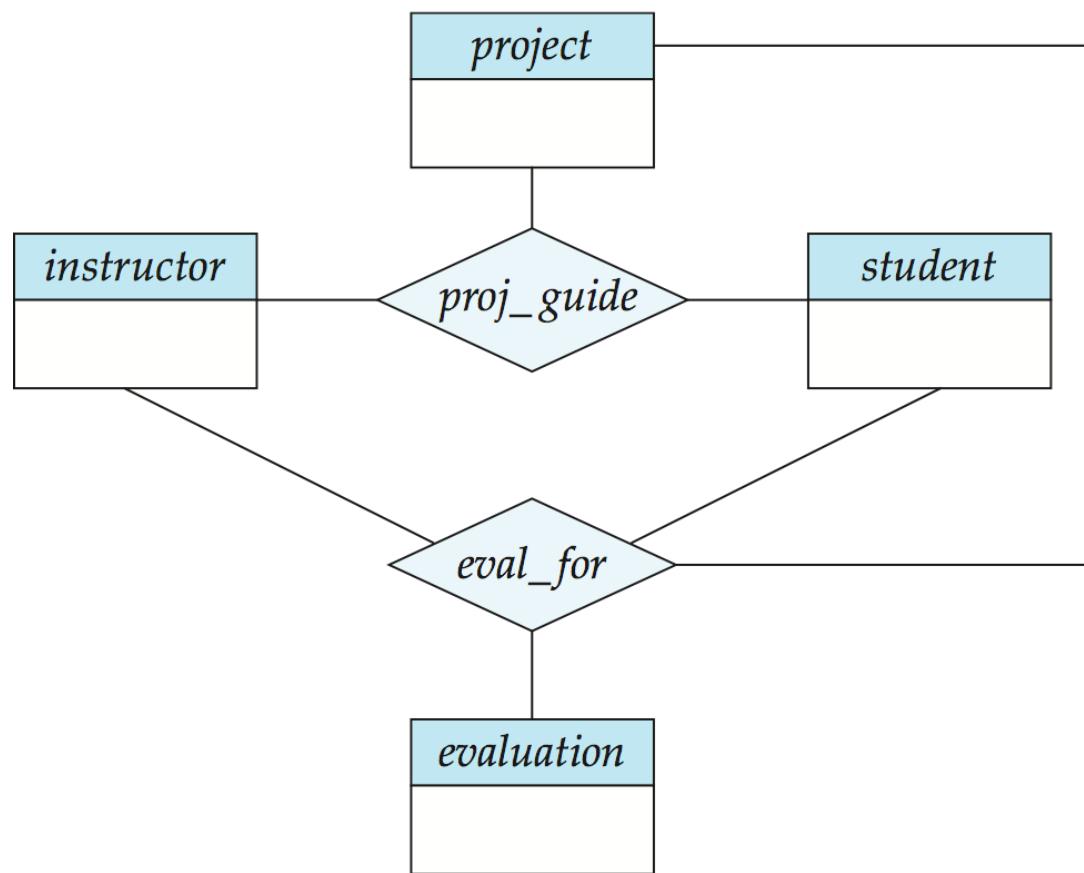


# Figure 7.21



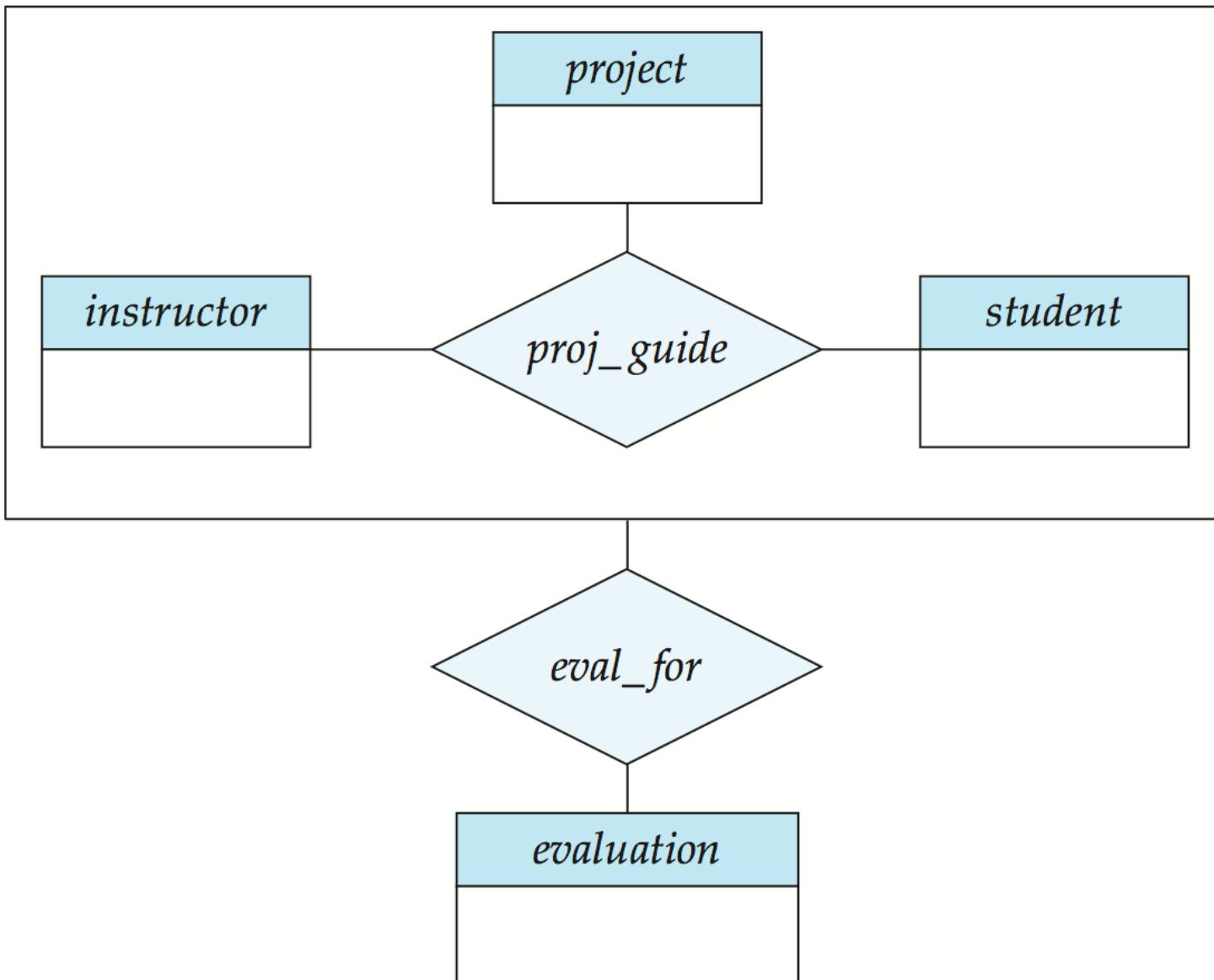


# Figure 7.22



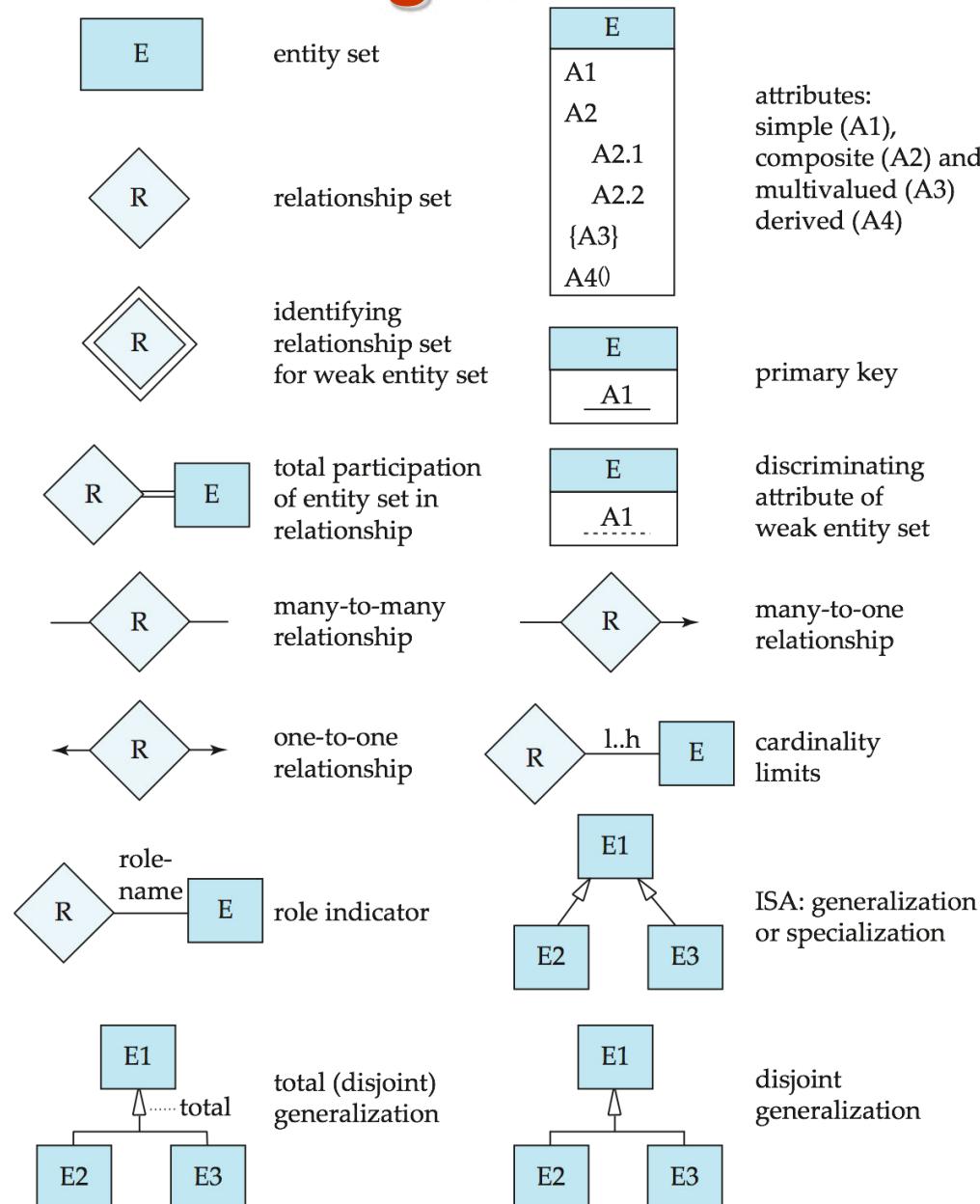


# Figure 7.23





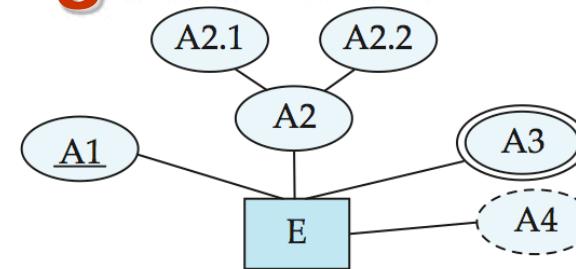
# Figure 7.24



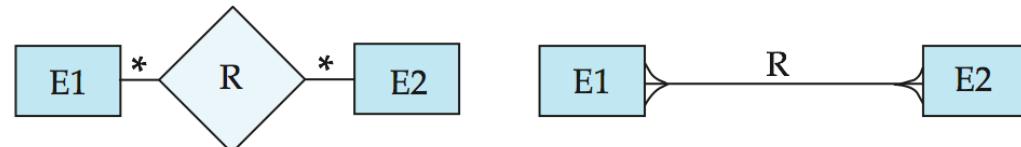


# Figure 7.25

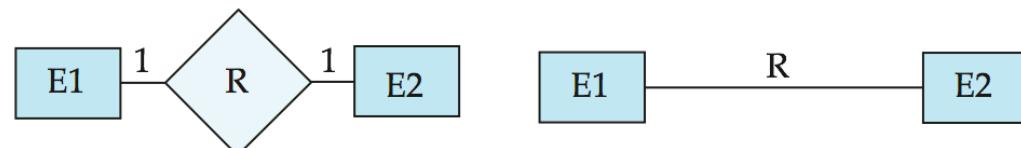
entity set E with simple attribute A1, composite attribute A2, multivalued attribute A3, derived attribute A4, and primary key A1



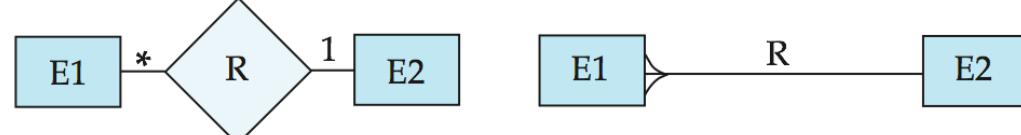
many-to-many relationship



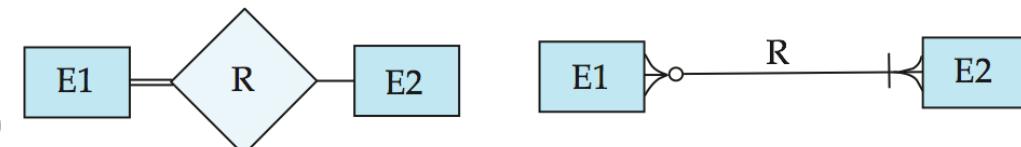
one-to-one relationship



many-to-one relationship



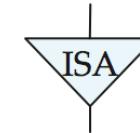
participation in R: total (E1) and partial (E2)



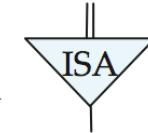
weak entity set



generalization



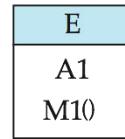
total generalization



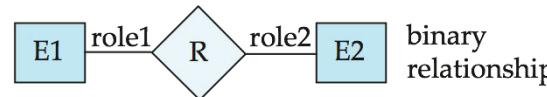


# Figure 7.26

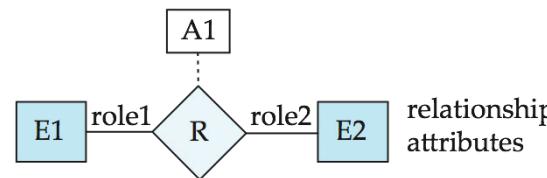
## ER Diagram Notation



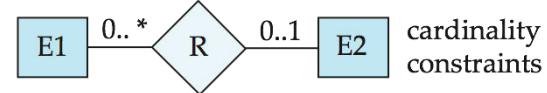
entity with attributes (simple, composite, multivalued, derived)



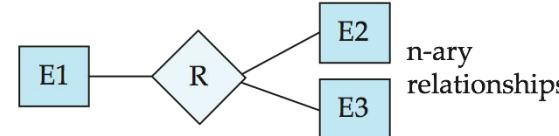
binary relationship



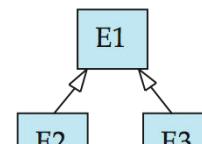
relationship attributes



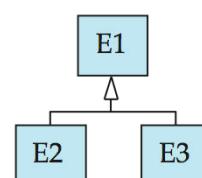
cardinality constraints



n-ary relationships

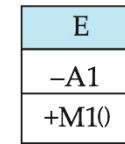


overlapping generalization

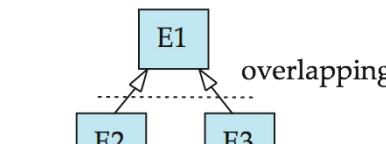
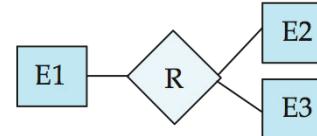
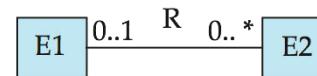
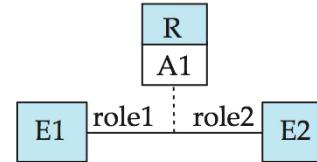
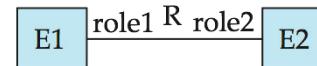


disjoint generalization

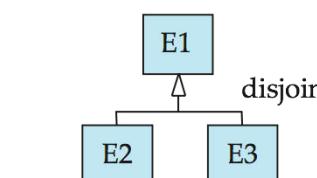
## Equivalent in UML



class with simple attributes and methods (attribute prefixes: + = public, - = private, # = protected)



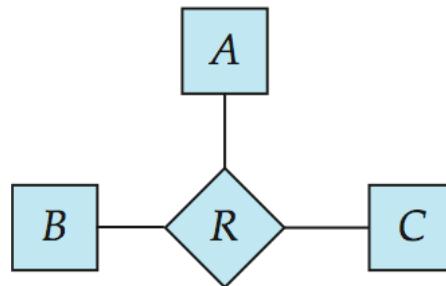
overlapping



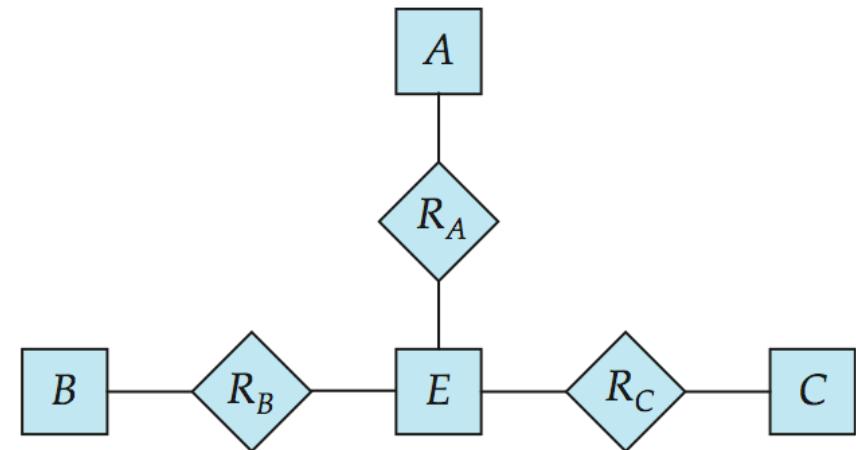
disjoint



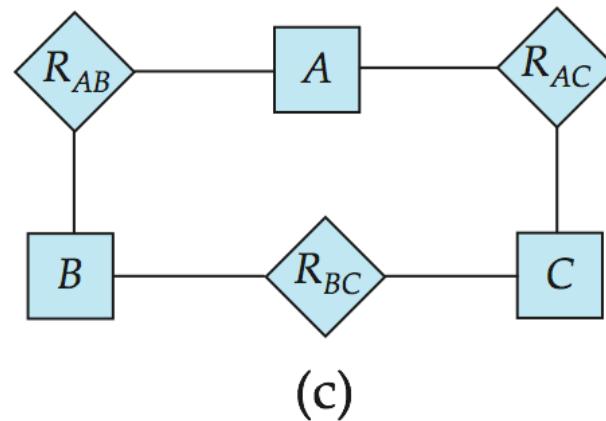
# Figure 7.27



(a)



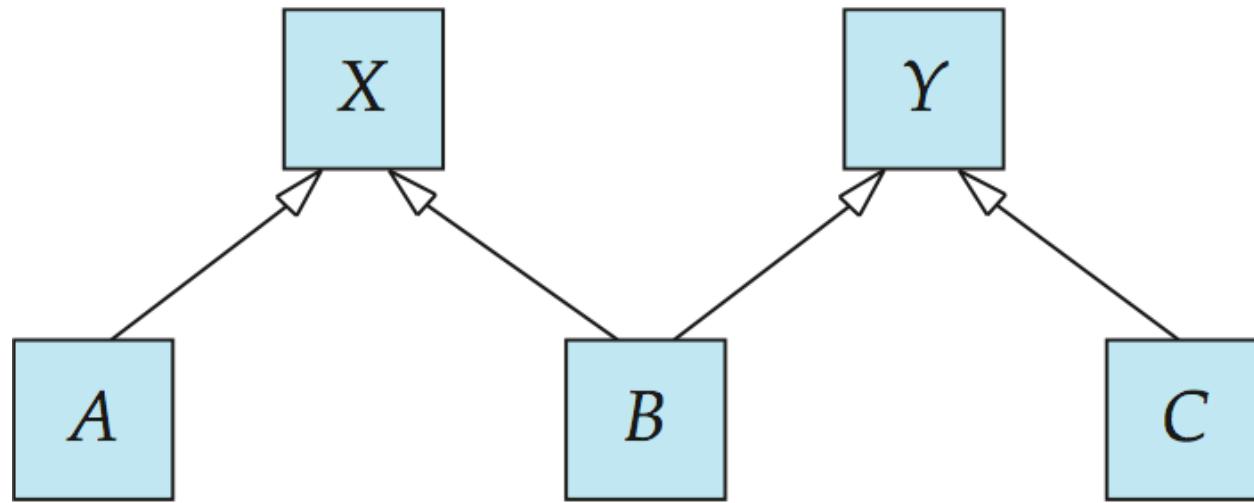
(b)



(c)

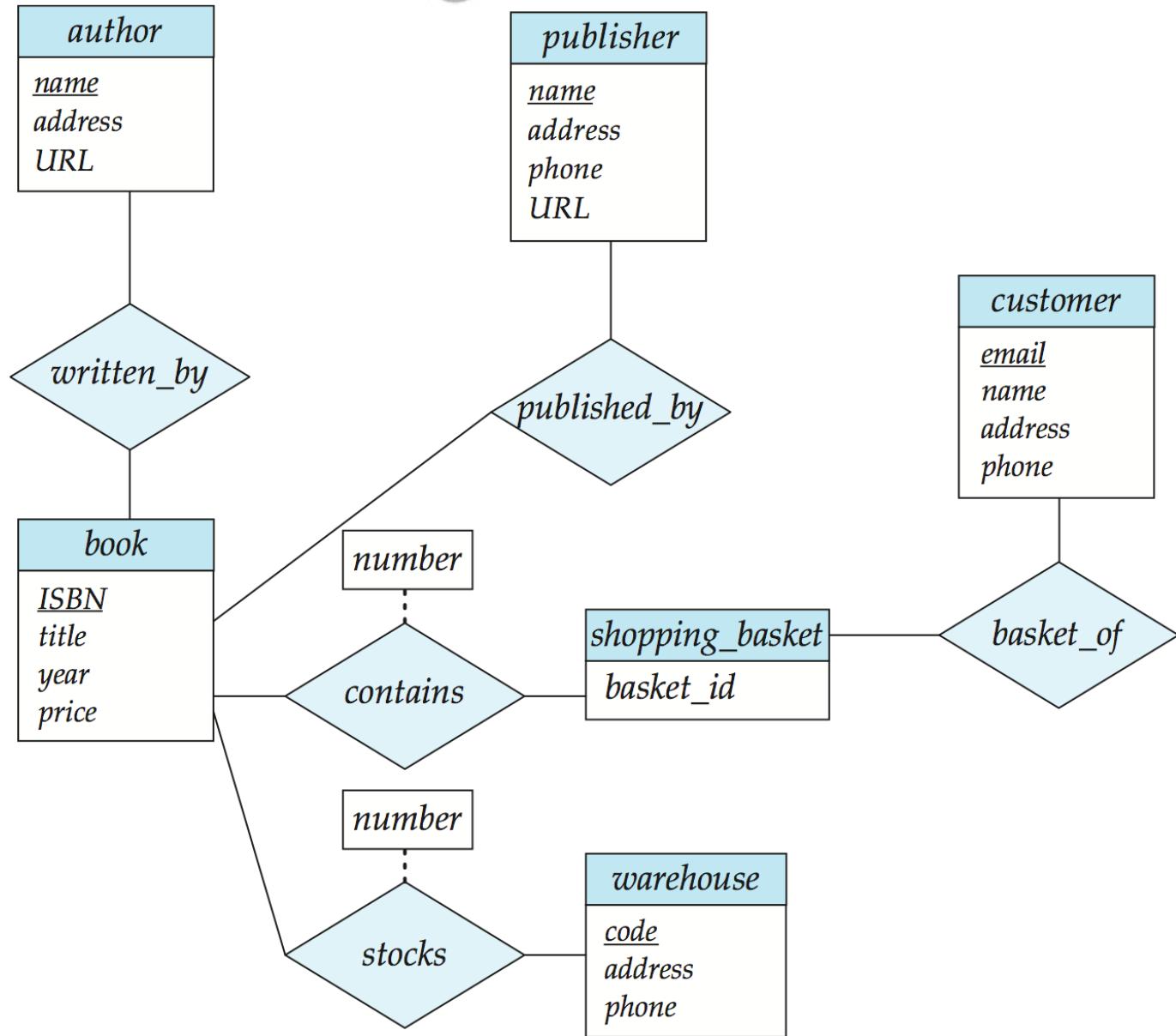


# Figure 7.28





# Figure 7.29





Das Bild kann zurzeit nicht angezeigt werden.

# Chapter 8: Relational Database Design

Database System Concepts, 6<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

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



# Chapter 8: Relational Database Design

- n Features of Good Relational Design
- n Atomic Domains and First Normal Form
- n Decomposition Using Functional Dependencies
- n Functional Dependency Theory
- n Algorithms for Functional Dependencies
- n Higher Normal Forms (I NF, II NF, III NF, and BCNF)
- n Decomposition Using Multivalued Dependencies and 4<sup>th</sup> NF



# Combine Schemas?

- n Suppose we combine *instructor* and *department* into *inst\_dept*
  - | (*No connection to relationship set inst\_dept*)
- n Result is possible repetition of information

| <i>ID</i> | <i>name</i> | <i>salary</i> | <i>dept_name</i> | <i>building</i> | <i>budget</i> |
|-----------|-------------|---------------|------------------|-----------------|---------------|
| 22222     | Einstein    | 95000         | Physics          | Watson          | 70000         |
| 12121     | Wu          | 90000         | Finance          | Painter         | 120000        |
| 32343     | El Said     | 60000         | History          | Painter         | 50000         |
| 45565     | Katz        | 75000         | Comp. Sci.       | Taylor          | 100000        |
| 98345     | Kim         | 80000         | Elec. Eng.       | Taylor          | 85000         |
| 76766     | Crick       | 72000         | Biology          | Watson          | 90000         |
| 10101     | Srinivasan  | 65000         | Comp. Sci.       | Taylor          | 100000        |
| 58583     | Califieri   | 62000         | History          | Painter         | 50000         |
| 83821     | Brandt      | 92000         | Comp. Sci.       | Taylor          | 100000        |
| 15151     | Mozart      | 40000         | Music            | Packard         | 80000         |
| 33456     | Gold        | 87000         | Physics          | Watson          | 70000         |
| 76543     | Singh       | 80000         | Finance          | Painter         | 120000        |



# Anomaly

## Types of Anomalies

- Insert
- Delete
- Update



## Insert Anomaly

- An **Insert Anomaly** occurs when certain attributes cannot be inserted into the database without the presence of other attributes.

## Update Anomaly

- An **Update Anomaly** exists when one or more instances of duplicated data is updated, but not all.

Solution for the problem ?

Schema decomposition



# What About Smaller Schemas?

- Not all decompositions are good. Suppose we decompose  $\text{employee}(ID, name, street, city, salary)$  into

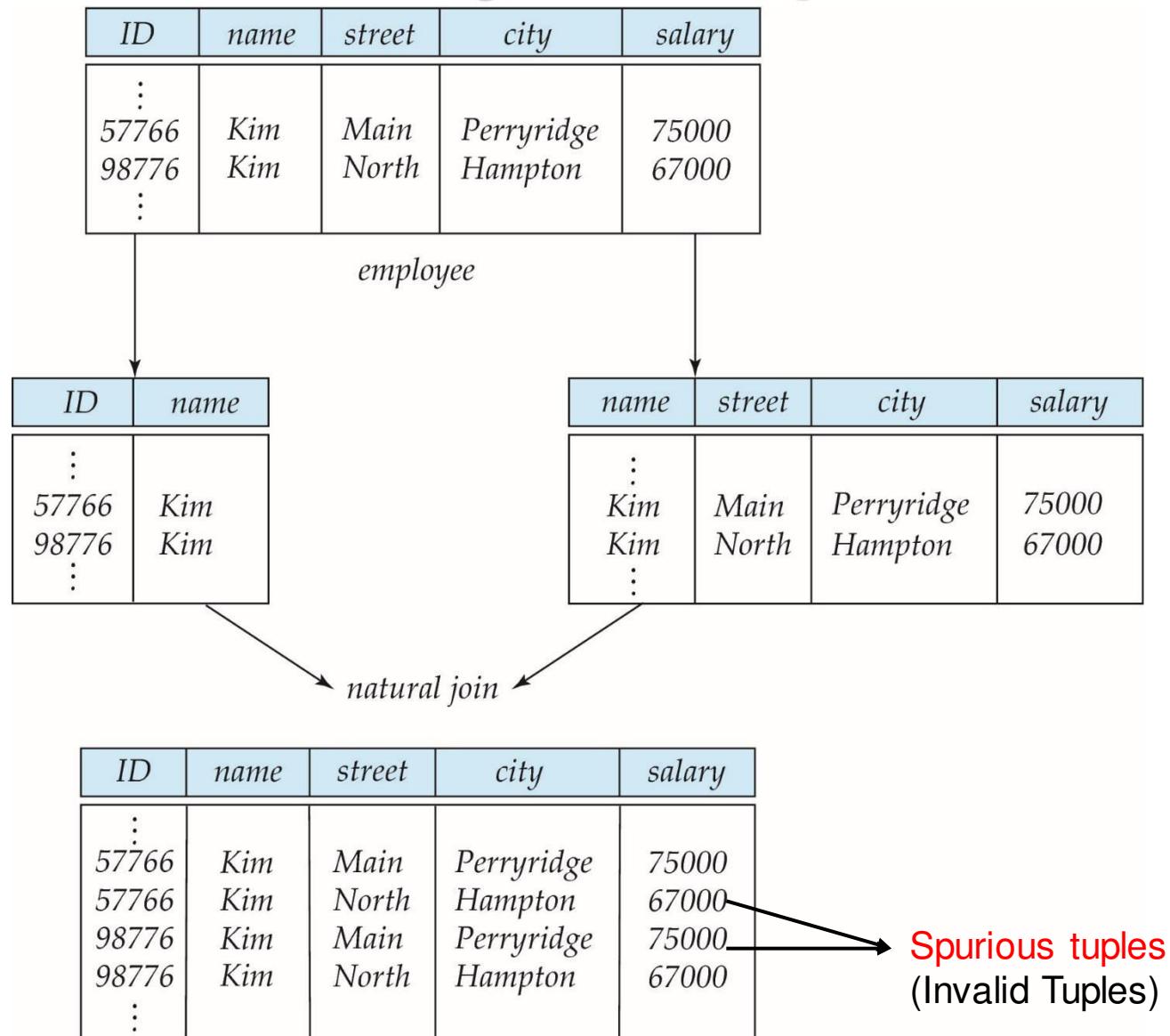
$\text{employee1 } (ID, name)$

$\text{employee2 } (name, street, city, salary)$

- Joining the tables result into **Spurious Tuples** (Invalid Tuples)
- The next slide shows how we lose information -- we cannot reconstruct the original  $\text{employee}$  relation -- and so, this is a **lossy decomposition**.



# A Lossy Decomposition





# Example of Lossless-Join Decomposition

- **Lossless join decomposition**
- Decomposition of  $R = (A, B, C)$   
 $R_1 = (A, B)$      $R_2 = (B, C)$

| A        | B | C |
|----------|---|---|
| $\alpha$ | 1 | A |
| $\beta$  | 2 | B |

$r$

| A        | B |
|----------|---|
| $\alpha$ | 1 |
| $\beta$  | 2 |

$\Pi_{A,B}(r)$

| B | C |
|---|---|
| 1 | A |
| 2 | B |

$\Pi_{B,C}(r)$

$\Pi_A(r) \bowtie \Pi_B(r)$

| A        | B | C |
|----------|---|---|
| $\alpha$ | 1 | A |
| $\beta$  | 2 | B |

No Spurious tuples



# Normalization

- Normalization is a systematic approach of **decomposing** tables to eliminate data **redundancy** and undesirable characteristics like Insertion, Update and Deletion **Anomalies**. And helps in removing spurious tuples.
- It is a **multi-step process** that puts data into tabular form by **removing** duplicated data from the relation tables
- Normal Forms (NF) : NF provides with the theoretical procedures that **determine** a relation's vulnerability towards the inconsistencies (**Redundancy or Anomalies**).
- Normal Forms like **1NF, 2NF, 3NF, BCNF, 4NF, 5NF** and **6NF** are in practice
- The **NFs** are applicable to **individual relations**.
- The entire database is said to be in Normal Form '4NF (for eg. ), if all of its **relations are in 4NF**



# First Normal Form

- Domain is **atomic** if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - ▶ Set of names, composite attributes, Multivalued attributes
- A relational schema R is in **first normal form** if the domains of all attributes of R are **atomic**
- **Non-atomic values** complicate storage and encourage redundant (repeated) storage of data
  - Example: Set of accounts stored with each customer, and set of owners stored with each account



# Goal — Devise a Theory for the Following

- Decide whether a particular relation  $R$  is in “good” form.
- In the case that a relation  $R$  is not in “good” form, decompose it into a set of relations  $\{R_1, R_2, \dots, R_n\}$  such that
  - Each relation is in good form – No redundancy
  - Decomposition is a lossless-join decomposition
  - Decomposition must preserve Functional dependency
- Our theory is based on:
  - Functional dependencies
  - Multivalued dependencies



# Functional Dependencies (FD)

- FD specifies the dependency among the attributes in a relation (R).
- Constraints on the set of legal relations.
- Require that the value for a certain **set of attributes determines uniquely the value for another set of attributes**.
- A functional dependency is a generalization of the notion of a *key*.



# Functional Dependencies (Cont.)

- Let  $R$  be a relation schema

$$\alpha \subseteq R \text{ and } \beta \subseteq R$$

- The **functional dependency**

$$\alpha \rightarrow \beta$$

**holds on**  $R$  if and only if for any legal relations  $r(R)$ , whenever any **two tuples  $t_1$  and  $t_2$**  of  $r$  agree on the attributes  $\alpha$ , they also agree on the attributes  $\beta$ . That is,

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

- Example: Consider  $r(A,B)$  with the following instance of  $r$ .

| A | B |
|---|---|
| 1 | 4 |
| 1 | 5 |
| 3 | 7 |
| 1 | 4 |
| 3 | 7 |
| 3 | 9 |

- On this instance,  $A \rightarrow B$  does **NOT** hold, but  $B \rightarrow A$  does hold.



# Functional Dependencies (Cont.)

- $K$  is a **superkey** for relation schema  $R$  if and only if  $K \rightarrow R$  (*all the attributes of  $R$* )
- $K$  is a **candidate key** for  $R$  if and only if
  - $K \rightarrow R$ , and
  - for no  $\alpha \subset K$ ,  $\alpha \rightarrow R$  (*i.e  $K$  must be minimal*)
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

*inst\_dept (ID, name, salary, dept\_name, building, budget ).*

We expect these functional dependencies to hold:

*dept\_name → building, Budget*

*and ID → building, name, budget, Salary*

but would **not expect the following to hold:**

*dept\_name → salary*



# Functional Dependencies (Cont.)

- The dependency of an attribute on a set of attributes is known as **trivial** functional dependency if the set of attributes includes that attribute.
- [Note: The following dependencies are also trivial:  
 $A \rightarrow A$  &  $B \rightarrow B$ ]
- Example:
  - ▶  $ID, name \rightarrow ID$
  - ▶  $name \rightarrow name$
- In general,  $\alpha \rightarrow \beta$  is trivial if  $\beta \subseteq \alpha$



# Closure Rules of Inference

- Armstrong's Axioms:
  - Reflexivity rule:
    - if  $\alpha$  is a set of attributes and  $\beta$  is contained in  $\alpha$  then  
 $\alpha \rightarrow \beta$
  - Augmentation rule:
    - given  $\alpha \rightarrow \beta$  and another set of attributes  $\gamma$ , then  
 $\gamma\alpha \rightarrow \gamma\beta$
  - Transitivity rule:
    - if  $\alpha \rightarrow \beta$  and  $\beta \rightarrow \gamma$ , then  $\alpha \rightarrow \gamma$



# Closure Rules of Inference Cont.

- Other Rules:
  - Union rule:
    - if  $\alpha \rightarrow \beta$  and  $\alpha \rightarrow \gamma$ , then  $\alpha \rightarrow \beta\gamma$
  - Decomposition rule:
    - if  $\alpha \rightarrow \beta\gamma$ , then  $\alpha \rightarrow \beta$  and  $\alpha \rightarrow \gamma$
  - Pseudotransitivity rule:
    - if  $\alpha \rightarrow \beta$  and  $\gamma\beta \rightarrow \delta$ , then  $\gamma\alpha \rightarrow \delta$



$F = \{A \rightarrow B, A \rightarrow C, CD \rightarrow E, B \rightarrow E, CD \rightarrow F\}$  of R

Can infer following FDs

1.  $A \rightarrow E$ , Using Transitivity Rule (using  $A \rightarrow B$  &  $B \rightarrow E$ )
2.  $A \rightarrow BC$ , Union Rule (using  $A \rightarrow B$ ,  $A \rightarrow C$ )
3.  $CD \rightarrow EF$  Union Rule
4.  $AD \rightarrow E$  Psuedotransitivity Rule (using  $A \rightarrow C$ ,  $CD \rightarrow E$ )



# Closure of a set of Attributes

**Given** a set of attributes  $A_1, \dots, A_n$

The **closure**,  $\{A_1, \dots, A_n\}^+$ , is the set of attributes B  
s.t.  $A_1, \dots, A_n \rightarrow B$

Example:

name  $\rightarrow$  color  
category  $\rightarrow$  department  
color, category  $\rightarrow$  price

Closures:

$$\text{name}^+ = \{\text{name}, \text{color}\}$$

$$\{\text{name}, \text{category}\}^+ = \{\text{name}, \text{category}, \text{color}, \text{department}, \text{price}\}$$

$$\text{color}^+ = \{\text{color}\}$$

2



# Example

In class:

$R(A,B,C,D,E,F)$

|                      |
|----------------------|
| $A, B \rightarrow C$ |
| $A, D \rightarrow E$ |
| $B \rightarrow D$    |
| $A, F \rightarrow B$ |

Compute  $\{A,B\}^+$      $X = \{A, B,$                                       }

Compute  $\{A, F\}^+$      $X = \{A, F,$                                       }

- $\{A,B\}^+ = \{A, B, C, D, E\}$
- $\{A, F\}^+ = \{A, F, B, D, E, C\}$ .    As  $AF \rightarrow R$ ,    Hence, AF is Super Key as well as candidate key.



# Closure of Attribute Sets

- Given a set of attributes  $\alpha$ , define the ***closure*** of  $\alpha$  **under**  $F$  (denoted by  $\alpha^+$ ) as the set of attributes **that are functionally determined by  $\alpha$  under  $F$**
- Algorithm to **compute  $\alpha^+$** , the closure of  $\alpha$  **under  $F$**

```
result := α ;
while (changes to result) do
 for each $\beta \rightarrow \gamma$ in F do
 begin
 if $\beta \subseteq result$ then result := result \cup γ
 end
```



# Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$
- $(AG)^+$ 
  1.  $result = AG$
  2.  $result = ABCG$  ( $A \rightarrow C$  and  $A \rightarrow B$ )
  3.  $result = ABCGH$  ( $CG \rightarrow H$  and  $CG \subseteq AGBC$ )
  4.  $result = ABCGHI$  ( $CG \rightarrow I$  and  $CG \subseteq AGBCH$ )

- Is  $AG$  a candidate key?

To test if  $\alpha$  is a candidate key, we compute  $\alpha^+$ , and check if  $\alpha^+$  contains all attributes of  $R$ .

Does  $AG \rightarrow R$ ? == Is  $(AG)^+ \supseteq R$



# Closure of a Set of Functional Dependencies

- Given a set  $F$  of functional dependencies, there are certain other functional dependencies that are logically implied by  $F$ .
  - For example: If  $A \rightarrow B$  and  $B \rightarrow C$ , then  
we can infer that  $A \rightarrow C$
- The set of all functional dependencies logically implied by  $F$  is the **closure** of  $F$ .
- We denote the *closure* of  $F$  by  $F^+$ .
- $F^+$  is a **superset** of  $F$ .



# Closure of a Set of Functional Dependencies

- We can find  $F^+$ , the closure of  $F$ , by repeatedly applying **Armstrong's Axioms**:
  - if  $\beta \subseteq \alpha$ , then  $\alpha \rightarrow \beta$  **(reflexivity)**
  - if  $\alpha \rightarrow \beta$ , then  $\gamma \alpha \rightarrow \gamma \beta$  **(augmentation)**
  - if  $\alpha \rightarrow \beta$ , and  $\beta \rightarrow \gamma$ , then  $\alpha \rightarrow \gamma$  **(transitivity)**
- These rules are
  - **sound** (generate only functional dependencies that actually hold), and
  - **complete** (generate all functional dependencies that hold).



# Procedure for Computing $F^+$

- To compute the closure of a set of functional dependencies  $F$ :

$$F^+ = F$$

**repeat**

**for each** functional dependency  $f$  in  $F^+$

    apply **reflexivity and augmentation** rules on  $f$

    add the resulting functional dependencies to  $F^+$

**for each** pair of functional dependencies  $f_1$  and  $f_2$  in  $F^+$

**if**  $f_1$  and  $f_2$  can be combined using **transitivity**

**then** add the resulting functional dependency to  $F^+$

**until**  $F^+$  does not change any further

**NOTE:** We shall see an alternative procedure for this task later



# Example

- $R = (A, B, C, G, H, I)$

$F = \{ A \rightarrow B$   
 $\quad A \rightarrow C$   
 $\quad CG \rightarrow H$   
 $\quad CG \rightarrow I$   
 $\quad B \rightarrow H\}$

- some members of  $F^+$

1.  $A \rightarrow H$

by transitivity from  $A \rightarrow B$  and  $B \rightarrow H$

2.  $AG \rightarrow I$

by augmenting  $A \rightarrow C$  with G, to get  $AG \rightarrow CG$   
and then transitivity with  $CG \rightarrow I$

3.  $CG \rightarrow HI$

by augmenting  $CG \rightarrow I$  to infer  $CG \rightarrow CGI$ ,  
and augmenting of  $CG \rightarrow H$  to infer  $CGI \rightarrow HI$ ,  
and then transitivity



# Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

## 1. Testing for Candidate key:

To test if  $\alpha$  is a candidate key, we compute  $\alpha^+$ , and check if  $\alpha^+$  contains all attributes of  $R$ .

## 2. Testing functional dependencies

- To check if a functional dependency  $\alpha \rightarrow \beta$  holds (or, in other words, is in  $F^+$ ), just check if  $\beta \subseteq \alpha^+$ .
- That is, we compute  $\alpha^+$  by using attribute closure, and then check if it contains  $\beta$ .



# Canonical Cover (Minimal cover of F)

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
  1. For example:  $A \rightarrow C$  is redundant in  $F = \{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$

□ Parts of a functional dependency may be redundant

2. E.g.: on RHS:  $\{A \rightarrow B, B \rightarrow C, A \rightarrow CD\}$  can be simplified to

$$\{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$$

*Extraneous attribute*

3. E.g.: on LHS:  $\{A \rightarrow B, B \rightarrow C, AB \rightarrow C\}$  can be simplified to

$$\{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$$

- Intuitively, a canonical cover of F is a “minimal” set of functional dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies
- Refer Korth book for this topic (section : 8.4.3 Canonical Cover)



# Canonical Cover

- A **canonical cover** for  $F$  is a set of dependencies  $F_c$  such that
  - $F$  logically implies all dependencies in  $F_c$ , and
  - $F_c$  logically implies all dependencies in  $F$ , and
  - No functional dependency in  $F_c$  contains an extraneous attribute, and
  - Each left side of functional dependency in  $F_c$  is unique.
- To compute a **canonical cover** for  $F$ :

$F_c = F$   
**repeat**

Use the **union rule** to **replace** any dependencies in  $F_c$   
 $\alpha_1 \rightarrow \beta_1$  and  $\alpha_1 \rightarrow \beta_2$  with  $\alpha_1 \rightarrow \beta_1 \beta_2$

Find a functional dependency  $\alpha \rightarrow \beta$  with an  
**extraneous attribute either in  $\alpha$  or in  $\beta$**

/\* Note: test for extraneous attributes done using  $F_c$ , not  $F^*$  \*/

If an extraneous attribute is found, delete it from  $\alpha \rightarrow \beta$   
**until  $F_c$  does not change**



# Computing a Canonical Cover

□  $R(A, B, C)$

$$\begin{aligned}F = & \{ A \rightarrow BC \\& B \rightarrow C \\& A \rightarrow B \\& AB \rightarrow C \}\end{aligned}$$

□ The canonical cover is:  $F_C = \{ A \rightarrow B \\ B \rightarrow C \}$

□ A canonical cover might not be unique. For instance, consider the set of functional dependencies  $F = \{A \rightarrow BC, B \rightarrow AC, \text{ and } C \rightarrow AB\}$ .

Ans:  $F_C = \{A \rightarrow B, B \rightarrow AC, C \rightarrow B\}$ .

OR  $F_C = \{A \rightarrow C, B \rightarrow C, \text{ and } C \rightarrow AB\}$ .



# Minimal Cover or Irreducible Sets or Canonical cover

1. Consider another set F of functional dependencies:

$$F = \{$$
  
$$A \rightarrow BC$$
  
$$CD \rightarrow E$$
  
$$B \rightarrow D$$
  
$$E \rightarrow A$$
  
$$\}$$

Ans: F itself is Minimal cover.

2. R(WXYZ)  $f = \{X \rightarrow W, WZ \rightarrow XY, Y \rightarrow WXZ\}$

Ans:  $X \rightarrow W, WZ \rightarrow Y, Y \rightarrow XZ$  is Minimal cover



# Examples

1. For example, suppose  $F$  contains  $AB \rightarrow CD$ ,  $A \rightarrow E$ , and  $E \rightarrow C$ .

To check if  $C$  is extraneous in  $AB \rightarrow CD$ , we compute the attribute closure of  $AB$  under  $F = \{AB \rightarrow D, A \rightarrow E, \text{ and } E \rightarrow C\}$ . The closure is  $ABCDE$ , which includes  $CD$ , so we infer that  $C$  is extraneous.

2. Consider the following set  $F$  of functional dependencies on schema  $(A, B, C)$ :

$$A \rightarrow BC$$

$$B \rightarrow C$$

$$A \rightarrow B$$

$$AB \rightarrow C$$

Let us compute the canonical cover for  $F$ .



- There are two functional dependencies with the same set of attributes on the left side of the arrow:

$$A \rightarrow BC$$

$$A \rightarrow B$$

We combine these functional dependencies into  $A \rightarrow BC$ .

- $A$  is extraneous in  $AB \rightarrow C$  because  $F$  logically implies  $(F - \{AB \rightarrow C\}) \cup \{B \rightarrow C\}$ . This assertion is true because  $B \rightarrow C$  is already in our set of functional dependencies.
- $C$  is extraneous in  $A \rightarrow BC$ , since  $A \rightarrow BC$  is logically implied by  $A \rightarrow B$  and  $B \rightarrow C$ .

Thus, our canonical cover is:

$$A \rightarrow B$$

$$B \rightarrow C$$



## FINDING CANDIDATE KEYS

1.  $F = \{ AB \rightarrow C, DE \rightarrow B, CD \rightarrow E \}$

If LHS+ determines R, then LHS+ is candidate key.

Ans: ADB, ADC, ADE

2.  $F = \{ CE \rightarrow D, D \rightarrow B, C \rightarrow A \}$       key: CE

3.  $F = \{ AB \rightarrow C, C \rightarrow D, B \rightarrow EA \}$

Ans: B is candidate key.



# Equivalence of FD sets

□ Given

$$F = \{A \rightarrow C, AC \rightarrow D, E \rightarrow ADH\} \quad G = \{A \rightarrow CD, E \rightarrow AH\}$$

Check whether they are equivalent .

**Solution:** If F covers G and G covers F then they are equivalence.

F covers G means, if all the FD's of G are inferred from F .



# SECOND NORMAL FORM (2NF)

- A relation schema  $R$  is in 2NF if **every non-prime (np) attribute  $A$  in  $R$  is *fully functionally dependent* on the primary or candidate keys of  $R$ .**  
or      there is **no partial functional dependency**. (i.e. part of the key  $\rightarrow$  np)
- Eg:  
Given:  $R(ABCD)$     and       $F = AB \rightarrow C$      $B \rightarrow D$   
 $R$  is in 2NF ?
- Steps:  
Find the candidate keys , prime and non-prime attributes :  
 $AB$  is a key  
A and B are **prime (p)** ; C and D are **non-prime (np)** attributes
- Solution to keep the table in 2NF: Decompose the table  $R1(ABC)$  and  $R2(BD)$
- Reason ?



# 3NF

- A relation (or a table, say R ) is in third normal form if it satisfies the following conditions: R is in second normal form and There is **no transitive functional dependency**. (i.e.  $np \rightarrow np$  shouldn't be there)

- Eg:

Given R(ABCD) and F= {AB->C C->D}

- Steps:

Find the candidate keys , prime and non-prime attributes :

Ans: AB is a key

- **Solution:** decompose the table by having separate table for each **Transitive functional dependency**, which violates 3NF.
- R1(ABC) and R2(CD)



# BCNF (Boyce–Codd normal form)

- BCNF: Consider the case where  $p/np \rightarrow$  prime attribute is not covered or handled in 2NF or 3NF.  
BCNF handles this.

BCNF says that LHS must be a super key /Key , RHS can be anything.

- Eg:
  - Given:  $R(A,B,C)$  and  $F=\{AB \rightarrow C, C \rightarrow B\}$
  - Find key: AB and AC

There is no Partial and Transitivity dependency. Hence R is in 3NF.

Is R in BCNF ?

Solution: Decompose the table.  $R2(\underline{C},B)$   $R1(A,C)$

Note: FD  $AB \rightarrow C$ , is not preserved



- BCNF----LHS is super key
- 3NF—no transitive dependency--- $np \rightarrow np$
- 2NF---no partial dependency--- $p \rightarrow np$  ( $p$  is part of candidate key)



# Identify in which normal form the R is ?

Given:  $R(ABCED)$      $F=AB \rightarrow CD$      $D \rightarrow A$      $BC \rightarrow DE$

Identify in which normal form the R is ?

□ Steps:

Find Key & Prime and Non-prime attributes. : AB, BC, BD

Check from Highest NF.

Ans: It is in 3NF

$R(UWXYZ)$      $X \rightarrow UY$      $Y \rightarrow Z$      $Z \rightarrow Y$      $UW \rightarrow X$     ?

key : UW and XW;    Hence prime attributes are UWX.

Ans: It is in 1NF



## decompose R to keep it in highest NF

- Given:  $R(ABCDEFGHIJ)$     $AB \rightarrow C$     $A \rightarrow DE$ ,  $B \rightarrow F$ ,  $D \rightarrow IJ$ ,  $F \rightarrow GH$

Keep it in Highest NF.

Solution:

Find the Key : Ans - Key is AB

Check in which NF the R is in : Ans - It is in 1NF

Decompose the table (**Start from the lowest NF**)

Ans:       $R1(A B, C)$

$R21(A D E)$   $R22(D I J)$

$R31(BF)$   $R32(FGH)$

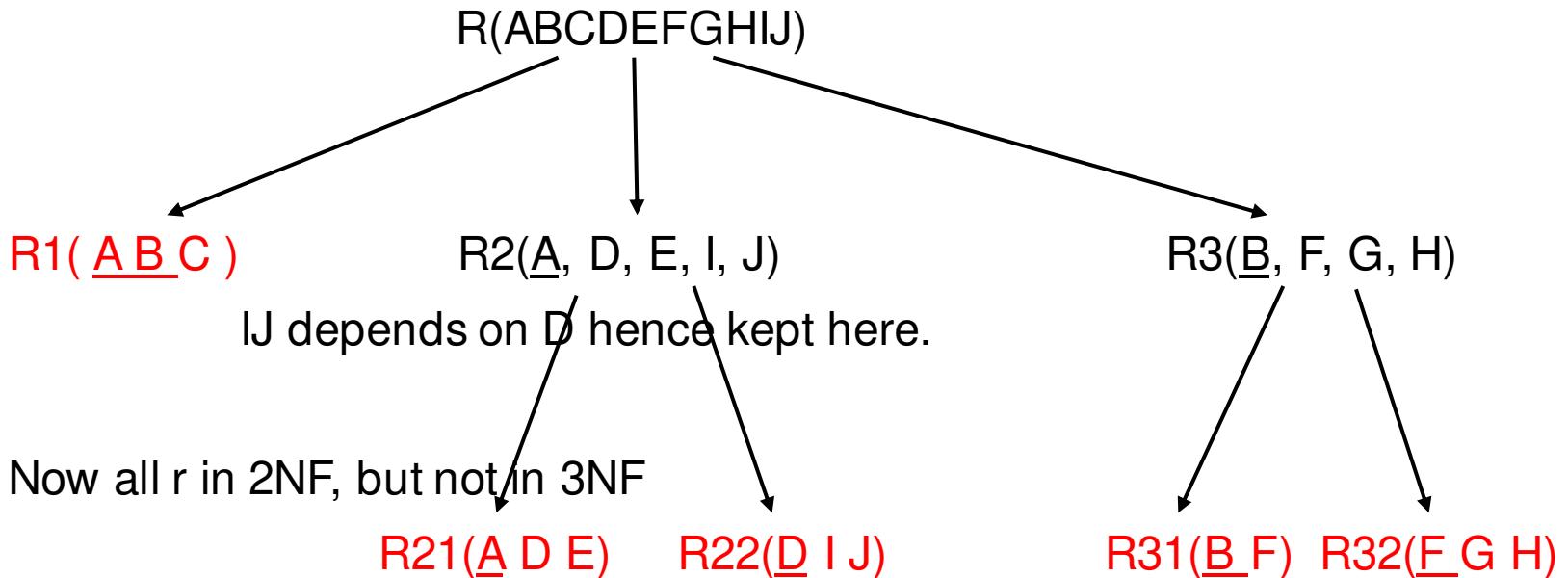
Now all are in BCNF.



# Decomposition Steps

- Given :  $R(ABCDEFGHIJ)$     $AB \rightarrow C$     $A \rightarrow DE$ ,  $B \rightarrow F$ ,  $D \rightarrow IJ$ ,  $F \rightarrow GH$

Decomposition:



Now all r in BCNF



## **decompose R to keep it in highest NF**

- R: CarReg, Hiredate, Make, Model, cutsno, Custnam, outletno, outletloc
- F: CarReg, Hiredate-> Custno, CustNo->CustName, model->Make , outletNo->outletloc, carreg->Model, Outletno
- Solution:
  - R11(castno, caustname)
  - R12(carreg, hiredate, custno)
  - R21(carreg, mode, outletno)
  - R22(model, make)
  - R23(Outletno, outletloc)



## *Test for preserving FD*

Given:  $R(A \dots F)$   $F = \{ A \rightarrow BCDEF \quad BC \rightarrow ADEF \quad B \rightarrow F \quad D \rightarrow E \}$   
 $R1(ABCD) \quad R2(BF) \quad R3(DE)$

□ Solution:

|                     |                   |                   |
|---------------------|-------------------|-------------------|
| $R1(ABCD)$          | $R2(BF)$          | $R3(DE)$          |
| $A \rightarrow BCD$ | $B \rightarrow F$ | $D \rightarrow E$ |
| $BC \rightarrow AD$ |                   |                   |

Let us check for  $A \rightarrow EF$  and  $BC \rightarrow EF$  using preserved dependencies

Compute  $A^+$  and  $BC^+$  using already preserved FDs  
Now, All the FD's are preserved



## *Test for preserving FD*

- Given:

$R(C S Z)$      $F=\{ CS \rightarrow Z \text{ and } Z \rightarrow C \}$

Decomposed into:  $R1(C S )$  and  $R2(Z C)$

Steps:

Here  $Z \rightarrow C$  is preserved

Now compute  $CS+=CS$  **using preserved dependencies.**

We can't derive  $CS \rightarrow Z$  hence, this is not dependency preserving.



Given:

$R(A,B,C)$  and  $F=\{C \rightarrow B, AB \rightarrow C\}$

key:  $AB$  and  $AC$

Decomposed into:  $R_2(C,B)$   $R_1(A,C)$

$C \rightarrow B$  is preserved

Now compute  $AB+=AB$  using preserved dependencies.

We can't derive  $AB \rightarrow C$  hence, this is not dependency preserving.



# Loss-less decomposition test

- 1. Given       $R(A B C D)$   
 $R1(B C) \quad R2(C A D)$   
 $C \rightarrow D, B \rightarrow C, C \rightarrow A$   
Ans: Loss-less decomposition
  
- 2. Given:  
 $R = (B, O, I, S, Q, D, T, P).$   
 $T \rightarrow P, I \rightarrow B, Q \rightarrow O, S \rightarrow D, IS \rightarrow Q$   
 $R1 (IB), \quad R2 (ISQ), \quad R3 (ISDO), \quad R4 (QSO), \quad R5 (TP).$   
Ans: Lossy decomposition
  
- **Solution:** Using matrix method



# Multivalued Dependencies (MVD)

University courses

| <u>Course</u> | <u>Book</u>  | <u>Lecturer</u> |
|---------------|--------------|-----------------|
| AHA           | Silberschatz | John D          |
| AHA           | Nederpelt    | John D          |
| AHA           | Silberschatz | William M       |
| AHA           | Nederpelt    | William M       |
| AHA           | Silberschatz | Christian G     |
| AHA           | Nederpelt    | Christian G     |
| OSO           | Silberschatz | John D          |
| OSO           | Silberschatz | William M       |

MVDs occur when **two or more independent multi valued facts about the same attribute occur within the same table.**

MVDs are : Course  $\rightarrow\!\!\!>$  Book and Course  $\rightarrow\!\!\!>$  Lecturer



## 4 NF

- A relation R is in 4NF if and only if the following conditions are satisfied:
  1. It should be in the Boyce-Codd Normal Form (BCNF).
  2. the table should **not have any Multi-valued Dependency**.

**University courses**

| <u>Course</u> | <u>Book</u>  | <u>Lecturer</u> |
|---------------|--------------|-----------------|
| AHA           | Silberschatz | John D          |
| AHA           | Nederpelt    | John D          |
| AHA           | Silberschatz | William M       |
| AHA           | Nederpelt    | William M       |
| AHA           | Silberschatz | Christian G     |
| AHA           | Nederpelt    | Christian G     |
| OSO           | Silberschatz | John D          |
| OSO           | Silberschatz | William M       |



| Course_Student_Book |              |                     |
|---------------------|--------------|---------------------|
| Course              | Student_Name | Text_Book           |
| Phyics              | Ankit        | Mechanics           |
| Phyics              | Ankit        | Optics              |
| Phyics              | Rahat        | Mechanics           |
| Phyics              | Rahat        | Optics              |
| Chemistry           | Ankit        | Organic_chemistry   |
| Chemistry           | Ankit        | InOrganic_chemistry |
| English             | Raj          | English_Literature  |
| English             | Raj          | English_Grammer     |

- **Following MVDs exists:** Course --> Student\_name a and Course --> Text book  
Here, Student\_name and Text\_book are independent of each other.
- Drawback: **Anomalies**
- Is it in BCNF ?
- What is the Key ?
- **Solution:** To put it into 4NF, two separate tables are formed as shown below:
  - COURSE\_STUDENT (Course, Student\_name)
  - COURSE\_BOOK (Course, text\_book)
- 4NF : R is already in BCNF and must not contain multi-valued dependencies



- A relation R having A, B, and C, as attributes can be non loss-decomposed into two projections R1(A,B) and R2(A,C) if and only if the MVD  $A \rightarrow\!\!> B|C$  hold in R.
- Looking again at the un-decomposed COURSE\_STUDENT\_BOOK table, it contains a multi-valued dependency as shown below:
  - Course  $\rightarrow\!\!> Student\_name$
  - Course  $\rightarrow\!\!> Text\_book$
- To put it into 4NF, two separate tables are formed as shown below:
  - COURSE\_STUDENT (Course, Student\_name)
  - COURSE\_BOOK (Course, text\_book)



# Multivalued Dependencies

- Suppose we record names of children, and phone numbers for instructors:
  - $inst\_child(ID, child\_name)$
  - $inst\_phone(ID, phone\_number)$
- If we were to combine these schemas to get
  - $inst\_info(ID, child\_name, phone\_number)$
  - Example data:
    - (99999, David, 512-555-1234)
    - (99999, David, 512-555-4321)
    - (99999, William, 512-555-1234)
    - (99999, William, 512-555-4321)
- This relation is in BCNF
  - Why?



# Multivalued Dependencies (MVDs)

- Let  $R$  be a relation schema and let  $\alpha \subseteq R$  and  $\beta \subseteq R$ . The **multivalued dependency**

$$\alpha \rightarrow\!\!\!\rightarrow \beta$$

holds on  $R$  if in any legal relation  $r(R)$ , for all pairs for tuples  $t_1$  and  $t_2$  in  $r$  such that  $t_1[\alpha] = t_2[\alpha]$ , there exist tuples  $t_3$  and  $t_4$  in  $r$  such that:

$$\begin{aligned}t_1[\alpha] &= t_2[\alpha] = t_3[\alpha] = t_4[\alpha] \\t_3[\beta] &= t_1[\beta] \\t_3[R - \beta] &= t_2[R - \beta] \\t_4[\beta] &= t_2[\beta] \\t_4[R - \beta] &= t_1[R - \beta]\end{aligned}$$

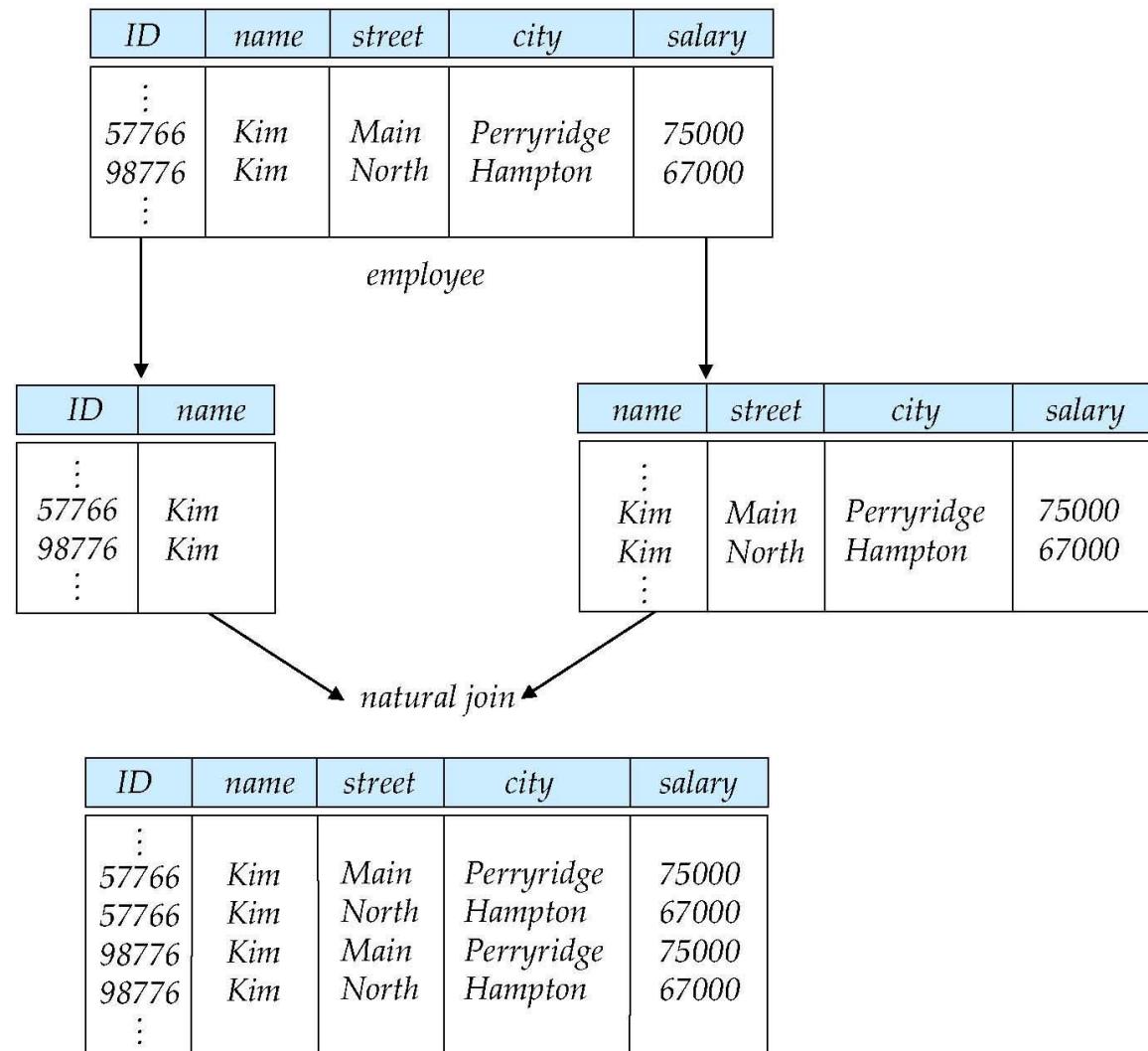


# Figure 8.02

| <i>ID</i> | <i>name</i> | <i>salary</i> | <i>dept_name</i> | <i>building</i> | <i>budget</i> |
|-----------|-------------|---------------|------------------|-----------------|---------------|
| 22222     | Einstein    | 95000         | Physics          | Watson          | 70000         |
| 12121     | Wu          | 90000         | Finance          | Painter         | 120000        |
| 32343     | El Said     | 60000         | History          | Painter         | 50000         |
| 45565     | Katz        | 75000         | Comp. Sci.       | Taylor          | 100000        |
| 98345     | Kim         | 80000         | Elec. Eng.       | Taylor          | 85000         |
| 76766     | Crick       | 72000         | Biology          | Watson          | 90000         |
| 10101     | Srinivasan  | 65000         | Comp. Sci.       | Taylor          | 100000        |
| 58583     | Califieri   | 62000         | History          | Painter         | 50000         |
| 83821     | Brandt      | 92000         | Comp. Sci.       | Taylor          | 100000        |
| 15151     | Mozart      | 40000         | Music            | Packard         | 80000         |
| 33456     | Gold        | 87000         | Physics          | Watson          | 70000         |
| 76543     | Singh       | 80000         | Finance          | Painter         | 120000        |



# Figure 8.03





# Figure 8.04

| A     | B     | C     | D     |
|-------|-------|-------|-------|
| $a_1$ | $b_1$ | $c_1$ | $d_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_2$ |
| $a_2$ | $b_2$ | $c_2$ | $d_2$ |
| $a_2$ | $b_3$ | $c_2$ | $d_3$ |
| $a_3$ | $b_3$ | $c_2$ | $d_4$ |

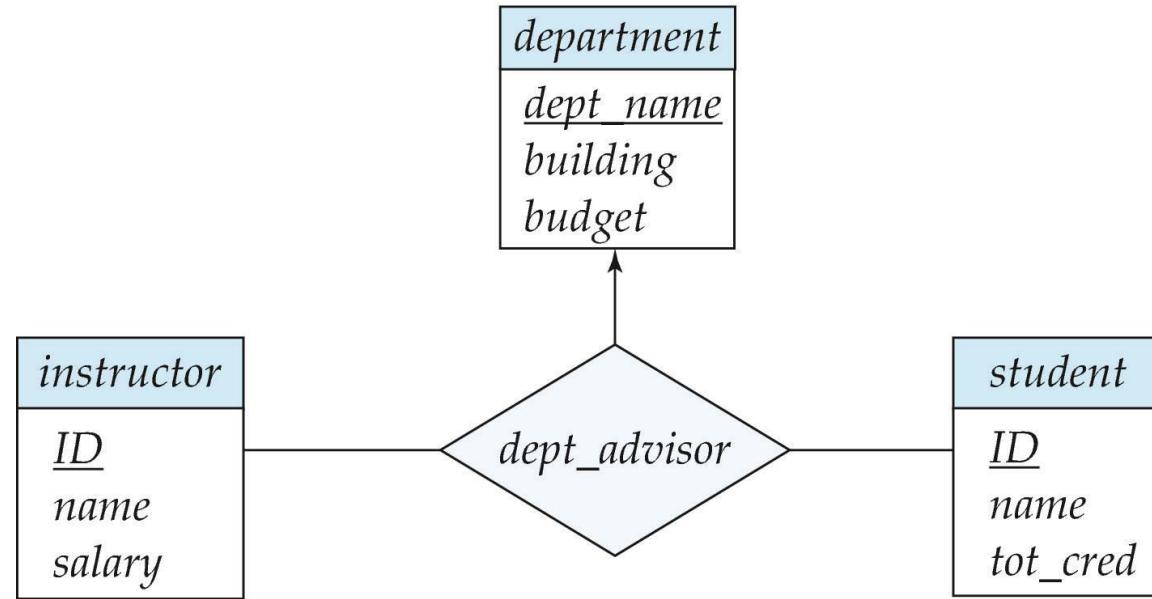


# Figure 8.05

| <i>building</i> | <i>room_number</i> | <i>capacity</i> |
|-----------------|--------------------|-----------------|
| Packard         | 101                | 500             |
| Painter         | 514                | 10              |
| Taylor          | 3128               | 70              |
| Watson          | 100                | 30              |
| Watson          | 120                | 50              |



# Figure 8.06





# Figure 8.14

| <i>dept_name</i> | <i>ID</i> | <i>street</i> | <i>city</i> |
|------------------|-----------|---------------|-------------|
| Physics          | 22222     | North         | Rye         |
| Physics          | 22222     | Main          | Manchester  |
| Finance          | 12121     | Lake          | Horseneck   |



# Figure 8.15

| <i>dept_name</i> | <i>ID</i> | <i>street</i> | <i>city</i> |
|------------------|-----------|---------------|-------------|
| Physics          | 22222     | North         | Rye         |
| Math             | 22222     | Main          | Manchester  |



# Figure 8.17

| A     | B     | C     |
|-------|-------|-------|
| $a_1$ | $b_1$ | $c_1$ |
| $a_1$ | $b_1$ | $c_2$ |
| $a_2$ | $b_1$ | $c_1$ |
| $a_2$ | $b_1$ | $c_3$ |



Das Bild kann zurzeit nicht angezeigt werden.

# Chapter 14: Transactions

Database System Concepts, 6<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

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



# Chapter 14: Transactions

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.



# Transactions

A series of one or more **SQL statements that are logically related** are termed as a **Transaction**.

E.g:

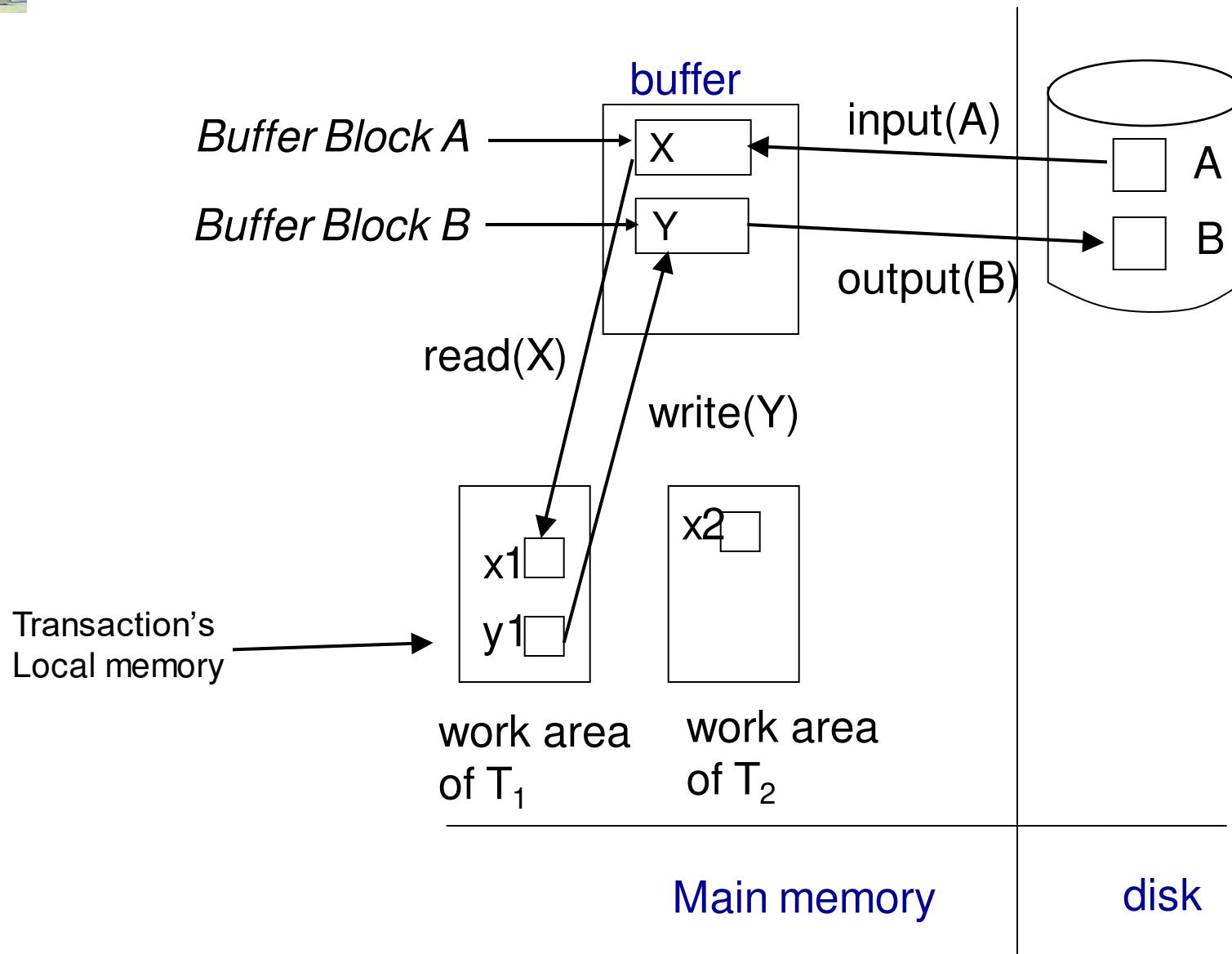
1. transaction to transfer \$50 from account A to account B:

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

2. **Withdraw** an amount 2,000 and **deposit** 10,000 for all the accounts. If the sum of balance of all accounts exceeds 2,00,000 then **undo the deposit just made**.



# Example of Data Access





# Transaction Issues

- Two main issues to deal with Transaction execution:  
It must make sure that Database is consistent state

- Failures of various kinds, such as hardware failures and system crashes : Leads to inconsistent database state
- Concurrent execution of multiple transactions



# ACID Properties

To preserve the integrity (consistency) of data the database system must ensure:

- **Atomicity.** Either all operations( or instructions) of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in **isolation** preserves the consistency of the database.
- **Isolation.** Although multiple transactions may **execute concurrently**, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions  $T_i$  and  $T_j$ , it appears to  $T_i$  that either  $T_j$  finished execution before  $T_i$  started, or  $T_j$  started execution after  $T_i$  finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the **database persist**, even if there are system failures.



# Example of Fund Transfer

- 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(R.Error in Query) or hardware(S/W crash)
- 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.



# Example of Fund Transfer (Cont.)

- 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)**
- **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



# Example of Fund Transfer (Cont.)

- **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.



# Concurrent Executions

- Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run **serially—that is, one at a time, each** starting only after the previous one has completed.
- Multiple transactions are allowed to run concurrently in the system.  
**Advantages are:**
  - **increased processor and disk utilization**, leading to better transaction *throughput*
    - ▶ E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time/waiting time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the **consistency of the database**.

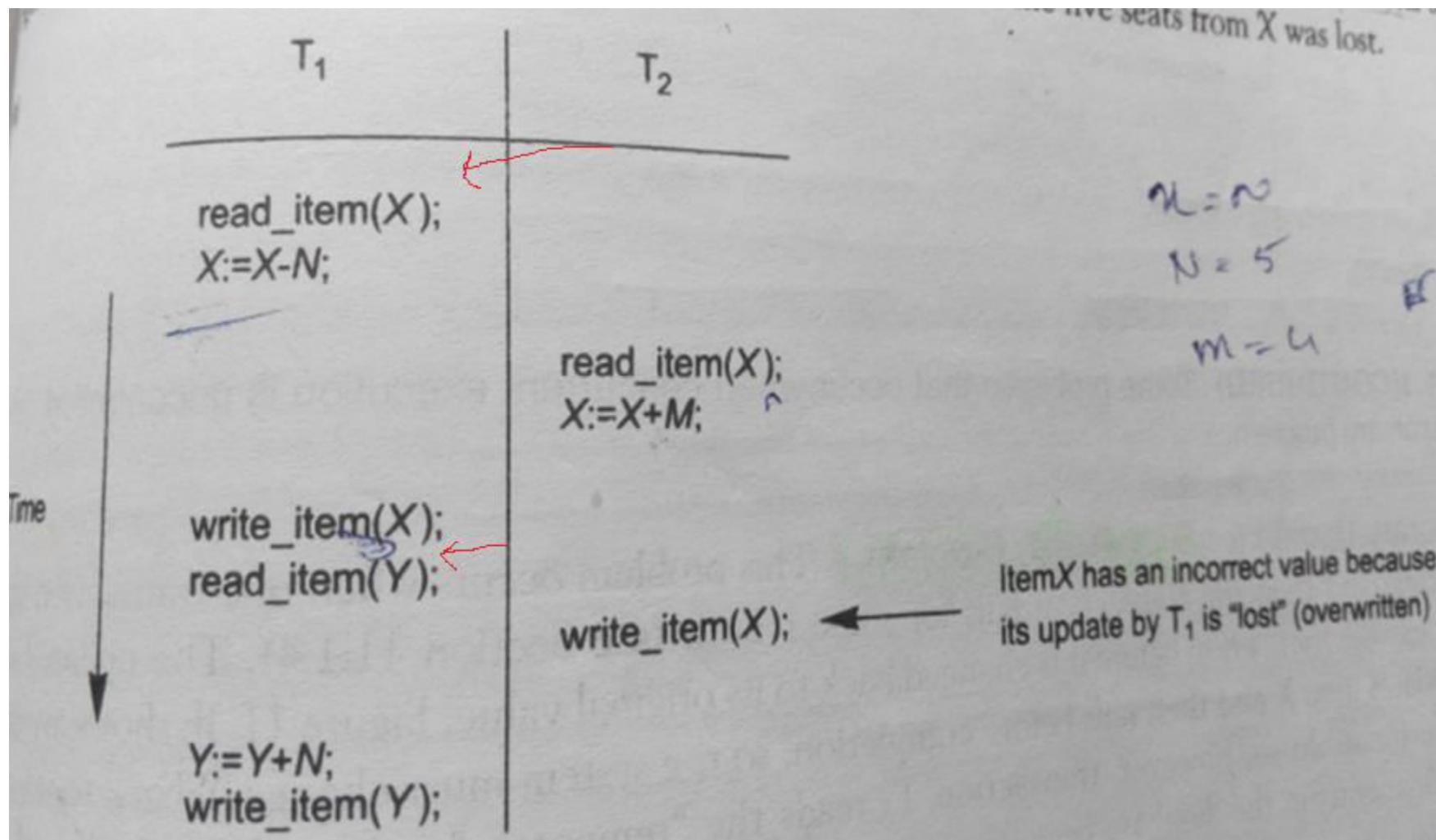


## **Problems faced when concurrent transactions are executed in an uncontrolled manner**

1. The **Lost Update** Problem
2. The temporary Update problem or **Dirty Read** problem
3. The **incorrect Summary** Problem

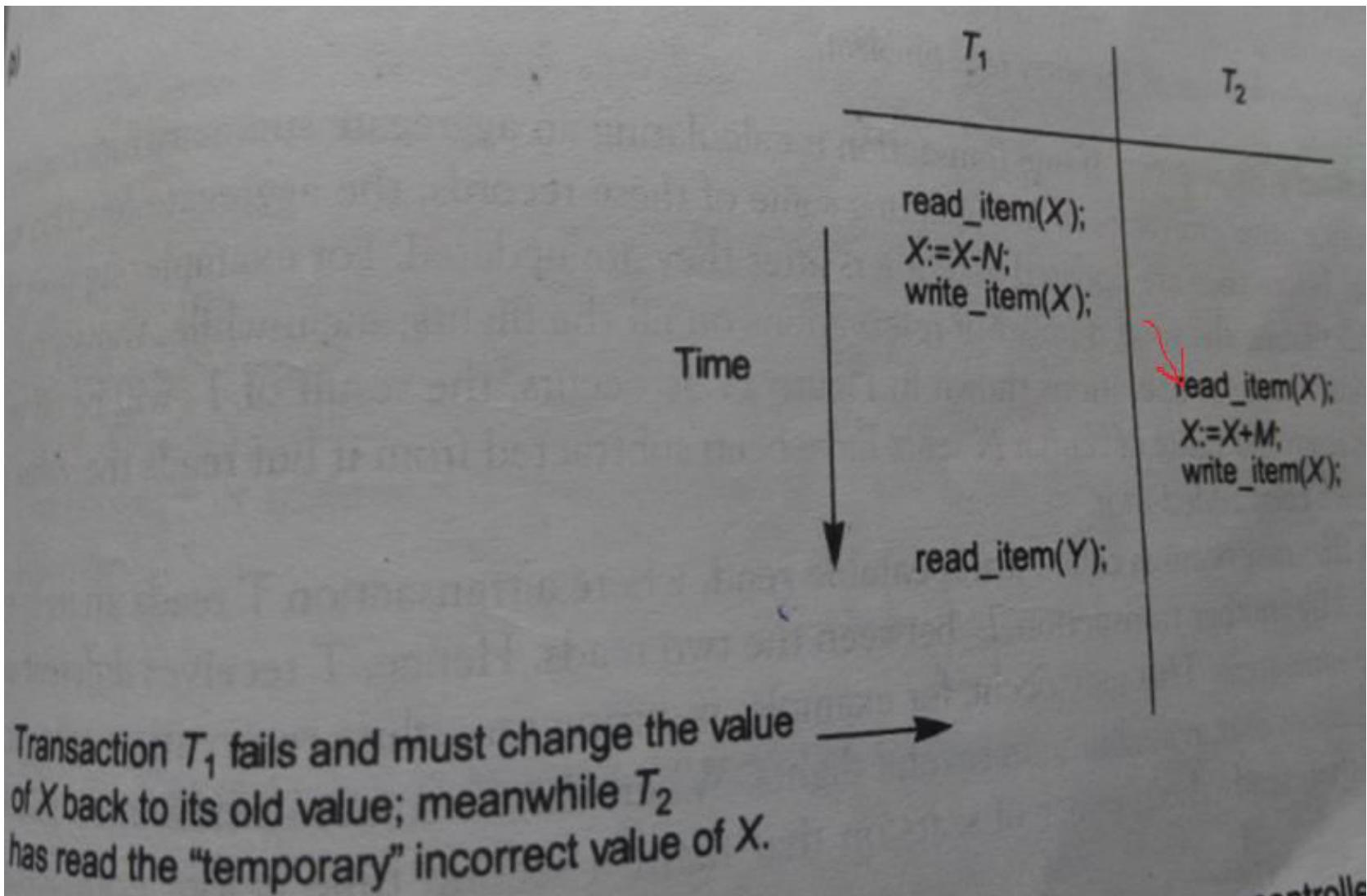


## The Lost Update Problem



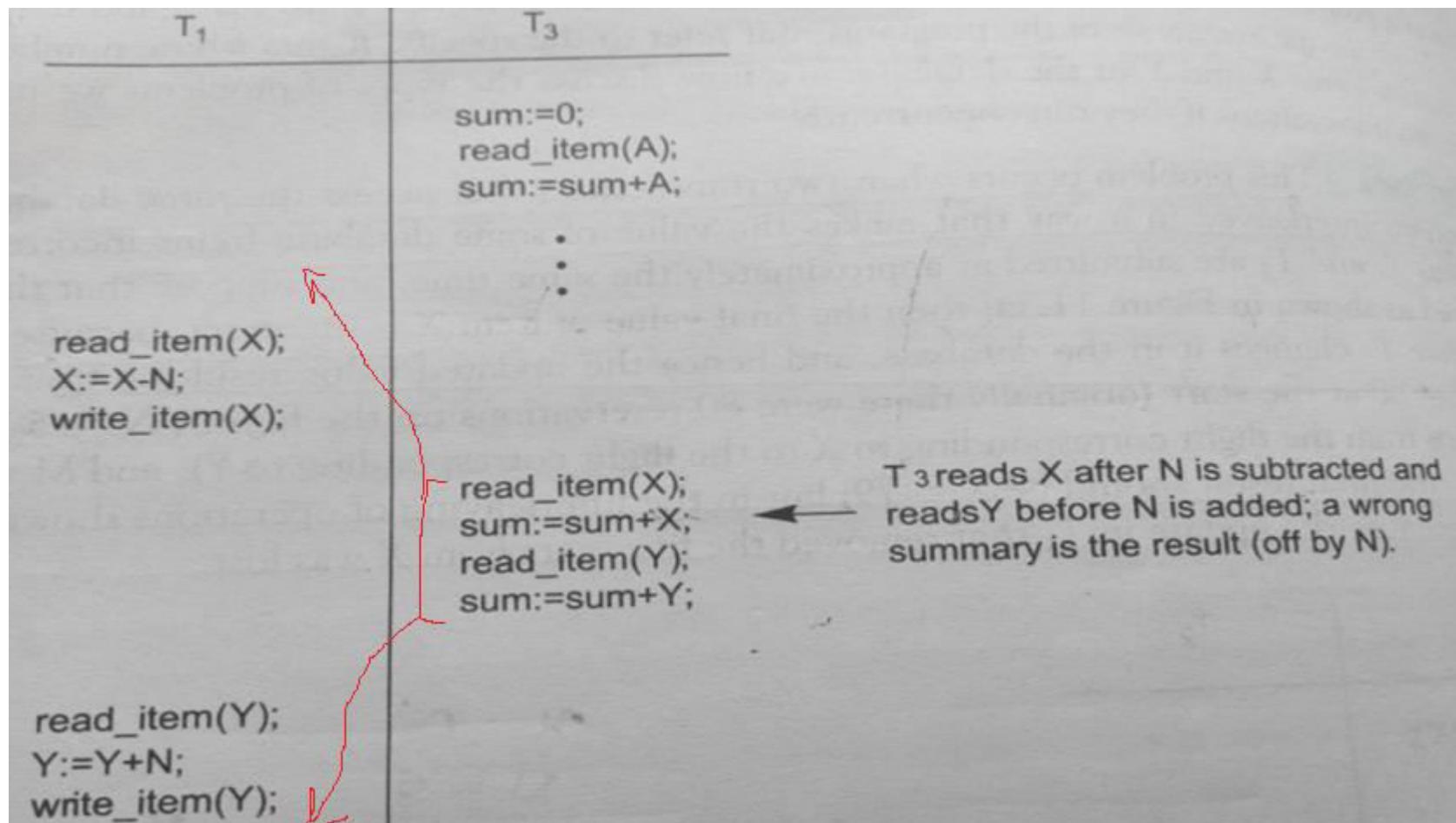


## The temporary Update problem or Dirty Read problem





## The incorrect Summary Problem



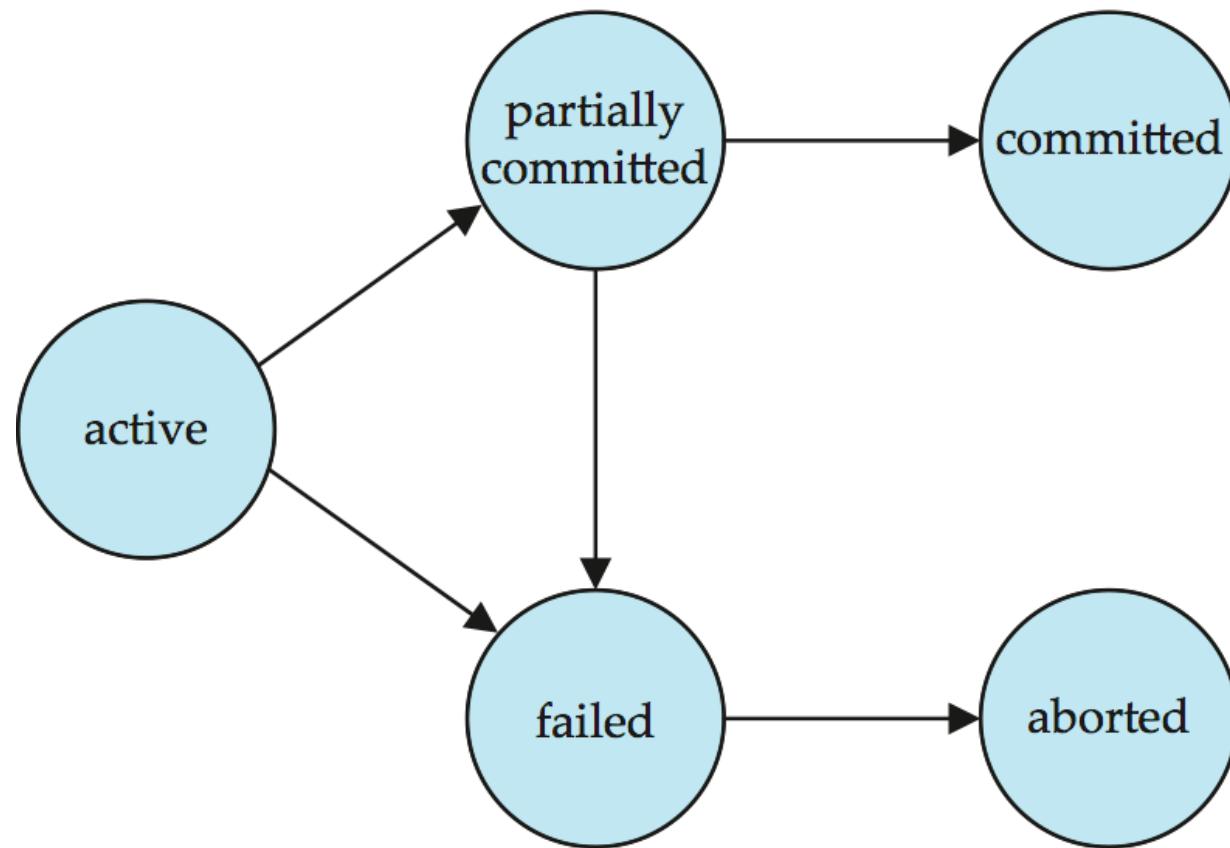


# Storage Structure

- **Volatile storage:** Information residing in volatile storage does not usually survive system crashes. Examples: **main memory** and **cache memory**.
- **Nonvolatile storage:** Information residing in nonvolatile storage survives system crashes. Ex: **secondary storage** devices such as magnetic disk and flash storage, used for online storage, and tertiary storage devices such as optical media, and magnetic tapes, used for archival storage.
- **Stable storage:** Information residing in stable storage is *never lost*



# Transaction State





# Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.  
Two options after it has been aborted:
  - restart the transaction
    - ▶ can be done only if no internal logical error
  - kill the transaction
- **Committed** – after successful completion.



# Transaction Concept

## COMMIT and ROLLBACK commands

- A transaction begins with the first executable SQL statement after a commit, rollback or connection made to the database.
- A transaction can be closed by using a commit or a rollback statement.
- **COMMIT** ends the current transaction and makes permanent changes made during the transaction
- **ROLLBACK** ends the transaction but undoes any changes made during the transaction
- Syntax: COMMIT; ROLLBACK ;



## Example for usage of **Commit, Rollback and SAVEPOINT**

- Withdraw an amount 2,000 and deposit 10,000 for all the accounts. If the sum of balance of all accounts exceeds 2,00,000 then **undo the deposit just made**.

```
Declare
 Total_bal numeric;
Begin
 Update account set balance=balance-2000;
 Savepoint deposit;
 Update account set balance=balance+10000;
 Select sum(balance) into total_bal from account;
 If total_bal > 200000 then
 Rollback to savepoint deposit;
 End if;
 Commit;
End;
```

- SAVEPOINT** marks and saves the current point in the processing of a transaction. When a savepoint is used with a **ROLLBACK** statement, parts of transaction can be undone.



# Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement



# Schedule 1

- Let  $T_1$  transfer \$50 from  $A$  to  $B$ , and  $T_2$  transfer 10% of the balance from  $A$  to  $B$ .
- A **serial schedule** in which  $T_1$  is followed by  $T_2$ :

| $T_1$                                                                                                      | $T_2$                                                                                                                               |
|------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| read ( $A$ )<br>$A := A - 50$<br>write ( $A$ )<br>read ( $B$ )<br>$B := B + 50$<br>write ( $B$ )<br>commit | read ( $A$ )<br>$temp := A * 0.1$<br>$A := A - temp$<br>write ( $A$ )<br>read ( $B$ )<br>$B := B + temp$<br>write ( $B$ )<br>commit |



# Schedule 2

- A serial schedule where  $T_2$  is followed by  $T_1$

| $T_1$ | $T_2$                                                                                                                               |
|-------|-------------------------------------------------------------------------------------------------------------------------------------|
|       | read ( $A$ )<br>$temp := A * 0.1$<br>$A := A - temp$<br>write ( $A$ )<br>read ( $B$ )<br>$B := B + temp$<br>write ( $B$ )<br>commit |



# Schedule 3

- Let  $T_1$  and  $T_2$  be the transactions defined previously. The following schedule is not a serial schedule (It is CONCURRENT Schedule),
- but it is equivalent to Schedule 1.

| $T_1$                                                    | $T_2$                                                                 |
|----------------------------------------------------------|-----------------------------------------------------------------------|
| read ( $A$ )<br>$A := A - 50$<br>write ( $A$ )           | read ( $A$ )<br>$temp := A * 0.1$<br>$A := A - temp$<br>write ( $A$ ) |
| read ( $B$ )<br>$B := B + 50$<br>write ( $B$ )<br>commit | read ( $B$ )<br>$B := B + temp$<br>write ( $B$ )<br>commit            |

In Schedules 1, 2 and 3, the sum  $A + B$  is preserved.



# Schedule 4

- The following concurrent schedule does not preserve the value of  $(A + B)$ . It has 2 Lost Update problems.

| $T_1$                                                                     | $T_2$                                                                                 |
|---------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| read ( $A$ )<br>$A := A - 50$                                             | read ( $A$ )<br>$temp := A * 0.1$<br>$A := A - temp$<br>write ( $A$ )<br>read ( $B$ ) |
| write ( $A$ )<br>read ( $B$ )<br>$B := B + 50$<br>write ( $B$ )<br>commit | $B := B + temp$<br>write ( $B$ )<br>commit                                            |



# Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A concurrent schedule is **serializable** if it is **equivalent to a serial schedule**.
  1. **conflict serializability**
  2. **view serializability**



# *Simplified view of transactions*

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.



# Conflict Serializability

- If a schedule  $S$  can be transformed into a schedule  $S'$  by a series of swaps of **non-conflicting instructions**, we say that  $S$  and  $S'$  are **conflict equivalent**.
- We say that a schedule  $S$  is **conflict Serializable** if it is **conflict equivalent to a serial schedule**



# Conflict Serializability

## Conflicting Instructions

- Instructions  $I_i$  and  $I_j$  of transactions  $T_i$  and  $T_j$  respectively, **conflict** if and only if there exists some item  $Q$  accessed by both  $I_i$  and  $I_j$ , and at least one of these instructions wrote  $Q$ .
  1.  $I_i = \text{read}(Q)$ ,  $I_j = \text{read}(Q)$ .  $I_i$  and  $I_j$  don't conflict.
  2.  $I_i = \text{read}(Q)$ ,  $I_j = \text{write}(Q)$ . They conflict.
  3.  $I_i = \text{write}(Q)$ ,  $I_j = \text{read}(Q)$ . They conflict
  4.  $I_i = \text{write}(Q)$ ,  $I_j = \text{write}(Q)$ . They conflict
- Intuitively, a conflict between  $I_i$  and  $I_j$  forces a (logical) **temporal order between them**.
  - If  $I_i$  and  $I_j$  are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.



# Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where  $T_2$  follows  $T_1$ , by series of swaps of non-conflicting instructions.
- Therefore Schedule 3 is conflict serializable.

| $T_1$                         | $T_2$                         |
|-------------------------------|-------------------------------|
| read ( $A$ )<br>write ( $A$ ) | read ( $A$ )<br>write ( $A$ ) |
| read ( $B$ )<br>write ( $B$ ) | read ( $B$ )<br>write ( $B$ ) |

Schedule 3

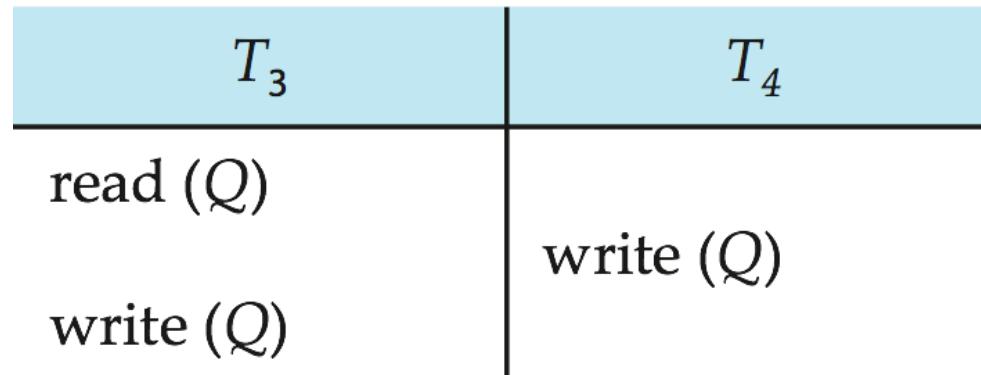
| $T_1$                                                          | $T_2$                                                          |
|----------------------------------------------------------------|----------------------------------------------------------------|
| read ( $A$ )<br>write ( $A$ )<br>read ( $B$ )<br>write ( $B$ ) | read ( $A$ )<br>write ( $A$ )<br>read ( $B$ )<br>write ( $B$ ) |

Schedule 6



# Conflict Serializability (Cont.)

- Example of a schedule that is **not conflict serializable**:



- We are **unable to swap instructions** in the above schedule to obtain either the serial schedule  $\langle T_3, T_4 \rangle$ , or the serial schedule  $\langle T_4, T_3 \rangle$ .

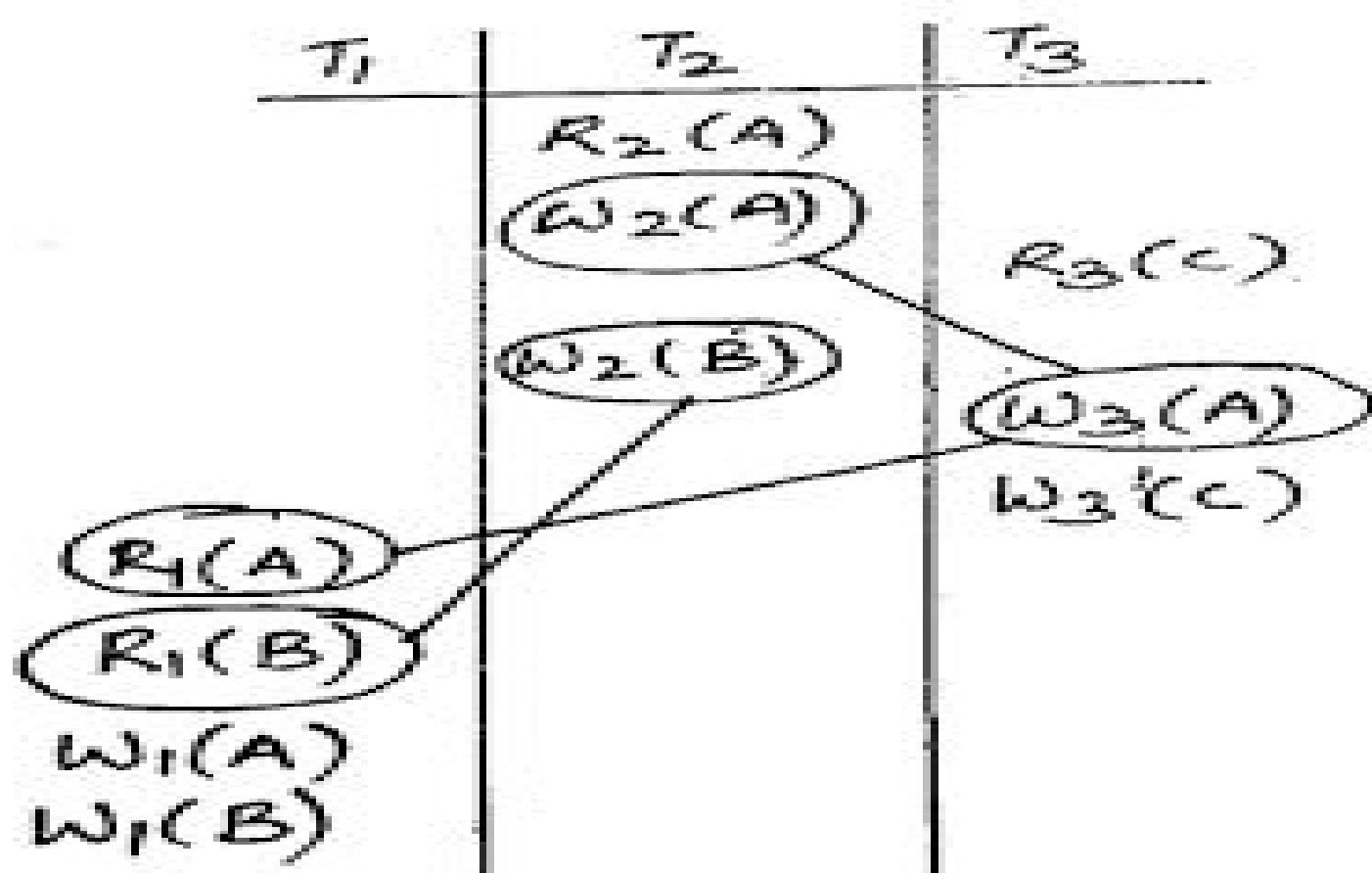
- Eg.

|     |      |      |
|-----|------|------|
| T1: | R(A) | W(A) |
| T2: | W(A) |      |
| T3: |      | W(A) |

|     |            |      |
|-----|------------|------|
| T1: | R(A), W(A) |      |
| T2: |            | W(A) |
| T3: |            | W(A) |



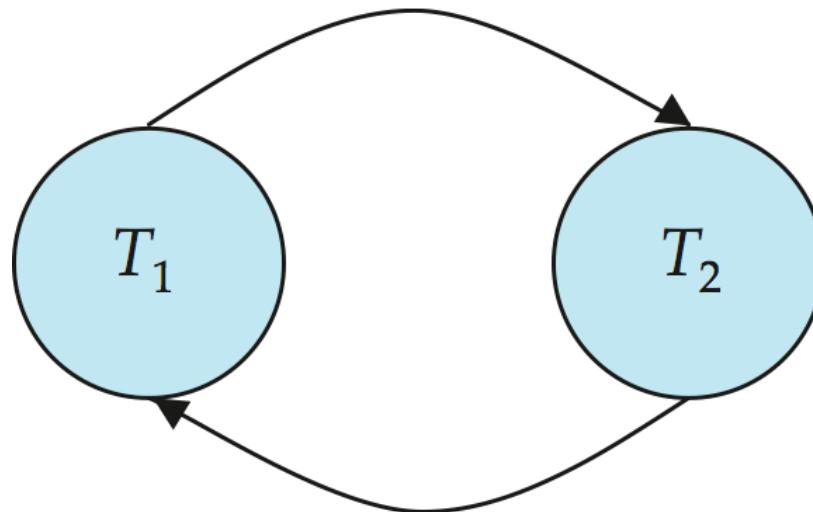
# Check Conflict Serializable, using Precedence Graph





# Testing for Serializability

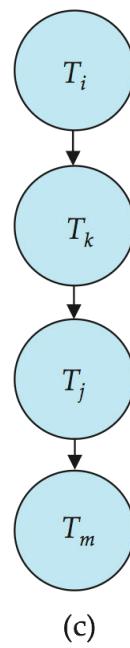
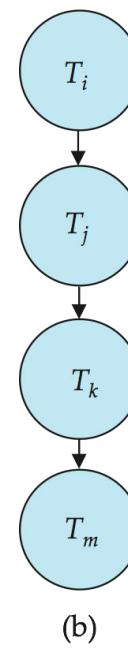
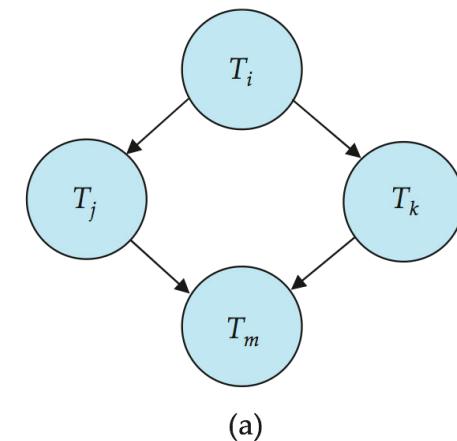
- Consider some schedule of a set of transactions  $T_1, T_2, \dots, T_n$
- **Precedence graph** — a directed graph where the vertices are the transactions (names).
- We draw an arc from  $T_i$  to  $T_j$  if the two transaction conflict, and  $T_i$  accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example 1**





# Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order  $n^2$  time, where  $n$  is the number of vertices in the graph.
  - (Better algorithms take order  $n + e$  where  $e$  is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be  
 $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$ 
    - ▶ Are there others?





T1

Read ( X )

T2

Read ( Y )

T3

Read ( Y )

Write ( Y )

Write ( X )

Write ( X )

Read ( X )

Write ( X )



| $T_1$         | $T_2$    | $T_3$          |
|---------------|----------|----------------|
| $\omega_1(A)$ |          |                |
|               | $R_2(A)$ |                |
|               |          | $\omega_3(A)$  |
|               |          |                |
|               |          | $\omega_1'(A)$ |
|               |          |                |
|               |          | $R_2(A)$       |
|               |          |                |
|               |          | $\omega_3(A)$  |
|               |          |                |
|               |          | $\omega_2(A)$  |
|               |          |                |
|               |          | $T_3$          |
| $S_1$         |          |                |
|               |          | $S_2$          |



# *View Serializability*

- ◆ Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

|          |      |
|----------|------|
| T1: R(A) | W(A) |
|----------|------|

|     |      |
|-----|------|
| T2: | W(A) |
|-----|------|

|     |      |
|-----|------|
| T3: | W(A) |
|-----|------|

S1

|                |
|----------------|
| T1: R(A), W(A) |
|----------------|

|     |      |
|-----|------|
| T2: | W(A) |
|-----|------|

|     |      |
|-----|------|
| T3: | W(A) |
|-----|------|

S2



# View Serializability

|          |      |
|----------|------|
| T1: R(A) | W(A) |
| T2:      | W(A) |
| T3:      | W(A) |

|                |      |
|----------------|------|
| T1: R(A), W(A) |      |
| T2:            | W(A) |
| T3:            | W(A) |



# View Serializability (Cont.)

- A schedule  $S$  is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

| $T_{27}$      | $T_{28}$      | $T_{29}$      |
|---------------|---------------|---------------|
| read ( $Q$ )  |               |               |
| write ( $Q$ ) | write ( $Q$ ) | write ( $Q$ ) |

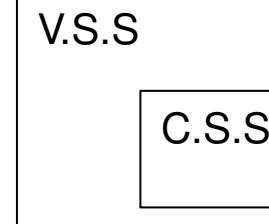
- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes**.



Example:

| T1   | T2   |
|------|------|
| R(A) |      |
|      | R(A) |
|      | W(A) |
|      | R(B) |
| W(A) |      |
| R(B) |      |
| W(B) |      |

Set of all Schedules



Check for View serializable with  $\langle T1, T2 \rangle$  and  $\langle T2, T1 \rangle$



- Example: Check for View equivalence

**T1**

R(A)

W(A)

**T2**

R(A)

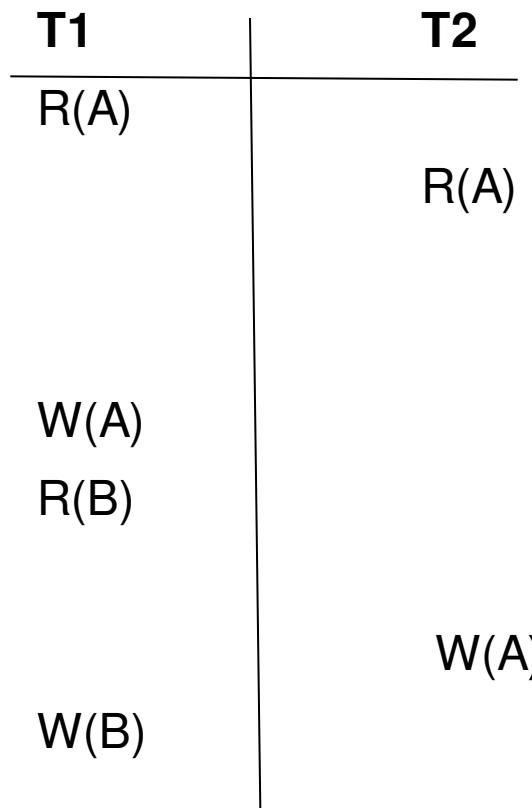
W(A)

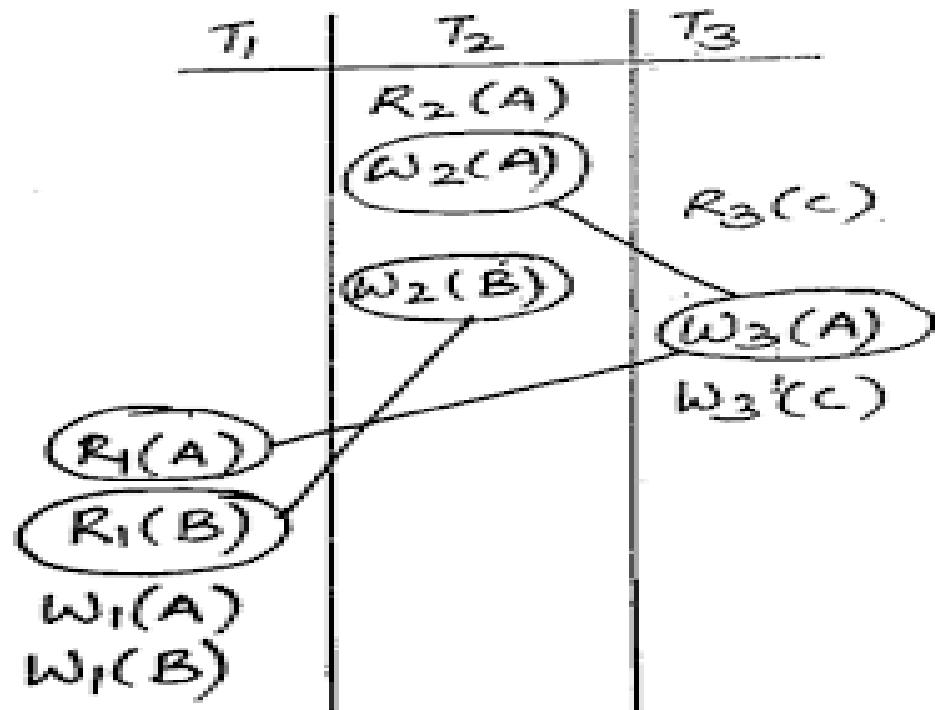
R(B)

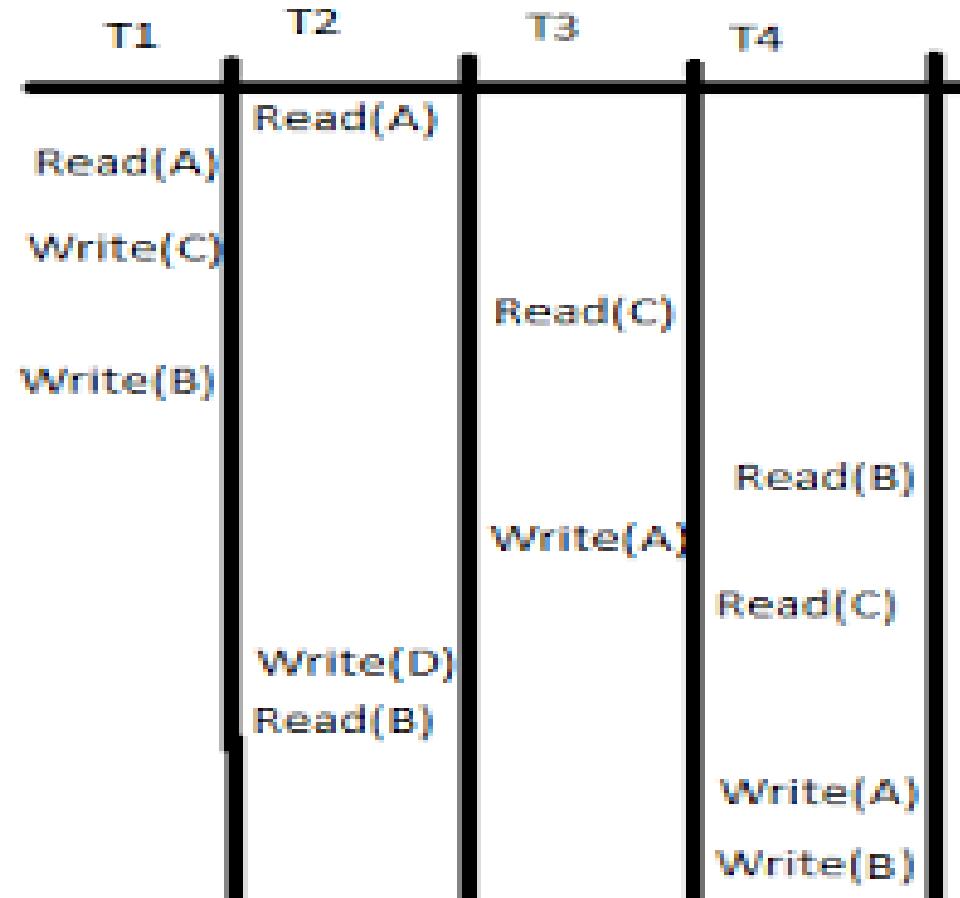
W(B)

R(B)

W(B)









# Other Notions of Serializability

- The schedule below produces same outcome as the serial schedule  $\langle T_1, T_5 \rangle$ , yet is not conflict equivalent or view equivalent to it.

| $T_1$                                          | $T_5$                                          |
|------------------------------------------------|------------------------------------------------|
| read ( $A$ )<br>$A := A - 50$<br>write ( $A$ ) | read ( $B$ )<br>$B := B - 10$<br>write ( $B$ ) |
| read ( $B$ )<br>$B := B + 50$<br>write ( $B$ ) | read ( $A$ )<br>$A := A + 10$<br>write ( $A$ ) |

- Determining such equivalence requires analysis of operations other than read and write.



# Recoverable Schedules

| $T_8$                         | $T_9$                                   |
|-------------------------------|-----------------------------------------|
| read ( $A$ )<br>write ( $A$ ) | read ( $A$ )<br>write ( $A$ )<br>commit |
| read ( $B$ )<br>abort         |                                         |

Schedule is not recoverable



# Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction  $T_j$  reads a data item previously written by a transaction  $T_i$ , then the commit operation of  $T_i$  appears before the commit operation of  $T_j$ .
- The following schedule (Schedule 11) is not recoverable if  $T_9$  commits immediately after the read

| $T_8$                         | $T_9$                                  |
|-------------------------------|----------------------------------------|
| read ( $A$ )<br>write ( $A$ ) |                                        |
|                               | read ( $A$ )<br>commit<br>read ( $B$ ) |

- If  $T_8$  should abort,  $T_9$  would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.



# Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

| $T_{10}$      | $T_{11}$      | $T_{12}$     |
|---------------|---------------|--------------|
| read ( $A$ )  |               |              |
| read ( $B$ )  |               |              |
| write ( $A$ ) | read ( $A$ )  |              |
|               | write ( $A$ ) |              |
| abort         |               | read ( $A$ ) |

- If  $T_{10}$  fails,  $T_{11}$  and  $T_{12}$  must also be rolled back.
- Can lead to the **undoing of a significant amount of work**



# Cascadeless Schedules

- **Cascade-less schedules** — cascading rollbacks cannot occur; for each pair of transactions  $T_i$  and  $T_j$  such that  $T_j$  reads a data item previously written by  $T_i$ , the commit operation of  $T_i$  appears before the read operation of  $T_j$ .
- Every cascade-less schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascade-less



# Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.



# Concurrency Control (Cont.)

- Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- Concurrency-control schemes **tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.**
- Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.



# Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids nonserializable schedules.
  - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.



# Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g. a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g. database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance



# Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.
  
- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
  - E.g. Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)



# Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - ▶ E.g. in JDBC, `connection.setAutoCommit(false);`



Das Bild kann zurzeit nicht angezeigt werden.

# End of Chapter 14

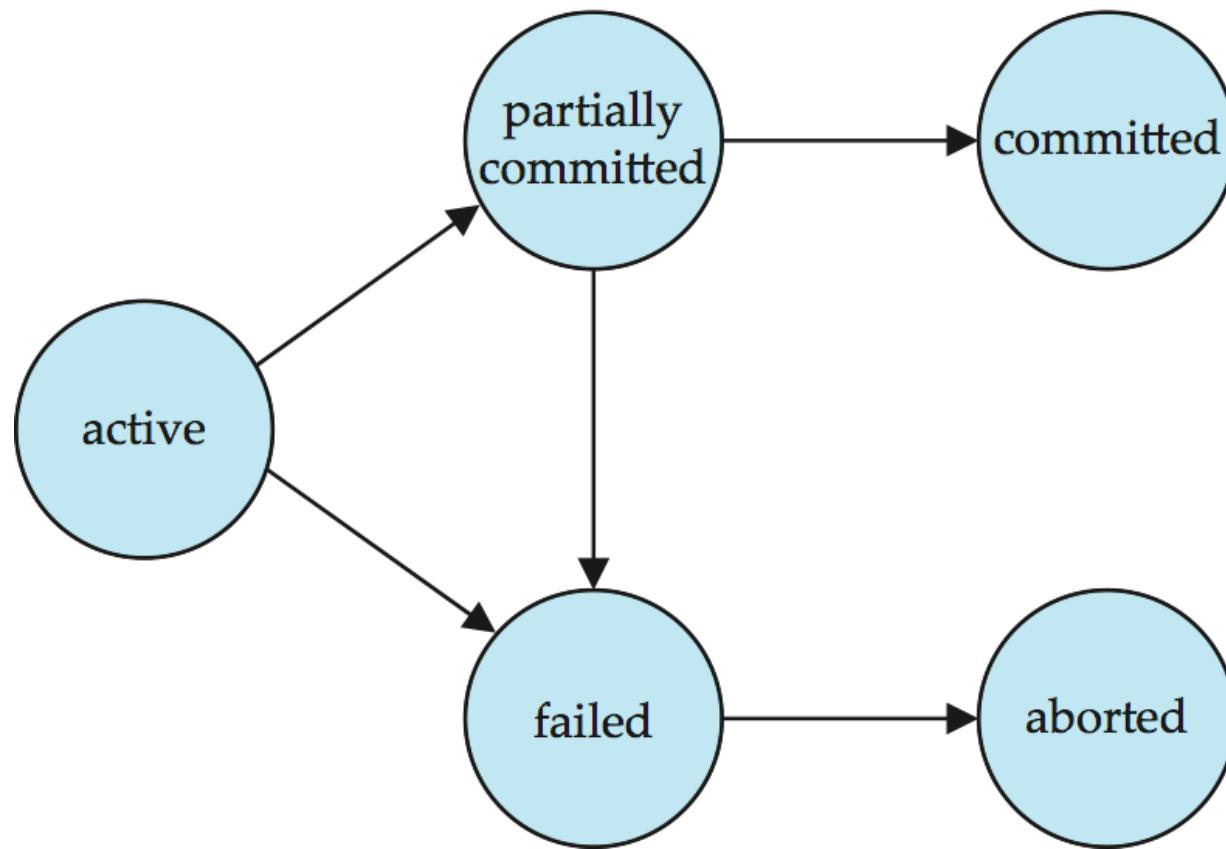
**Database System Concepts, 6<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan

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



# Figure 14.01





# Figure 14.02

| $T_1$                                                                                                      | $T_2$                                                                                                                               |
|------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| read ( $A$ )<br>$A := A - 50$<br>write ( $A$ )<br>read ( $B$ )<br>$B := B + 50$<br>write ( $B$ )<br>commit | read ( $A$ )<br>$temp := A * 0.1$<br>$A := A - temp$<br>write ( $A$ )<br>read ( $B$ )<br>$B := B + temp$<br>write ( $B$ )<br>commit |



# Figure 14.03

| $T_1$                                                                                                                                                                                                                                                                                                                                                                                                    | $T_2$ |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| <p>read (<math>A</math>)<br/><math>temp := A * 0.1</math><br/><math>A := A - temp</math><br/>write (<math>A</math>)<br/>read (<math>B</math>)<br/><math>B := B + temp</math><br/>write (<math>B</math>)<br/>commit</p> <p>read (<math>A</math>)<br/><math>A := A - 50</math><br/>write (<math>A</math>)<br/>read (<math>B</math>)<br/><math>B := B + 50</math><br/>write (<math>B</math>)<br/>commit</p> |       |



# Figure 14.04

| $T_1$                                                    | $T_2$                                                                 |
|----------------------------------------------------------|-----------------------------------------------------------------------|
| read ( $A$ )<br>$A := A - 50$<br>write ( $A$ )           | read ( $A$ )<br>$temp := A * 0.1$<br>$A := A - temp$<br>write ( $A$ ) |
| read ( $B$ )<br>$B := B + 50$<br>write ( $B$ )<br>commit | read ( $B$ )<br>$B := B + temp$<br>write ( $B$ )<br>commit            |



# Figure 14.05

| $T_1$                                                                     | $T_2$                                                                                 |
|---------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| read ( $A$ )<br>$A := A - 50$                                             | read ( $A$ )<br>$temp := A * 0.1$<br>$A := A - temp$<br>write ( $A$ )<br>read ( $B$ ) |
| write ( $A$ )<br>read ( $B$ )<br>$B := B + 50$<br>write ( $B$ )<br>commit | $B := B + temp$<br>write ( $B$ )<br>commit                                            |



# Figure 14.06

| $T_1$                         | $T_2$                         |
|-------------------------------|-------------------------------|
| read ( $A$ )<br>write ( $A$ ) | read ( $A$ )<br>write ( $A$ ) |
| read ( $B$ )<br>write ( $B$ ) | read ( $B$ )<br>write ( $B$ ) |



# Figure 14.07

| $T_1$         | $T_2$         |
|---------------|---------------|
| read ( $A$ )  |               |
| write ( $A$ ) |               |
| read ( $B$ )  | read ( $A$ )  |
| write ( $B$ ) | write ( $A$ ) |
|               | read ( $B$ )  |
|               | write ( $B$ ) |

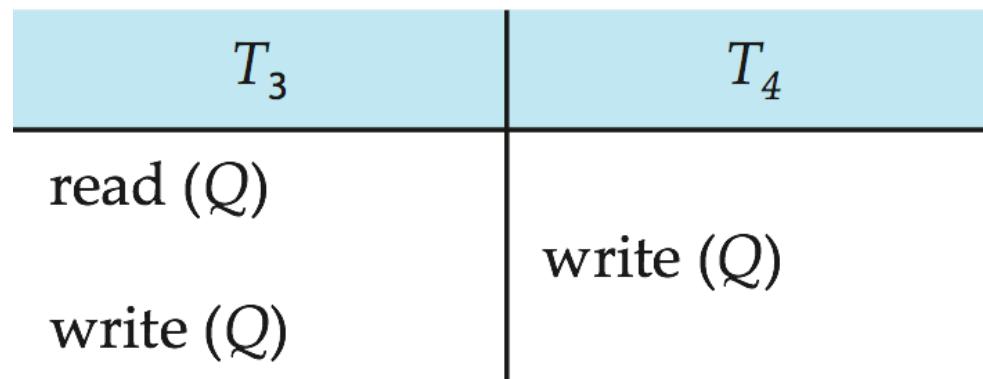


# Figure 14.08

| $T_1$                                                          | $T_2$                                                          |
|----------------------------------------------------------------|----------------------------------------------------------------|
| read ( $A$ )<br>write ( $A$ )<br>read ( $B$ )<br>write ( $B$ ) | read ( $A$ )<br>write ( $A$ )<br>read ( $B$ )<br>write ( $B$ ) |

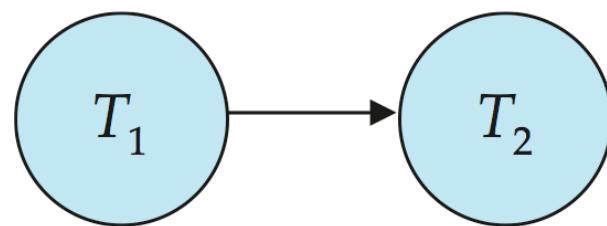


# Figure 14.09

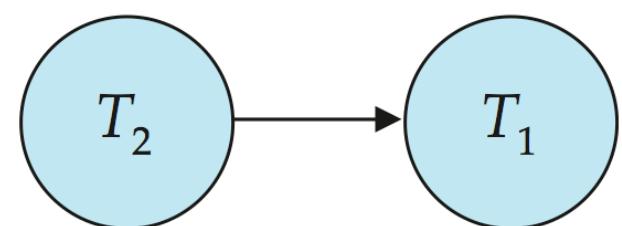




# Figure 14.10



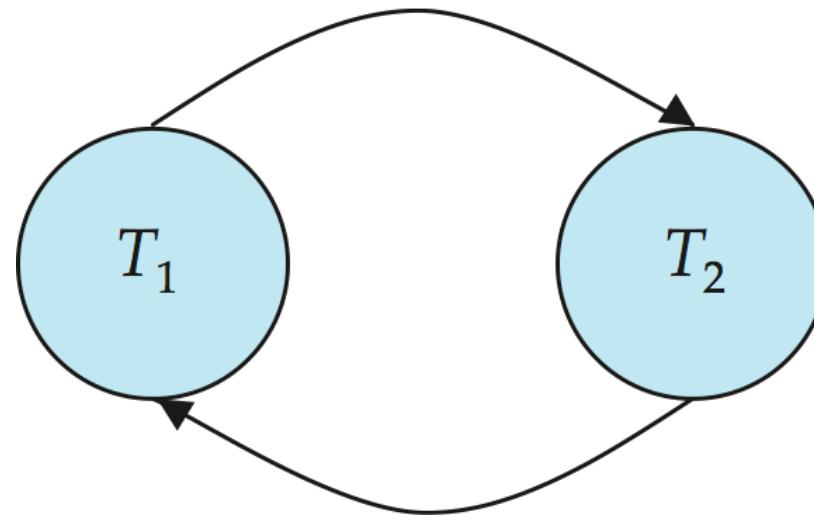
(a)



(b)

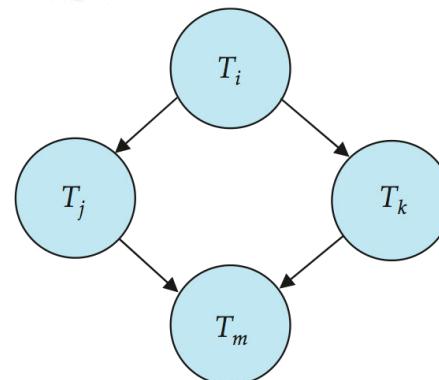


# Figure 14.11

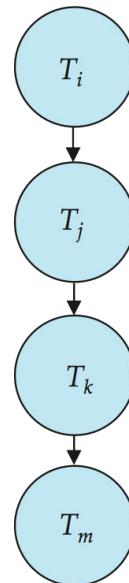




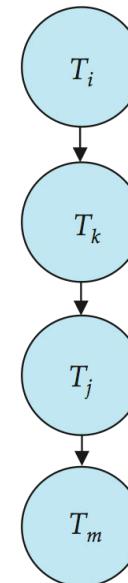
# Figure 14.12



(a)



(b)



(c)



# Figure 14.13

| $T_1$                                          | $T_5$                                          |
|------------------------------------------------|------------------------------------------------|
| read ( $A$ )<br>$A := A - 50$<br>write ( $A$ ) | read ( $B$ )<br>$B := B - 10$<br>write ( $B$ ) |
| read ( $B$ )<br>$B := B + 50$<br>write ( $B$ ) |                                                |
|                                                | read ( $A$ )<br>$A := A + 10$<br>write ( $A$ ) |



# Figure 14.14

| $T_8$                         | $T_9$                  |
|-------------------------------|------------------------|
| read ( $A$ )<br>write ( $A$ ) |                        |
| read ( $B$ )                  | read ( $A$ )<br>commit |

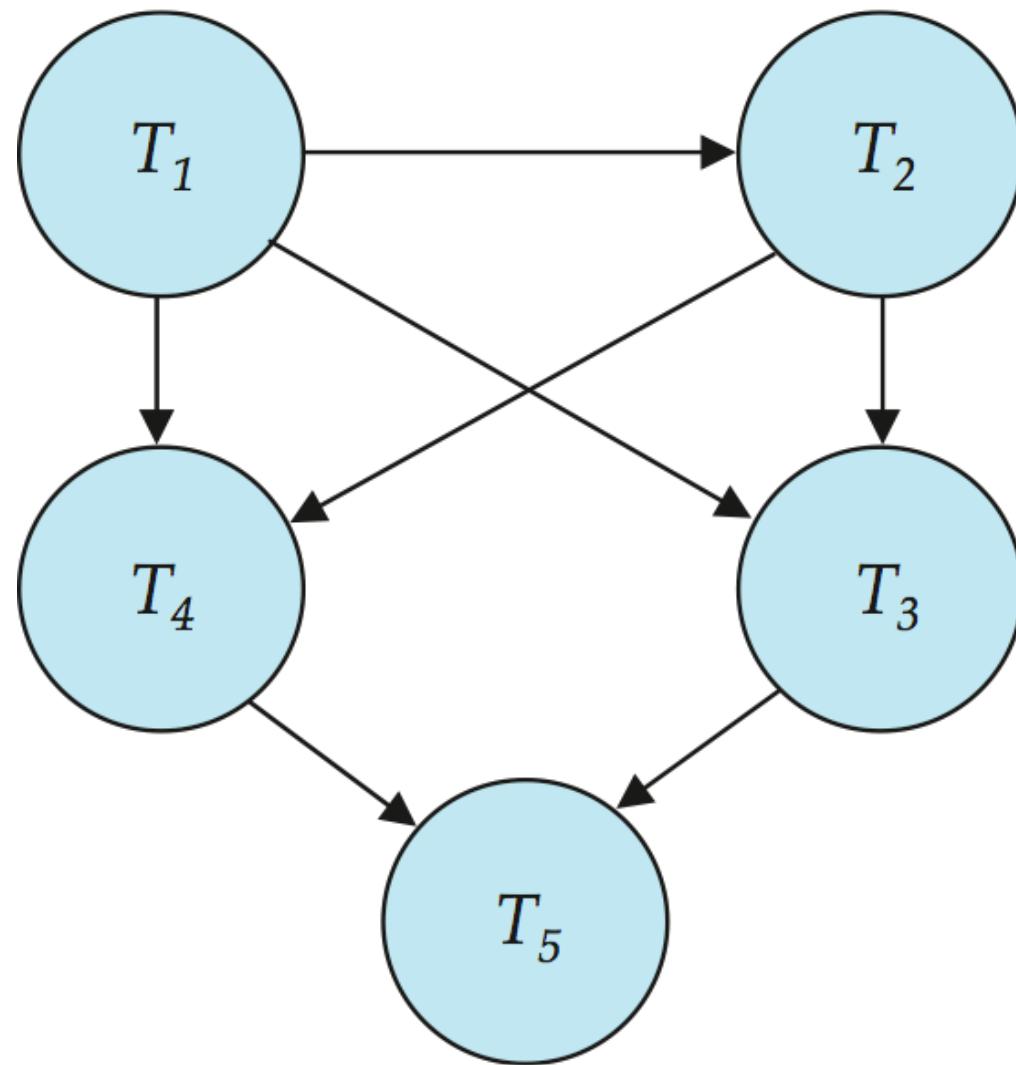


# Figure 14.15

| $T_{10}$      | $T_{11}$      | $T_{12}$     |
|---------------|---------------|--------------|
| read ( $A$ )  |               |              |
| read ( $B$ )  |               |              |
| write ( $A$ ) | read ( $A$ )  |              |
| abort         | write ( $A$ ) | read ( $A$ ) |



# Figure 14.16





Das Bild kann zurzeit nicht angezeigt werden.

# Chapter 15 : Concurrency Control

Database System Concepts, 6<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

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



# Chapter 15: Concurrency Control

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiple Granularity
- Multiversion Schemes
- Insert and Delete Operations
- Concurrency in Index Structures



# Concurrent Schedules

- Advantage of Concurrent Schedules
- Problem associated with the execution of concurrent schedules
  - Lost Update
  - Dirty Read
  - Incorrect Summary



# Concurrency Control Protocols

Goal: Generate Concurrent Schedules

Should satisfy following facts:

- Schedule is **serializable** : Conflict or View serializable
- Recoverable
- Cascade-less
- Increase the level of **concurrency** (To improve system performance)
- Reduce the protocol **overhead**



# Lock based protocol

## Lock Mechanism

*exclusive (X) for Write operation*

*shared (S) for Read operation*

*Concurrency control Manager*

**lock-X** instruction

**lock-S** instruction



# Lock-Based Protocols

- A **lock** is a mechanism to control concurrent access to a data item
- It is a variable associated with a data item in the database and describes the status of the item w.r.t. possible operations that can be applied on to item.
- Generally there is **one lock for each item**.
- **Binary Locks**: can have two states or values: **locked and unlocked** (1 and 0)
- Data items can be locked in two modes :
  1. **exclusive (X) mode**. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. **shared (S) mode**. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to **concurrency-control manager**. Transaction can proceed only after request is granted.



# Lock-Based Protocols (Cont.)

## ■ Lock-compatibility matrix

|   | S     | X     |
|---|-------|-------|
| S | true  | false |
| X | false | false |

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
  - but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.



# Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

**T1: lock-S(A);**

**read (A);**

**unlock(A);**

→ Assume T2, modifies Both A and B

**lock-S(B);**

**read (B);**

**unlock(B);**

**display(A+B) (Sum is *Incorrect* )**

*May lead to incorrect Summary Problem:* solution is .... 2 phase locking Protocol

- Locking as above is **not sufficient to guarantee serializability** — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.



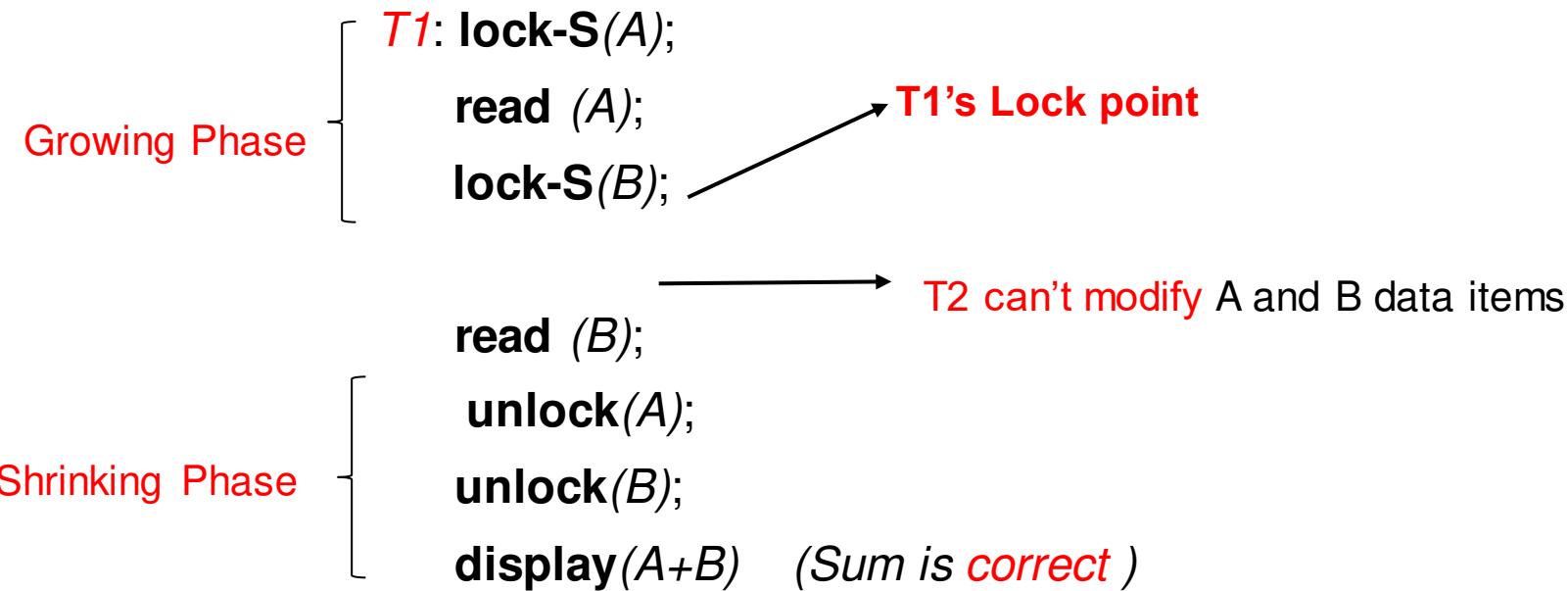
# The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures serializability.
- It can be proved that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock).



# 2P (phase) Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:



*Incorrect summary problem is solved using 2PL protocol where A and B data items are unlocked after accessing them. So once A and B are unlocked, now T2 can access them for modification.*



# Pitfalls of Lock-Based Protocols

- Consider the partial schedule

S1 (Schedule)

|        | $T_3$          | $T_4$          |
|--------|----------------|----------------|
| Time ↓ | lock-x ( $B$ ) |                |
|        | read ( $B$ )   |                |
|        | $B := B - 50$  |                |
|        | write ( $B$ )  |                |
|        | lock-x ( $A$ ) | lock-s ( $A$ ) |
|        |                | read ( $A$ )   |
|        |                | lock-s ( $B$ ) |

- Neither  $T_3$  nor  $T_4$  can make progress — executing **lock-S( $B$ )** causes  $T_4$  to wait for  $T_3$  to release its lock on  $B$ , while executing **lock-X( $A$ )** causes  $T_3$  to wait for  $T_4$  to release its lock on  $A$ .
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of  $T_3$  or  $T_4$  must be rolled back and its locks released.



# Pitfalls of Lock-Based Protocols (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a **sequence of other transactions** request and are granted an S-lock on the same item.
  - The same transaction is **repeatedly rolled back due to deadlocks**.
- Concurrency control manager can be designed to **prevent starvation**.



# Example for Starvation

T1

T2

T3

T4

S(A)

UL(A)  
C (commite);

S(A)

UL(A)  
C

S(A)

UL(A)  
C; // T1 resumes Here

Time ↓

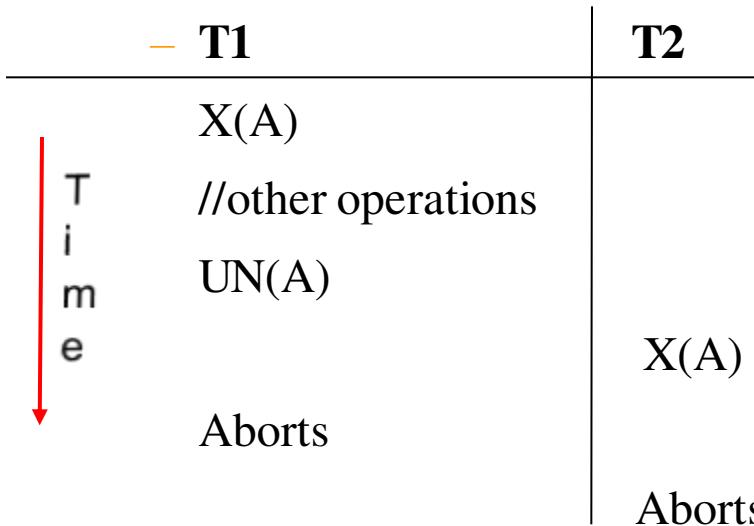
X(A) // T1 Waiting

S(A): Shared Lock on A  
X(A): Exclusive Lock on A



# The Two-Phase Locking Protocol (Cont.)

- Cascading roll-back is possible under two-phase locking.

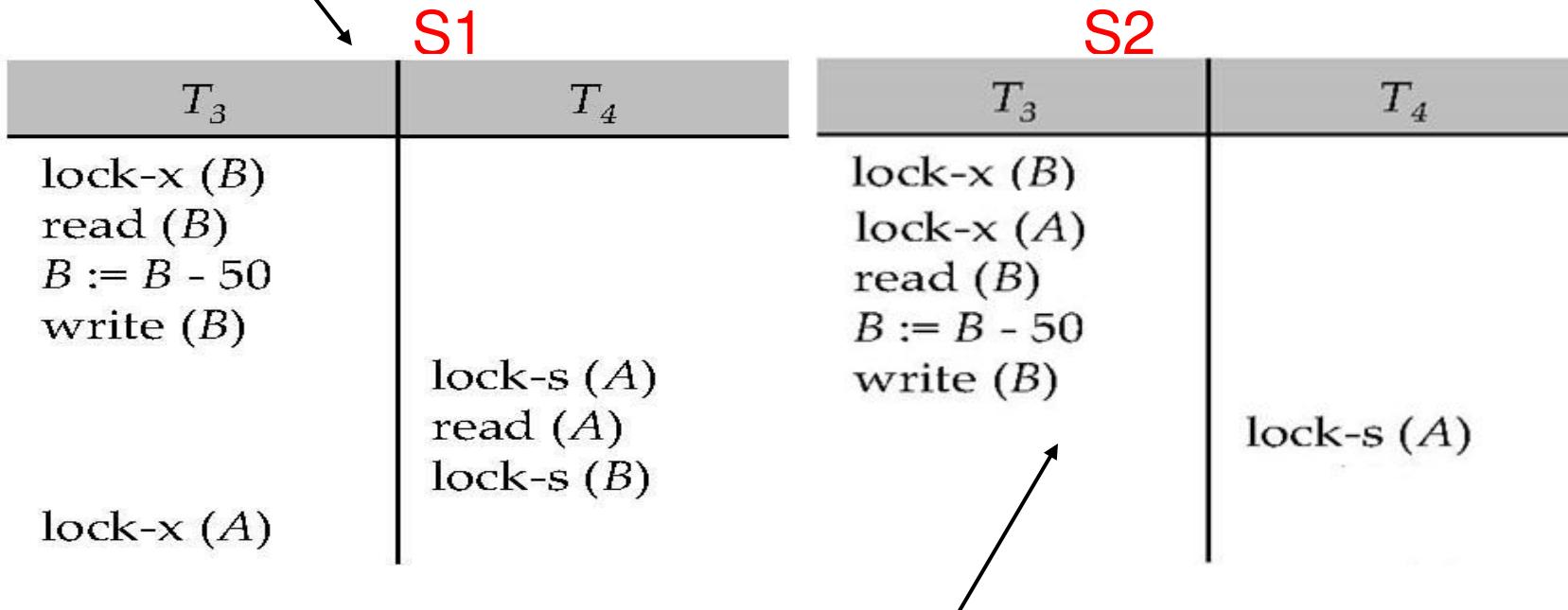


- Solution: strict two-phase locking Protocol:
  - Here a transaction must **hold all its exclusive locks till it commits/aborts**. After commit it must unlock.
- Rigorous two-phase locking Protocol is even stricter: here **all locks** are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit. **Concurrency level of the schedule is reduced**.



# Static / Conservative 2PL protocol

- Deadlock:



- Solution: Static/ Conservative 2PL protocol
  - Gets Lock in **Atomic** Manner
- No Deadlock
- Reduces the concurrency level of the schedule



# Lock Conversions

To improve the concurrency level

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-R on item
    - can acquire a lock-W on item
    - can convert a lock-R to a lock-W (upgrade)
  - Second Phase:
    - can release a lock-R
    - can release a lock-W
    - can convert a lock-W to a lock-R (downgrade)



# Lock Conversions

$T_3$

lock-x ( $B$ )

lock-x ( $A$ )

read ( $A$ )

write ( $A$ )

write ( $B$ )

read ( $B$ )

$T_3$

lock-s ( $A$ )

read ( $A$ )

(upgrade)

lock-x ( $A$ )

write ( $A$ )

lock-x ( $B$ )

write ( $B$ )

lock-s ( $B$ )

(downgrade)

read ( $B$ )



# Automatic Acquisition of Locks

- A transaction  $T_i$  issues the standard read/write instruction, without explicit locking calls.
- The operation **read( $D$ )** is processed as:

```
if T_i has a lock on D
 then
 read(D)
 else begin
 if necessary wait until no other
 transaction has a lock-W on D
 grant T_i a lock-R on D ;
 read(D)
 end
```



# Automatic Acquisition of Locks (Cont.)

- **write( $D$ )** is processed as:

**if**  $T_i$  has a **lock-W** on  $D$

**then**

**write( $D$ )**

**else begin**

            if necessary wait until no other trans. has any lock on  $D$ ,

**if**  $T_i$  has a **lock-R** on  $D$

**then**

**upgrade** lock on  $D$  to **lock-W**

**else**

                    grant  $T_i$  a **lock-W** on  $D$

**write( $D$ )**

**end;**

- All locks are released after commit or abort



# Deadlock Handling

## ■ Schedule with deadlock

| $T_1$                                                              | $T_2$                                                          |
|--------------------------------------------------------------------|----------------------------------------------------------------|
| <b>lock-X on A</b><br>write (A)<br><br>wait for <b>lock-X on B</b> | <b>lock-X on B</b><br>write (B)<br>wait for <b>lock-X on A</b> |



# Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- *Deadlock prevention* protocols ensure that the **system will never enter into a deadlock state.**
- Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration or **conservative** two phase locking).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol)



# More Deadlock Prevention Strategies

- Following schemes use **transaction timestamps** for the sake of deadlock prevention alone.
- **wait-die** scheme — non-preemptive
  - If  $TS(T_i) < TS(T_j)$  then  $T_i$  is allowed to wait otherwise abort  $T_i$  and restart it later **with the same timestamp**.
  - Older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item
- **wound-wait** scheme — preemptive
  - If  $TS(T_i) < TS(T_j)$  then abort  $T_j$  and restart it later with the same timestamp otherwise  $T_i$  is allowed to wait
  - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.



# Deadlock prevention (Cont.)

- Both in *wait-die* and in *wound-wait* schemes, a **rolled back transactions is restarted with its original timestamp**. Older transactions thus have precedence over newer ones, and starvation is hence avoided.
- Timeout-Based Schemes:
  - a transaction waits for a **lock only for a specified amount of time**. After that, the wait times out and the transaction is rolled back.
  - thus deadlocks are not possible
  - simple to implement; but **starvation** is possible. Also difficult to determine good value of the timeout interval.

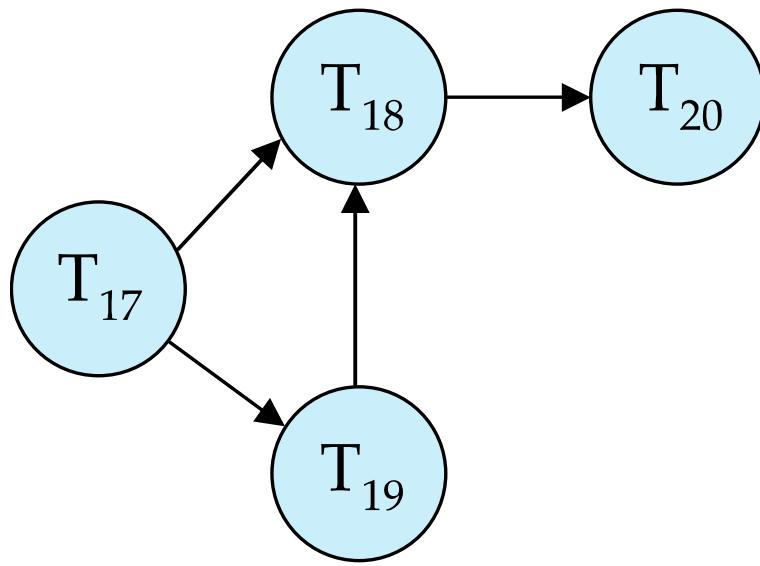


# Deadlock Detection

- Deadlock Detection where we periodically check to see if the system is in a state of deadlock. This can happen if the transaction weight is less and so they repeatedly lock the item.
- Deadlocks can be described as **a *wait-for graph***, which consists of a pair  $G = (V, E)$ ,
  - $V$  is a set of vertices (all the **transactions** in the system)
  - $E$  is a set of edges; each element is an ordered pair  $T_i \rightarrow T_j$ .
- If  $T_i \rightarrow T_j$  is in  $E$ , then there is a directed edge from  $T_i$  to  $T_j$ , implying that  $T_i$  is waiting for  $T_j$  to release a data item.
- When  $T_i$  requests a data item currently being held by  $T_j$ , then the edge  $T_i \rightarrow T_j$  is inserted in the wait-for graph. This edge is removed only when  $T_j$  is no longer holding a data item needed by  $T_i$ .
- The **system is in a deadlock state** if and only if the **wait-for graph has a cycle**. Must invoke a deadlock-detection algorithm periodically to look for cycles.

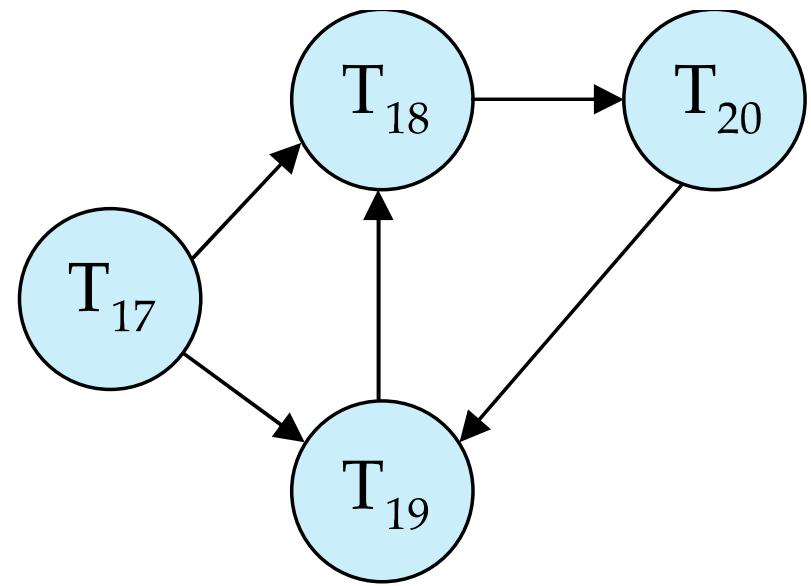


# Deadlock Detection (Cont.)



Schedule – 1

Wait-for graph **without a cycle**



Schedule – 2

Wait-for graph **with a cycle**



# Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as **victim that will incur minimum cost** (completed execution of minimum no. of instructions).
  - Rollback -- determine how far to roll back transaction
    - ▶ **Total rollback:** Abort the transaction and then **restart it**.
    - ▶ More effective to roll back transaction only as far **as necessary to break deadlock**.
  - **Starvation** happens if same transaction is always chosen as victim. Include the **number of rollbacks** in the cost factor **to avoid starvation**



# Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction  $T_i$  has time-stamp  $\text{TS}(T_i)$ , a new transaction  $T_j$  is assigned time-stamp  $\text{TS}(T_j)$  such that  $\text{TS}(T_i) < \text{TS}(T_j)$ .
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data ‘Q’ two timestamp values:
  - **W-timestamp(Q)** is the largest time-stamp (Youngest Transaction) of any transaction that executed **write(Q)** successfully.
  - **R-timestamp(Q)** is the largest time-stamp of any transaction that executed **read(Q)** successfully.



# Timestamp-Based Protocols (Cont.)

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are **executed in timestamp order**.
- Suppose a transaction  $T_i$  issues a **read( $Q$ )**
  1. If  $TS(T_i) < W\text{-timestamp}(Q)$ , then  $T_i$  needs to read a value of  $Q$  that was already overwritten.
    - Hence, the **read** operation is **rejected**, and  $T_i$  is rolled back.
  2. If  $TS(T_i) > W\text{-timestamp}(Q)$ , then the **read** operation is executed, and  $R\text{-timestamp}(Q)$  is set to **max(R-timestamp( $Q$ ), TS( $T_i$ ))**.



# Timestamp-Based Protocols (Cont.)

- Suppose that transaction  $T_i$  issues **write(Q)**.
  1. If  $TS(T_i) < R\text{-timestamp}(Q)$ , then the value of  $Q$  that  $T_i$  is producing was needed previously, and the system assumed that that value would never be produced.
    - Hence, the **write** operation is **rejected**, and  $T_i$  is rolled back.
  2. If  $TS(T_i) < W\text{-timestamp}(Q)$ , then  $T_i$  is attempting to write an obsolete value of  $Q$ .
    - Hence, this **write** operation is **rejected**, and  $T_i$  is rolled back.
  3. Otherwise, the **write** operation is executed, and  $W\text{-timestamp}(Q)$  is set to  $TS(T_i)$ .



# Example Use of the Protocol

A partial schedule for several data items **for transactions with timestamps 1, 2, 3, 4, 5.** Apply Timestamp based protocol.

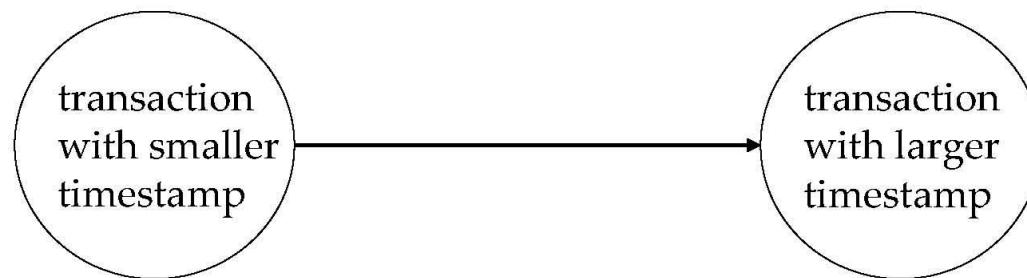
Schedule - 1

| $T_1$        | $T_2$                 | $T_3$                          | $T_4$ | $T_5$                          |
|--------------|-----------------------|--------------------------------|-------|--------------------------------|
|              |                       |                                |       | read ( $X$ )                   |
| read ( $Y$ ) | read ( $Y$ )          |                                |       |                                |
|              |                       | write ( $Y$ )<br>write ( $Z$ ) |       |                                |
|              | read ( $Z$ )<br>abort |                                |       | read ( $Z$ )                   |
| read ( $X$ ) |                       |                                |       |                                |
|              |                       | read ( $W$ )                   |       |                                |
|              |                       | write ( $W$ )<br>abort         |       |                                |
|              |                       |                                |       | write ( $Y$ )<br>write ( $Z$ ) |



# Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol **guarantees serializability** since all the arcs in the **precedence graph** are of the form:



Thus, there will be **no cycles** in the precedence graph

- Timestamp protocol **ensures freedom from deadlock** as no transaction ever waits.
- But the schedule may **not be cascade-free**, and **may not even be recoverable**: Due to absence of Locking mechanism



# Validation-Based Protocol

- Execution of transaction  $T_i$  is done in **three phases**.
  1. **Read and execution phase**: Transaction  $T_i$  writes only to temporary local variables
  2. **Validation phase**: Transaction  $T_i$  performs a ``validation test'' to determine if local variables **can be written without violating serializability**.
  3. **Write phase**: If  $T_i$  is validated, the updates are applied to the database; otherwise,  $T_i$  is rolled back.
- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
  - Assume for simplicity that the validation and write phase occur together, atomically and serially
    - ▶ I.e., only one transaction executes validation/write at a time.
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation



# Validation-Based Protocol (Cont.)

- Each transaction  $T_i$  has 3 timestamps
  - $\text{Start}(T_i)$  : the time when  $T_i$  started its execution
  - $\text{Validation}(T_i)$ : the time when  $T_i$  entered its validation phase
  - $\text{Finish}(T_i)$  : the time when  $T_i$  finished its write phase
- **Serializability order** is determined by timestamp given at validation time, to increase concurrency.
  - Thus  $\text{TS}(T_i)$  is given the value of  $\text{Validation}(T_i)$ .



# Validation Test for Transaction $T_j$

- To handle clash between  $T_i$ ' Write and  $T_j$ ' validation phases:
- Validation Test for Transaction  $T_j$  **successful**

If for all  $T_i$  with  $TS(T_i) < TS(T_j)$  either one of the following condition holds :

1. **finish( $T_i$ ) < start( $T_j$ )** (i.e.  $T_j$  starts after  $T_i$  finishes )
2. **finish( $T_i$ ) < validation( $T_j$ )** and the set of data items written by  $T_i$  **does not intersect** with the set of data items read by  $T_j$ . (i.e  $T_j$ 's Read phase is overlapping with Write phases of the  $T_i$ )

then validation succeeds and  $T_j$  can be committed.

Otherwise, validation fails and  $T_j$  is aborted.

- *Justification:* Either the first condition is satisfied, and there is no **overlapped execution**, or the second condition is satisfied and
  - the writes of  $T_i$  do not affect reads of  $T_j$  since they occur after  $T_i$  has finished its reads.
  - the writes of  $T_i$  do not affect reads of  $T_j$  since  $T_j$  does not read any item written by  $T_i$ .



# Validation protocol

- Cascade-less : Since Validation test make sure that Ti reads only the committed values
- No deadlock : Since make use of Timestamp
- Suffer from Starvation
  - When validation test fails.



# Schedule Produced by Validation

- Example of schedule produced using validation

Schedule – 1

| $T_{25}$                                                          | $T_{26}$                                                       |
|-------------------------------------------------------------------|----------------------------------------------------------------|
| read ( $B$ )                                                      | read ( $B$ )<br>$B := B - 50$<br>read ( $A$ )<br>$A := A + 50$ |
| read ( $A$ )<br>$\langle validate \rangle$<br>display ( $A + B$ ) | $\langle validate \rangle$                                     |

$\langle validate \rangle$  → T26' Validation test fails



Das Bild kann zurzeit nicht angezeigt werden.

# End of Chapter

Thanks to Alan Fekete and Sudhir Jorwekar for Snapshot  
Isolation examples

**Database System Concepts, 6<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan

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



# Figure 15.01

|   | S     | X     |
|---|-------|-------|
| S | true  | false |
| X | false | false |



# Figure 15.04

| $T_1$                                                                                  | $T_2$                                                                                                                               | concurrency-control manager                                                                              |
|----------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|
| lock-x ( $B$ )<br><br>read ( $B$ )<br>$B := B - 50$<br>write ( $B$ )<br>unlock ( $B$ ) | lock-s ( $A$ )<br><br>read ( $A$ )<br>unlock ( $A$ )<br>lock-s ( $B$ )<br><br>read ( $B$ )<br>unlock ( $B$ )<br>display ( $A + B$ ) | grant-x ( $B, T_1$ )<br><br>grant-s ( $A, T_2$ )<br><br>grant-s ( $B, T_2$ )<br><br>grant-x ( $A, T_2$ ) |
| lock-x ( $A$ )<br><br>read ( $A$ )<br>$A := A + 50$<br>write ( $A$ )<br>unlock ( $A$ ) |                                                                                                                                     |                                                                                                          |



# Figure 15.07

| $T_3$                                                                                  | $T_4$                                            |
|----------------------------------------------------------------------------------------|--------------------------------------------------|
| lock-x ( $B$ )<br>read ( $B$ )<br>$B := B - 50$<br>write ( $B$ )<br><br>lock-x ( $A$ ) | lock-s ( $A$ )<br>read ( $A$ )<br>lock-s ( $B$ ) |



# Figure 15.08

| $T_5$                                                                                               | $T_6$                                                             | $T_7$                          |
|-----------------------------------------------------------------------------------------------------|-------------------------------------------------------------------|--------------------------------|
| lock-x ( $A$ )<br>read ( $A$ )<br>lock-s ( $B$ )<br>read ( $B$ )<br>write ( $A$ )<br>unlock ( $A$ ) | lock-x ( $A$ )<br>read ( $A$ )<br>write ( $A$ )<br>unlock ( $A$ ) | lock-s ( $A$ )<br>read ( $A$ ) |
|                                                                                                     |                                                                   |                                |

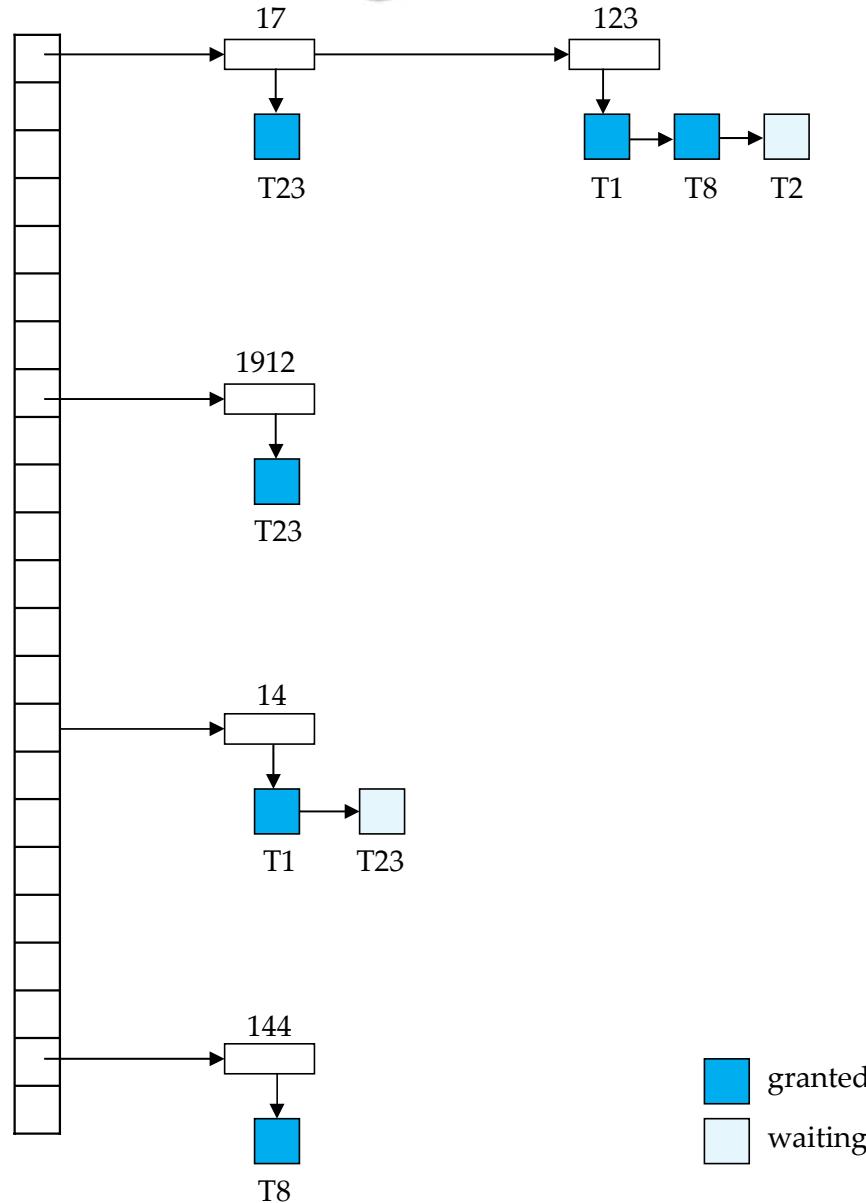


# Figure 15.09

| $T_8$             | $T_9$              |
|-------------------|--------------------|
| lock-s ( $a_1$ )  | lock-s ( $a_1$ )   |
| lock-s ( $a_2$ )  | lock-s ( $a_2$ )   |
| lock-s ( $a_3$ )  | unlock-s ( $a_3$ ) |
| lock-s ( $a_4$ )  | unlock-s ( $a_4$ ) |
| lock-s ( $a_n$ )  |                    |
| upgrade ( $a_1$ ) |                    |

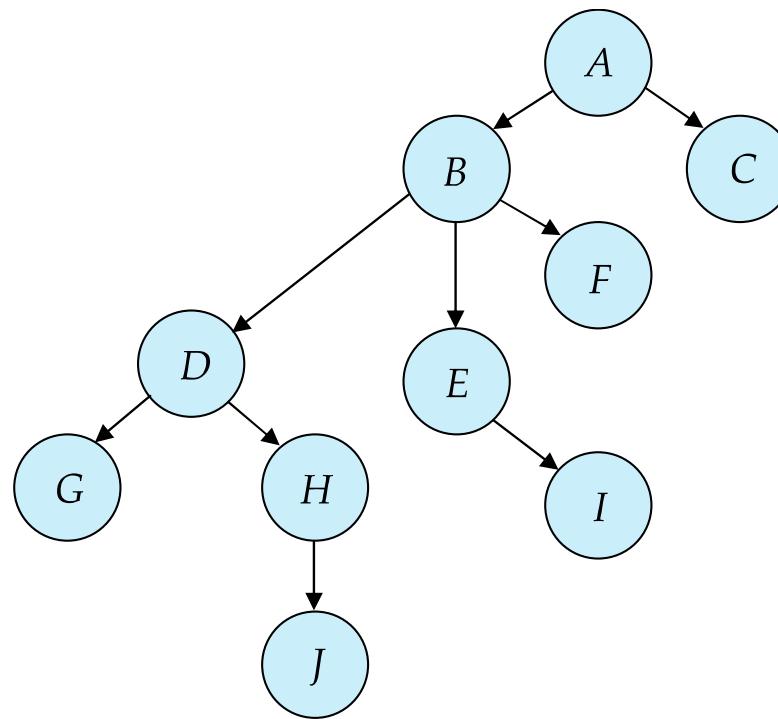


# Figure 15.10





# Figure 15.11



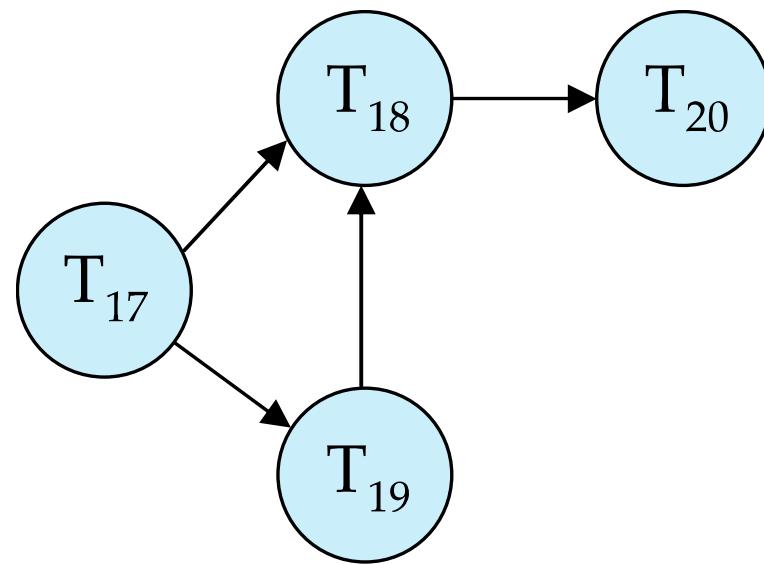


# Figure 15.12

| T <sub>10</sub>                                      | T <sub>11</sub>                        | T <sub>12</sub>          | T <sub>13</sub>                                      |
|------------------------------------------------------|----------------------------------------|--------------------------|------------------------------------------------------|
| lock-x (B)                                           | lock-x (D)<br>lock-x (H)<br>unlock (D) |                          |                                                      |
| lock-x (E)<br>lock-x (D)<br>unlock (B)<br>unlock (E) | unlock (H)                             | lock-x (B)<br>lock-x (E) |                                                      |
| lock-x (G)<br>unlock (D)                             |                                        |                          | lock-x (D)<br>lock-x (H)<br>unlock (D)<br>unlock (H) |
| unlock (G)                                           |                                        | unlock (E)<br>unlock (B) |                                                      |

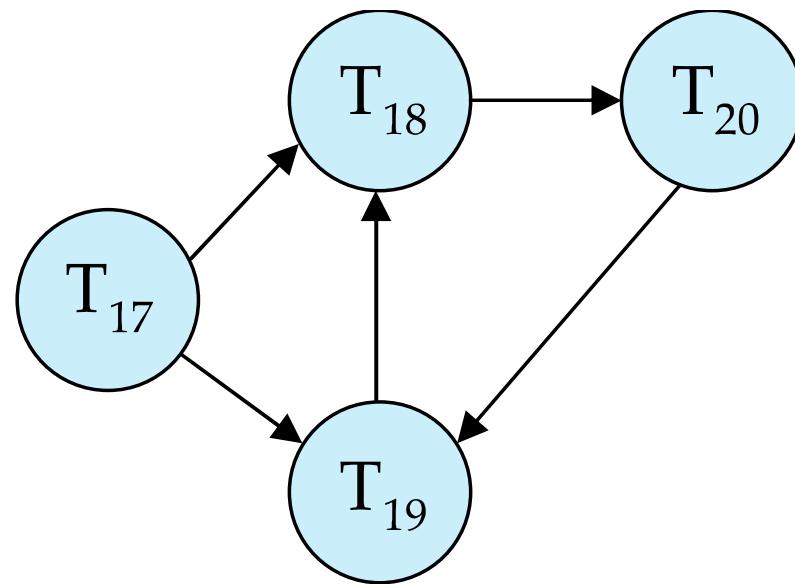


# Figure 15.13



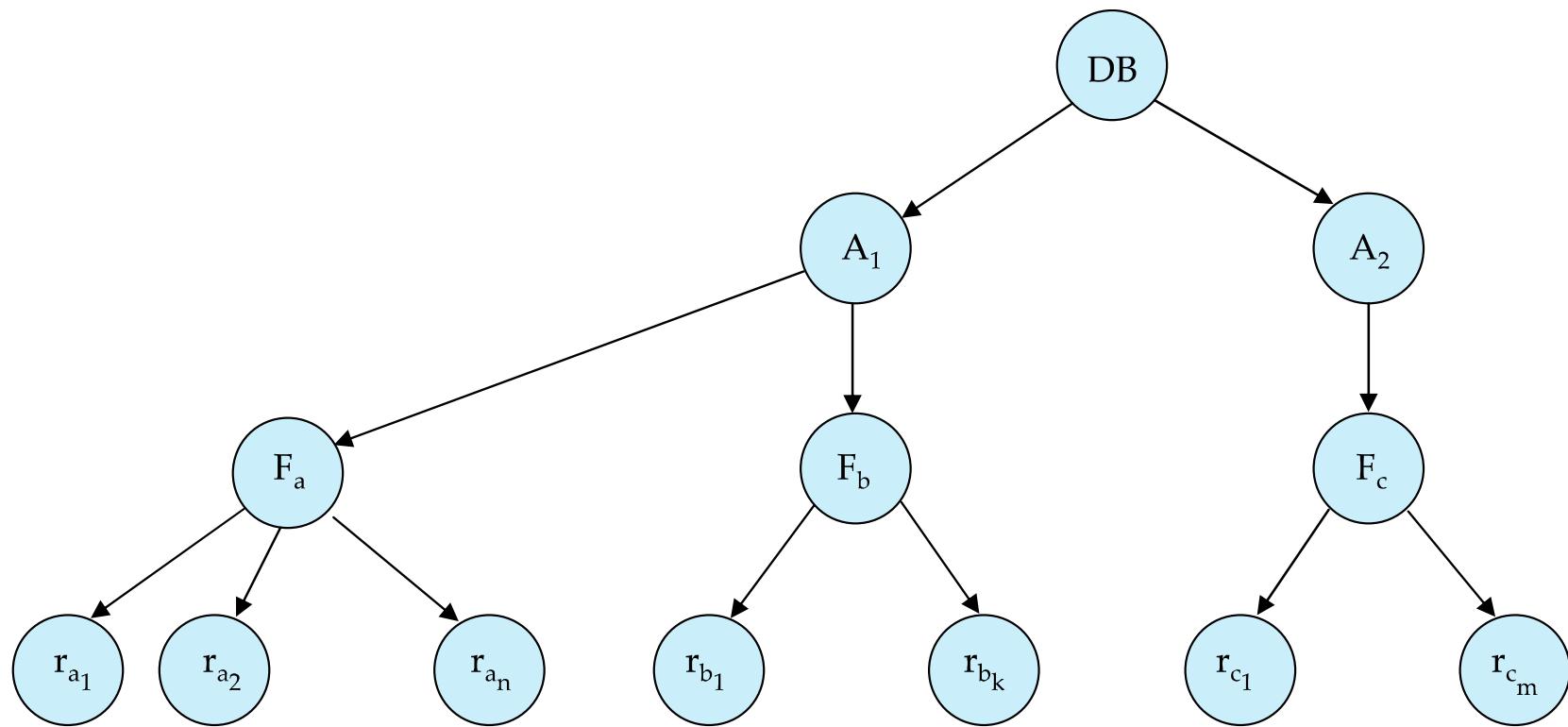


# Figure 15.14





# Figure 15.15





# Figure 15.16

|     | IS    | IX    | S     | SIX   | X     |
|-----|-------|-------|-------|-------|-------|
| IS  | true  | true  | true  | true  | false |
| IX  | true  | true  | false | false | false |
| S   | true  | false | true  | false | false |
| SIX | true  | false | false | false | false |
| X   | false | false | false | false | false |



# Figure 15.17

| $T_{25}$            | $T_{26}$                                              |
|---------------------|-------------------------------------------------------|
| read ( $B$ )        | read ( $B$ )<br>$B := B - 50$<br>write ( $B$ )        |
| read ( $A$ )        | read ( $A$ )                                          |
| display ( $A + B$ ) | $A := A + 50$<br>write ( $A$ )<br>display ( $A + B$ ) |



# Figure 15.18

| $T_{27}$      | $T_{28}$      |
|---------------|---------------|
| read ( $Q$ )  |               |
| write ( $Q$ ) | write ( $Q$ ) |



# Figure 15.19

| $T_{25}$                                                          | $T_{26}$                                                       |
|-------------------------------------------------------------------|----------------------------------------------------------------|
| read ( $B$ )                                                      | read ( $B$ )<br>$B := B - 50$<br>read ( $A$ )<br>$A := A + 50$ |
| read ( $A$ )<br>$\langle validate \rangle$<br>display ( $A + B$ ) | $\langle validate \rangle$<br>write ( $B$ )<br>write ( $A$ )   |

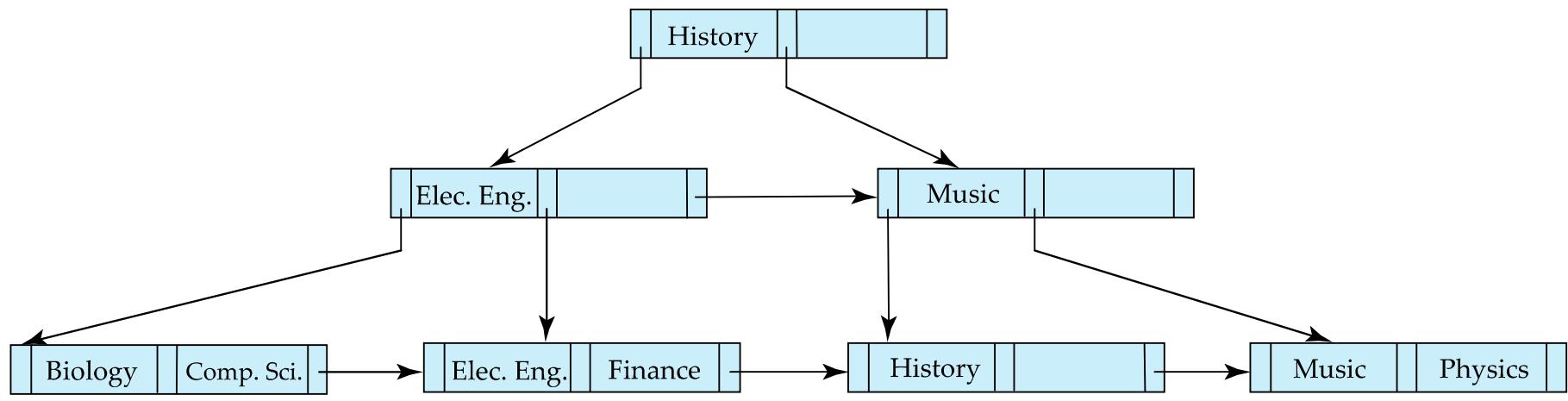


# Figure 15.20

| $T_{32}$                                         | $T_{33}$                                                          |
|--------------------------------------------------|-------------------------------------------------------------------|
| lock-s ( $Q$ )<br>read ( $Q$ )<br>unlock ( $Q$ ) | lock-x ( $Q$ )<br>read ( $Q$ )<br>write ( $Q$ )<br>unlock ( $Q$ ) |
| lock-s ( $Q$ )<br>read ( $Q$ )<br>unlock ( $Q$ ) |                                                                   |

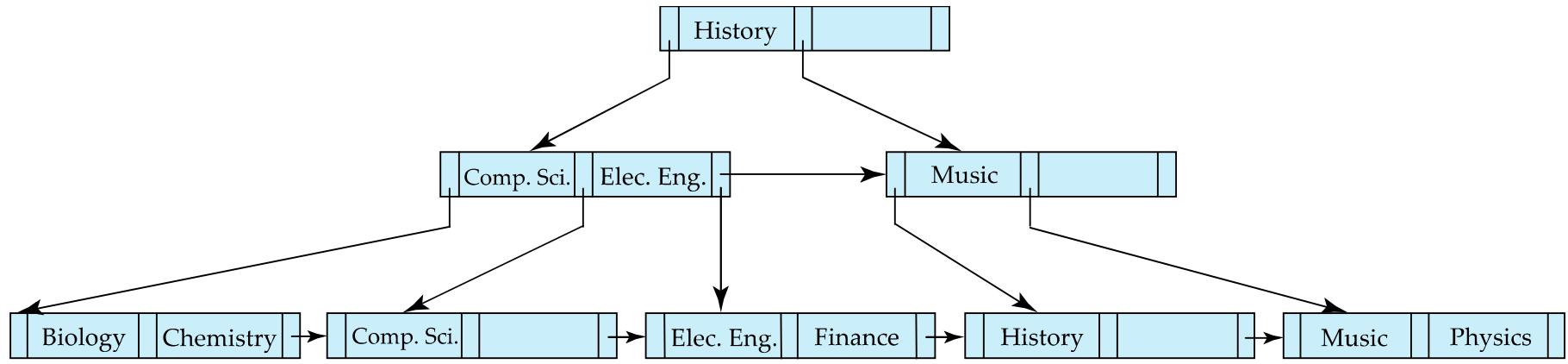


# Figure 15.21





# Figure 15.22





# Figure 15.23

|   | S     | X     | I     |
|---|-------|-------|-------|
| S | true  | false | false |
| X | false | false | false |
| I | false | false | true  |



# Figure in-15.1

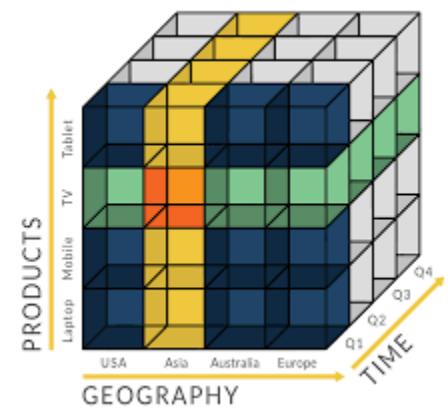
| $T_{27}$      | $T_{28}$      | $T_{29}$      |
|---------------|---------------|---------------|
| read ( $Q$ )  | write ( $Q$ ) |               |
| write ( $Q$ ) |               | write ( $Q$ ) |

# NoSQL and MongoDB

Prepared by referencing the following

<https://docs.mongodb.com/manual/tutorial/>

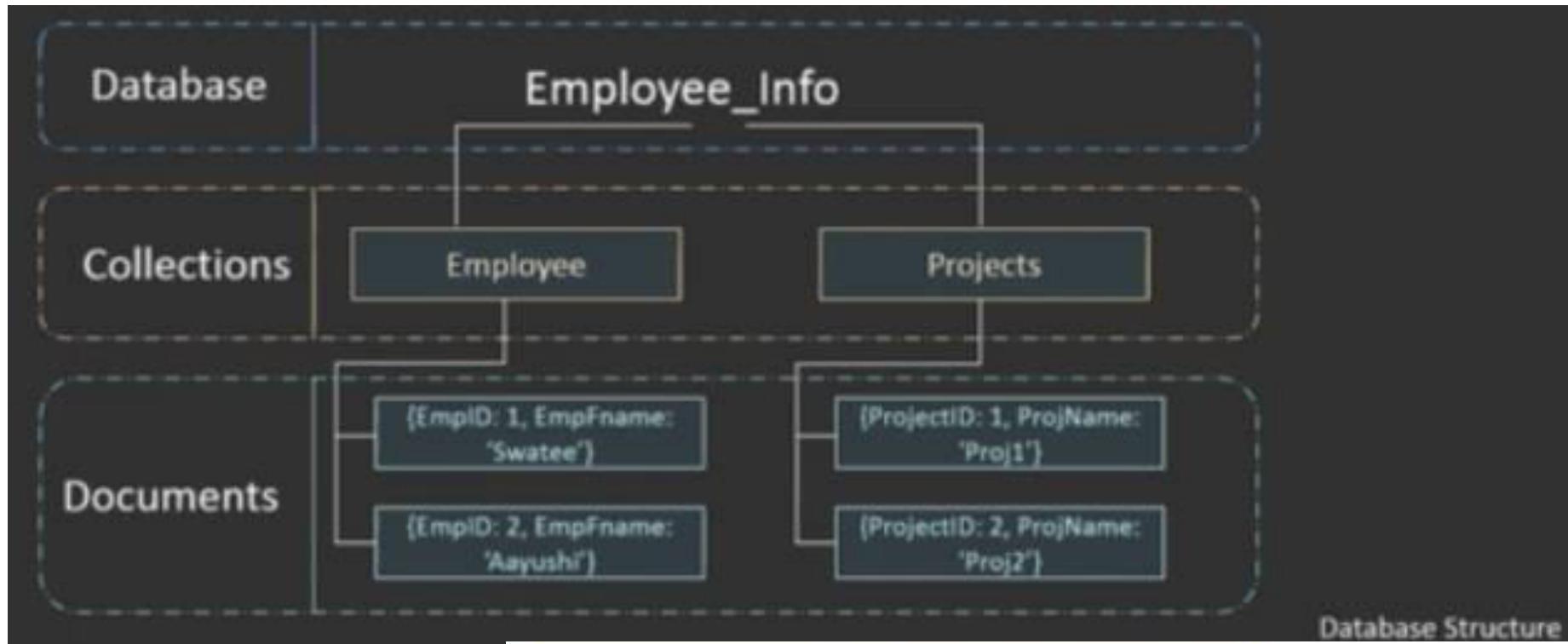
- A database Management System provides the mechanism to store and retrieve the data. There are different kinds of database Management Systems:
  1. RDBMS (Relational Database Management Systems)
  2. NoSQL (Not only SQL)



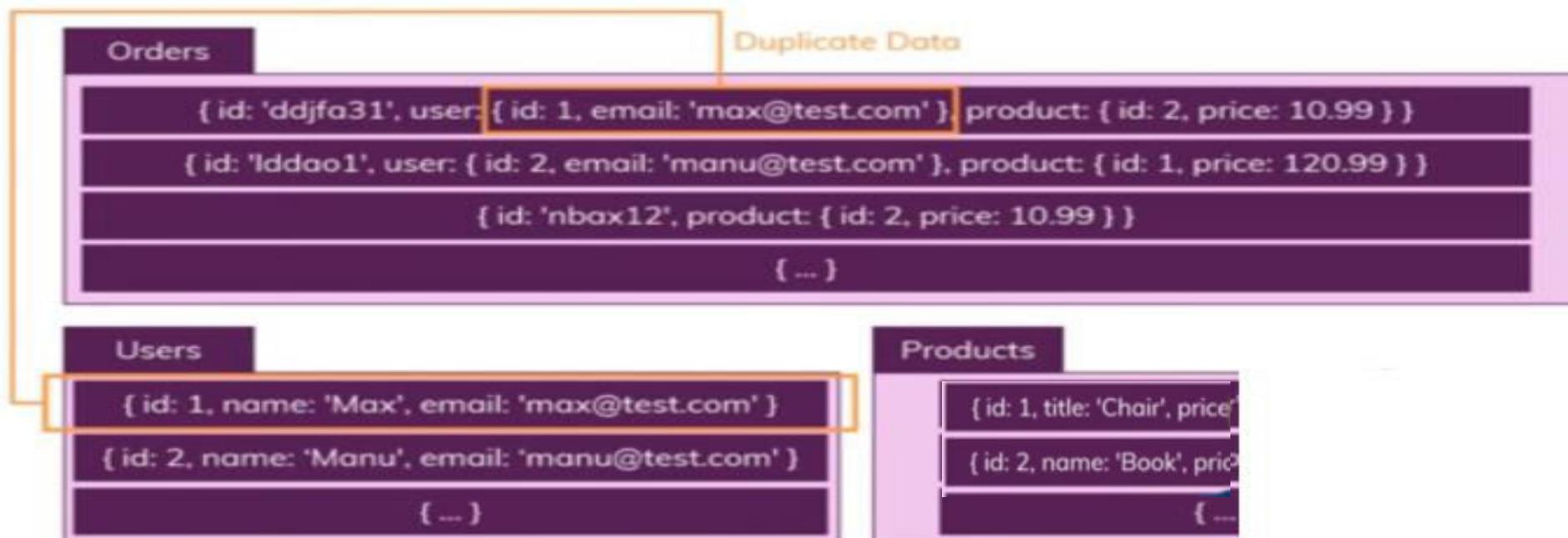
# introduction to NoSQL

- A **NoSQL** originally referring to non SQL or **non relational** is a database that provides a mechanism for **storage and retrieval of data**. This **data is modeled in means other than the tabular relations** used in relational databases.
- NoSQL databases are different than relational databases like MySql. In relational database you need to create the table, define schema, set the data types of fields etc before you can actually insert the data.

In NoSQL you don't have to worry about that, you can insert, update data **without creating schema**.



Database Structure



```
{
 first_name: 'Paul',
 surname: 'Miller',
 cell: 447557505611,
 city: 'London',
 location: [45.123,47.232],
 Profession: ['banking', 'finance', 'trader'],
 cars: [
 { model: 'Bentley',
 year: 1973,
 value: 100000, ... },
 { model: 'Rolls Royce',
 year: 1965,
 value: 330000, ... }
]
}
```

Fields

String

Number

Geo-Coordinates

Typed field values

Fields can contain arrays

Fields can contain an array of sub-documents

# NoSQL

1. **NoSQL** is a non-relational DMS
2. Does not require a fixed schema, avoids joins, and is easy to scale.
3. The purpose of using a NoSQL database is for distributed data stores
4. NoSQL is used for Big data and real-time web apps
5. For example, companies like Twitter, Facebook, Google collect terabytes of user data per day
6. NoSQL database stands for "Not Only SQL" or "Not SQL."
7. Traditional RDBMS uses SQL syntax to store and retrieve data for further insights
8. **NoSQL** can store structured, semi-structured, unstructured and polymorphic data (in one **collection** we have many versions of **document** schema)

# Features of NoSQL

## Non-relational

- NoSQL databases never follow the relational model
- Never provide tables with flat fixed-column records
- Doesn't require object-relational mapping and data normalization
- No complex features like query languages, query planners, referential integrity joins, ACID

## Schema-free

- NoSQL databases are either schema-free or have relaxed schemas
- Do not require any sort of definition of the schema of the data
- Offers heterogeneous structures of data in the same domain

# Types of NoSQL Databases

- Key-value Pair Based **Databases**
- Column-oriented **Databases**
- Graphs based **Databases**
- Document-oriented **Databases**

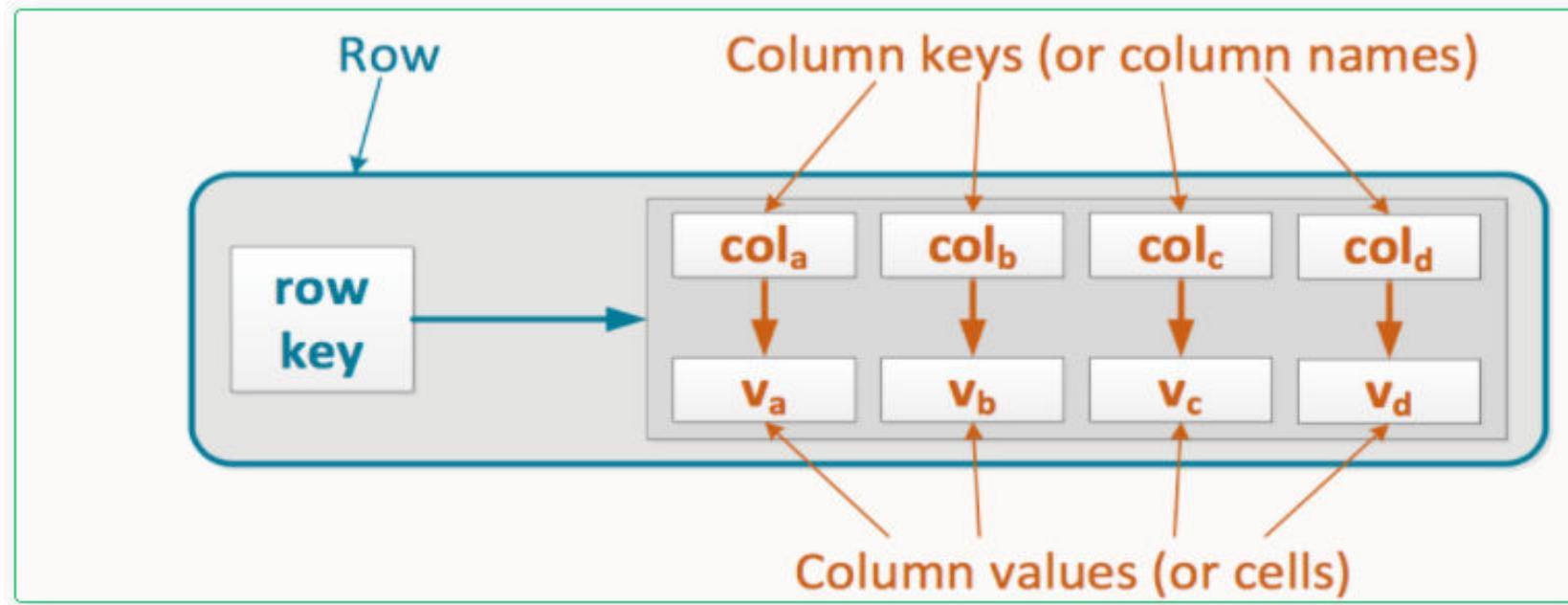
# Key Value Pair Based

| Key        | Value        |
|------------|--------------|
| Name       | Joe Bloggs   |
| Age        | 42           |
| Occupation | Stunt Double |
| Height     | 175cm        |
| Weight     | 77kg         |

# Column-oriented Graph

| ColumnFamily |             |       |       |
|--------------|-------------|-------|-------|
| Row<br>Key   | Column Name |       |       |
|              | Key         | Key   | Key   |
|              | Value       | Value | Value |
|              | Column Name |       |       |
|              | Key         | Key   | Key   |
|              | Value       | Value | Value |

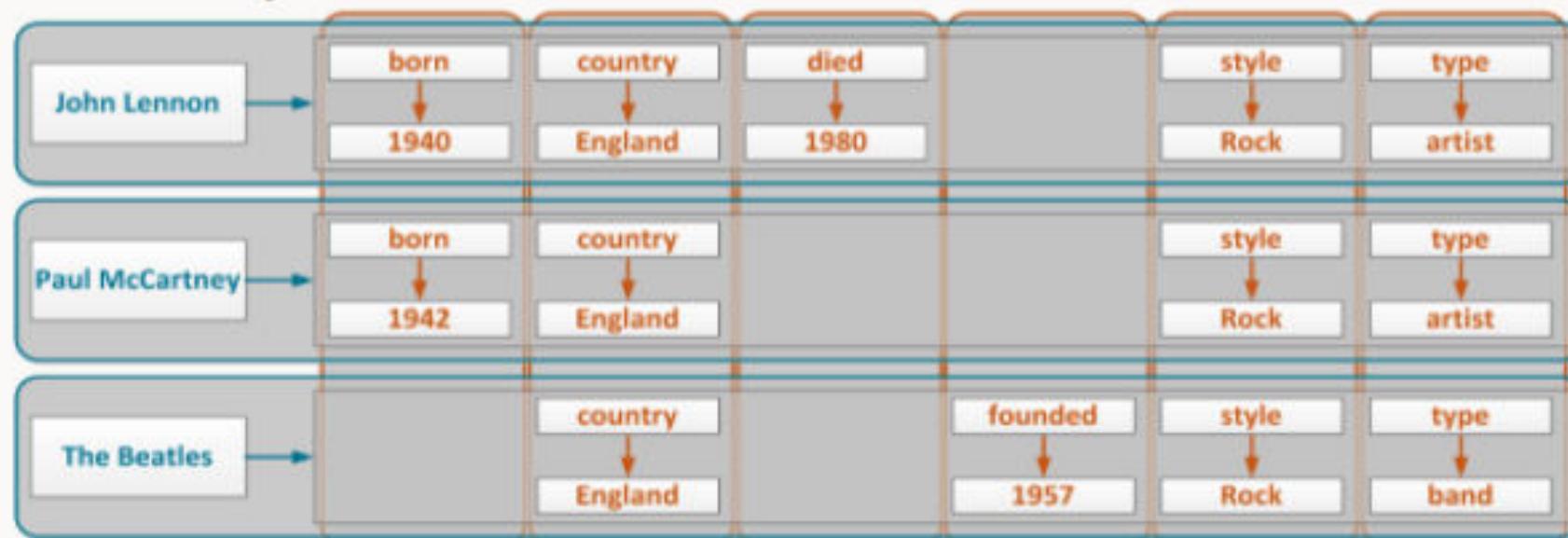
- column stores use columns to store data. You can group related columns into column families. Individual rows then constitute a column family



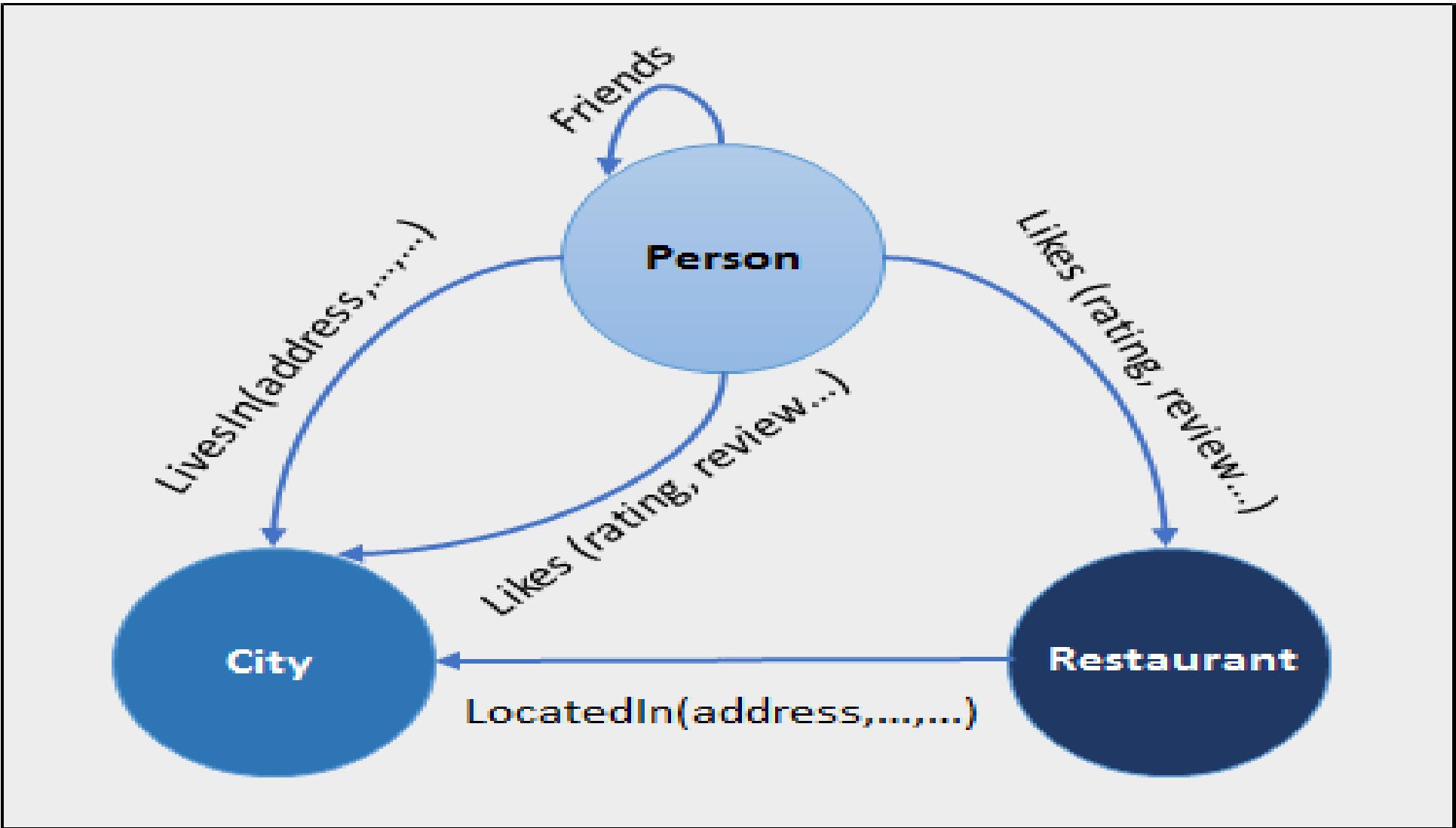
- Table with single-row partitions



- Column family view



# Graphs based



# Document-oriented

| Col1 | Col2 | Col3 | Col4 |
|------|------|------|------|
| Data | Data | Data | Data |
| Data | Data | Data | Data |
| Data | Data | Data | Data |

Document 1

```
{
 "prop1": data,
 "prop2": data,
 "prop3": data,
 "prop4": data
}
```

Document 2

```
{
 "prop1": data,
 "prop2": data,
 "prop3": data,
 "prop4": data
}
```

Document 3

```
{
 "prop1": data,
 "prop2": data,
 "prop3": data,
 "prop4": data
}
```

```
{
 first_name: 'Paul',
 surname: 'Miller',
 cell: 447557505611,
 city: 'London',
 location: [45.123,47.232],
 Profession: ['banking', 'finance', 'trader'],
 cars: [
 { model: 'Bentley',
 year: 1973,
 value: 100000, ... },
 { model: 'Rolls Royce',
 year: 1965,
 value: 330000, ... }
]
}
```

Fields

String

Number

Geo-Coordinates

Typed field values

Fields can contain arrays

Fields can contain an array of sub-documents

- **Types of NoSQL database:**

Types of NoSQL databases and the name of the databases system that falls in that category are:

- MongoDB falls in the category of NoSQL **document based database**.
- **Key value store:** Memcached, Redis, Coherence
- **Column :** Hbase, Big Table, Accumulo, Cassandra
- **Document based:** MongoDB, CouchDB, Cloudant

| SQL                                                           | NOSQL                                                          |
|---------------------------------------------------------------|----------------------------------------------------------------|
| RELATIONAL DATABASE MANAGEMENT SYSTEM (RDBMS)                 | Non-relational or distributed database system.                 |
| These databases have fixed or static or predefined schema     | They have dynamic schema                                       |
| These databases are not suited for hierarchical data storage. | These databases are best suited for hierarchical data storage. |
| These databases are best suited for complex queries           | These databases are not so good for complex queries            |
| Vertically Scalable                                           | Horizontally scalable                                          |

## Difference between RDBMS and NoSql

### RDBMS

- ▶ **Stands for Relational Database Management System**
- ▶ It is completely a structured way of storing data.
- ▶ The amount of data stored in RDBMS depends on physical memory of the system or in other words it is vertically scalable.
- ▶ In RDBMS schema represents logical view in which data is organized and tells how the relation are associates.
- ▶ It is a mixture of open and closed development models. like oracle, apache and so on.
- ▶ RDBMS databases are table based databases This means that SQL databases represent data in form of tables which consists of n number of rows of data
- ▶ RDBMS have predefined schemas.
- ▶ For defining and manipulating the data RDBMS use structured query language i.e. SQL which is very powerful.
- ▶ **RDBMS database examples:** MySql, Oracle, Sqlite, Postgres and MS-SQL.
- ▶ RDBMS database is well suited for the complex queries as compared to NoSql.
- ▶ If we talk about the type of data then RDBMS are not best fit for hierarchical data storage
- ▶ **Scalability:** RDBMS database is vertically scalable so to manage the increasing load by increase in CPU, RAM, SSD on a single server.
- ▶ RDBMS is best suited for high transactional based application and its more stable and promise for the atomicity and integrity of the data.
- ▶ RDBMS support large scale deployment and get support from there vendors.
- ▶ **Properties:** ACID properties(Atomicity, Consistency, Isolation, Durability).

### NoSql

- ▶ **Stands for Not Only SQL**
- ▶ It is completely a unstructured way of storing data.
- ▶ While in Nosql there is no limit you can scale it horizontally.
- ▶ Work on only open source development models.
- ▶ NoSQL databases are document based, key-value pairs, graph databases or wide-column stores. whereas NoSQL databases are the collection of key-value pair, documents, graph databases or wide-column stores which do not have standard schema definitions which it needs to adhered to.
- ▶ NoSql have dynamic schema with the unstructured data.
- ▶ It uses UnQL i.e. unstructured query language and focused on collection of documents and vary from database to database.
- ▶ **NoSQL database examples:** MongoDB, BigTable, Redis, RavenDb, Cassandra, Hbase, Neo4j and CouchDb
- ▶ NoSql is not well suited for complex queries on high level it does not have standard interfaces to perform that queries.
- ▶ NoSql is best fit for hierarchical data storage because it follows the key-value pair way of data similar to JSON. Hbase is the example for the same.
- ▶ **Scalability:** as we know Nosql database is horizontally scalable so to handle the large traffic you can add few servers to support that.
- ▶ NoSql is still rely on community support and for large scale NoSql deployment only limited experts are available.
- ▶ **Properties:** Follow Brewers CAP theorem(Consistency, Availability and Partition tolerance).

MySQL database is owned and maintained by the Oracle Corporation. So, has a good active community.

Best fit for data with tables and rows

Works better for small datasets

Frequent updates

Strong dependency on multi-row transactions

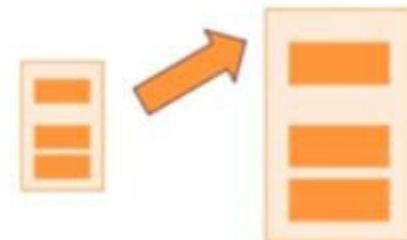
Modify large volume of records

The community of MySQL is much better than that of MongoDB.

Horizontal Scaling



Vertical Scaling



Add More Servers (and merge Data into one Database)

Improve Server Capacity / Hardware

Example of SQL

ORACLE



Example of NoSQL



- **Advantages of NoSQL:**

There are many advantages of working with NoSQL databases such as MongoDB and Cassandra. The main advantages are **high scalability** and **high availability**.

- **High scalability –**

NoSQL database use sharding for horizontal scaling. Partitioning of data and placing it on multiple machines in such a way that the order of the data is preserved is sharding.

Vertical scaling means adding more resources to the existing machine whereas horizontal scaling means adding more machines to handle the data. Vertical scaling is not that easy to implement but horizontal scaling is easy to implement. Examples of horizontal scaling databases are MongoDB, Cassandra etc. NoSQL can handle huge amount of data because of scalability, as the data grows NoSQL scale itself to handle that data in efficient manner.

- **High availability –**

Auto replication feature in NoSQL databases makes it highly available because in case of any failure data replicates itself to the previous consistent state.

- **Disadvantages of NoSQL:**  
NoSQL has the following disadvantages.
- **Narrow focus –**  
NoSQL databases have very narrow focus as it is **mainly designed for storage** but it provides very little functionality. Relational databases are a better choice in the field of Transaction Management than NoSQL.
- **Open-source –**  
NoSQL is open-source database. There is no reliable standard for NoSQL yet. In other words two database systems are likely to be unequal.
- **Management challenge –**  
The purpose of big data tools is to make management of a large amount of data as simple as possible. But it is not so easy. Data management in NoSQL is much more complex than a relational database. NoSQL, in particular, has a reputation for being challenging to install and even more hectic to manage on a daily basis.
- **GUI is not available –**  
GUI mode tools to access the database is not flexibly available in the market.
- **Backup –**  
Backup is a great weak point for some NoSQL databases like MongoDB. MongoDB has no approach for the backup of data in a consistent manner.
- **Large document size –**  
Some database systems like MongoDB and CouchDB store data in JSON format. Which means that documents are quite large (BigData, network bandwidth, speed), and having descriptive key names actually hurts, since they increase the document size.

# Introduction to MongoDB

- **MongoDB** is a cross-platform open-source **document-oriented database** program
- **MongoDB** database's has the ability to scale up with ease and hold very large amounts of data.
- **MongoDB** stores documents in collections within databases.
- **MongoDB** uses JSON-like documents (**JavaScript Object Notation**)
- [BSON](#) is a binary serialization format used to store documents on MongoDB
- In MongoDB, each document stored in a collection requires a unique [\\_id](#) field that acts as a [primary key](#).

# SQL to MongoDB Mapping Chart

| SQL Terms/Concepts                                              | MongoDB Terms/Concepts                                                          |
|-----------------------------------------------------------------|---------------------------------------------------------------------------------|
| database                                                        | database                                                                        |
| table                                                           | collection                                                                      |
| row                                                             | document or BSON document                                                       |
| column                                                          | field                                                                           |
| index                                                           | index                                                                           |
| table joins                                                     | \$lookup, embedded documents                                                    |
| primary key                                                     | primary key                                                                     |
| Specify any unique column or column combination as primary key. | In MongoDB, the primary key is automatically set to the <code>_id</code> field. |

# Database executables

|                 | MongoDB                | MySQL                  | Oracle                  | Informix                  | DB2                        |
|-----------------|------------------------|------------------------|-------------------------|---------------------------|----------------------------|
| Database Server | <a href="#">mongod</a> | <a href="#">mysqld</a> | <a href="#">oracle</a>  | <a href="#">IDS</a>       | <a href="#">DB2 Server</a> |
| Database Client | <a href="#">mongo</a>  | <a href="#">mysql</a>  | <a href="#">sqlplus</a> | <a href="#">DB-Access</a> | <a href="#">DB2 Client</a> |

- Document Database
- A record in MongoDB is a **document**, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects. The values of fields may include other **documents**, **arrays**, and **arrays of documents**

```
{
 name: "sue",
 age: 26,
 status: "A",
 groups: ["news", "sports"]
}
```



The diagram illustrates a MongoDB document structure. It consists of a left brace '{' followed by four key-value pairs: 'name: "sue"', 'age: 26', 'status: "A"', and 'groups: [ "news", "sports" ]'. To the right of each pair, a blue arrow points from the field name to its corresponding value, with the label 'field: value' written next to the arrow.

- The advantages of using documents are:
- Documents (i.e. objects) correspond to native data types in many programming languages.
- Embedded documents and arrays reduce need for expensive joins.
- Dynamic schema supports fluent polymorphism.

- **Collections/Views/On-Demand Materialized Views**
- MongoDB stores documents in [collections](#). Collections are analogous to tables in relational databases.
- In addition to collections, MongoDB supports:
- Read-only [Views](#) (Starting in MongoDB 3.4)
- [On-Demand Materialized Views](#) (Starting in MongoDB 4.2).

## MongoDB Web Shell

Click to connect

Connecting...

MongoDB shell version v4.2.0

connecting to: mongodb://127.0.0.1:27017/?

authSource=admin&compressors=disabled&gssapiServiceName=mongodb

Implicit session: session { "id" : UUID("007c14c9-a959-4272-a6a0-1cbd81945877") }

MongoDB server version: 4.2.0

type "help" for help

>>> show dbs

admin 0.000GB

config 0.000GB

local 0.000GB

>>> show collections

>>> |

# MongoDB **CRUD** Operations

**C-create/insert**

**R-query**

**U- update**

**D-delete**

Create or insert operations add new [\*\*documents\*\*](#) to a [\*\*collection\*\*](#).

If the collection does not currently exist, insert operations will create the collection.

MongoDB provides the following **methods** to insert documents into a collection:

•[\*\*db.collection.insertOne\(\)\*\*](#) *New in version 3.2*

•[\*\*db.collection.insertMany\(\)\*\*](#) *New in version 3.2*

## Insert a Single Document

`db.collection.insertOne()` inserts a *single* document into a collection.

The following example inserts a new document into the `inventory` collection. If the document does not specify an `_id` field, MongoDB adds the `_id` field with an ObjectId value to the new document. See [Insert Behavior](#).

copy

```
db.inventory.insertOne(
 { item: "canvas", qty: 100, tags: ["cotton"], size: { h: 28, w: 35.5, uom: "cm" } }
)
```



embedded

## Insert Multiple Documents

*New in version 3.2.*

`db.collection.insertMany()` can insert *multiple* documents into a collection. Pass an array of documents to the method.

The following example inserts three new documents into the `inventory` collection. If the documents do not specify an `_id` field, MongoDB adds the `_id` field with an `ObjectId` value to each document. See [Insert Behavior](#).

copy

```
db.inventory.insertMany([
 { item: "journal", qty: 25, tags: ["blank", "red"], size: { h: 14, w: 21, uom: "cm" } },
 { item: "mat", qty: 85, tags: ["gray"], size: { h: 27.9, w: 35.5, uom: "cm" } },
 { item: "mousepad", qty: 25, tags: ["gel", "blue"], size: { h: 19, w: 22.85, uom: "cm" } }
])
```

# Query documents

## Select All Documents in a Collection

- To select all documents in the collection, pass an **empty document** as the **query filter parameter** to the **find method**. The query filter parameter determines the select criteria:

```
db.inventory.find({})
```

- This is equivalent to ***select \* from inventory***

# Specify Equality Condition

To specify equality conditions, use <field>:<value> expressions in the [query filter document](#)

```
db.inventory.find ({ status : "D" })
```

This is equivalent to

```
Select * from inventory where status="D";
```

# Specify Conditions Using Query Operators

The following example retrieves all documents from the `inventory` collection where `status` equals **either "A" or "D"**

```
db.inventory.find ({ status : { $in : ["A" , "D"] } })
```

You can express it using `$or` operator also

```
db.inventory.find ({ status : { $or : ["A" , "D"] } })
```

# Specify AND Conditions

A compound query can specify conditions for more than one field in the collection's documents. Implicitly, a logical AND conjunction connects the clauses of a compound query so that the query selects the documents in the collection that match all the conditions.

The following query finds all the documents in the inventory whose status is “D” and quantity is **less than** 30

```
db.inventory.find ({ status :"D" , qty: { $lt:30 } })
```

## Specify OR Conditions

Using the `$or` operator, you can specify a compound query that joins each clause with a logical OR conjunction so that the query selects the documents in the collection that match at least one condition.

The following example retrieves all documents in the collection where the `status` equals "A" or `qty` is less than (`$lt`) 30:

copy

```
db.inventory.find({ $or: [{ status: "A" }, { qty: { $lt: 30 } }] })
```

The operation corresponds to the following SQL statement:

copy

```
SELECT * FROM inventory WHERE status = "A" OR qty < 30
```

## Specify AND as well as OR Conditions

In the following example, the compound query document selects all documents in the collection where the `status` equals "A" **and** either `qty` is less than (`$lt`) 30 **or** `item` starts with the character p:

```
db.inventory.find({
 status: "A",
 $or: [{ qty: { $lt: 30 } }, { item: /^p/ }]
})
```

copy

The operation corresponds to the following SQL statement:

```
SELECT * FROM inventory WHERE status = "A" AND (qty < 30 OR item LIKE "p%")
```

copy

### NOTE:

MongoDB supports regular expressions `$regex` queries to perform string pattern matches.

# Query on Embedded/Nested Documents

- Populate the collection

copy

```
db.inventory.insertMany([
 { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
 { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "A" },
 { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
 { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
 { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" }
]);
```

## Match an Embedded/Nested Document

To specify an equality condition on a field that is an embedded/nested document, use the [query filter document](#) { <field>: <value> } where <value> is the document to match.

For example, the following query selects all documents where the field **size** equals the document { h: 14, w: 21, uom: "cm" }:

copy

```
db.inventory.find({ size: { h: 14, w: 21, uom: "cm" } })
```

Equality matches on the whole embedded document require an *exact* match of the specified <value> document, including the field order. For example, the following query does not match any documents in the **inventory** collection:

copy

```
db.inventory.find({ size: { w: 21, h: 14, uom: "cm" } })
```

## Query on Nested Field

To specify a query condition on fields in an embedded/nested document, use dot notation ("field.nestedField").

**NOTE:**

When querying using dot notation, the field and nested field must be inside quotation marks.

### Specify Equality Match on a Nested Field

The following example selects all documents where the field `uom` nested in the `size` field equals "in":

```
db.inventory.find({ "size.uom": "in" })
```

copy

## Specify Match using Query Operator

A query filter document can use the [query operators](#) to specify conditions in the following form:

copy

```
{ <field1>: { <operator1>: <value1> }, ... }
```

The following query uses the less than operator (`$lt`) on the field `h` embedded in the `size` field:

copy

```
db.inventory.find({ "size.h": { $lt: 15 } })
```

## Specify AND Condition

The following query selects all documents where the nested field `h` is less than `15`, the nested field `uom` equals `"in"`, and the `status` field equals `"D"`:

copy

```
db.inventory.find({ "size.h": { $lt: 15 }, "size.uom": "in", status: "D" })
```

# Query an Array

Cop

```
db.inventory.insertMany([
 { item: "journal", qty: 25, tags: ["blank", "red"], dim_cm: [14, 21] },
 { item: "notebook", qty: 50, tags: ["red", "blank"], dim_cm: [14, 21] },
 { item: "paper", qty: 100, tags: ["red", "blank", "plain"], dim_cm: [14, 21] },
 { item: "planner", qty: 75, tags: ["blank", "red"], dim_cm: [22.85, 30] },
 { item: "postcard", qty: 45, tags: ["blue"], dim_cm: [10, 15.25] }
]);
```

## Match an Array

To specify equality condition on an array, use the query document { <field>: <value> } where <value> is the exact array to match, including the order of the elements.

The following example queries for all documents where the field `tags` value is an array with exactly two elements, "red" and "blank", in the specified order:

copy

```
db.inventory.find({ tags: ["red", "blank"] })
```

If, instead, you wish to find an array that contains both the elements "red" and "blank", without regard to order or other elements in the array, use the `$all` operator:

copy

```
db.inventory.find({ tags: { $all: ["red", "blank"] } })
```

## Query an Array for an Element

To query if the array field contains at least **one** element with the specified value, use the filter { <field>: <value> } where <value> is the element value.

The following example queries for all documents where tags is an array that contains the string "red" as one of its elements:

```
db.inventory.find({ tags: "red" })
```

copy

To specify conditions on the elements in the array field, use [query operators](#) in the query filter document:

copy

```
{ <array field>: { <operator1>: <value1>, ... } }
```

For example, the following operation queries for all documents where the array `dim_cm` contains at least one element whose value is greater than 25.

copy

```
db.inventory.find({ dim_cm: { $gt: 25 } })
```

## Query an Array with Compound Filter Conditions on the Array Elements

The following example queries for documents where the `dim_cm` array contains elements that in some combination satisfy the query conditions; e.g., one element can satisfy the greater than 15 condition and another element can satisfy the less than 20 condition, or a single element can satisfy both:

copy

```
db.inventory.find({ dim_cm: { $gt: 15, $lt: 20 } })
```

## Query for an Array Element that Meets Multiple Criteria

Use `$elemMatch` operator to specify multiple criteria on the elements of an array such that at least one array element satisfies all the specified criteria.

The following example queries for documents where the `dim_cm` array contains at least one element that is both greater than (`$gt`) 22 and less than (`$lt`) 30:

copy

```
db.inventory.find({ dim_cm: { $elemMatch: { $gt: 22, $lt: 30 } } })
```

## Query for an Element by the Array Index Position

Using [dot notation](#), you can specify query conditions for an element at a particular index or position of the array. The array uses zero-based indexing.

**NOTE:**

When querying using dot notation, the field and nested field must be inside quotation marks.

The following example queries for all documents where the second element in the array `dim_cm` is greater than 25:

```
db.inventory.find({ "dim_cm.1": { $gt: 25 } })
```

copy

## Query an Array by Array Length

Use the `$size` operator to query for arrays by number of elements. For example, the following selects documents where the array `tags` has 3 elements.

copy

```
db.inventory.find({ "tags": { $size: 3 } })
```