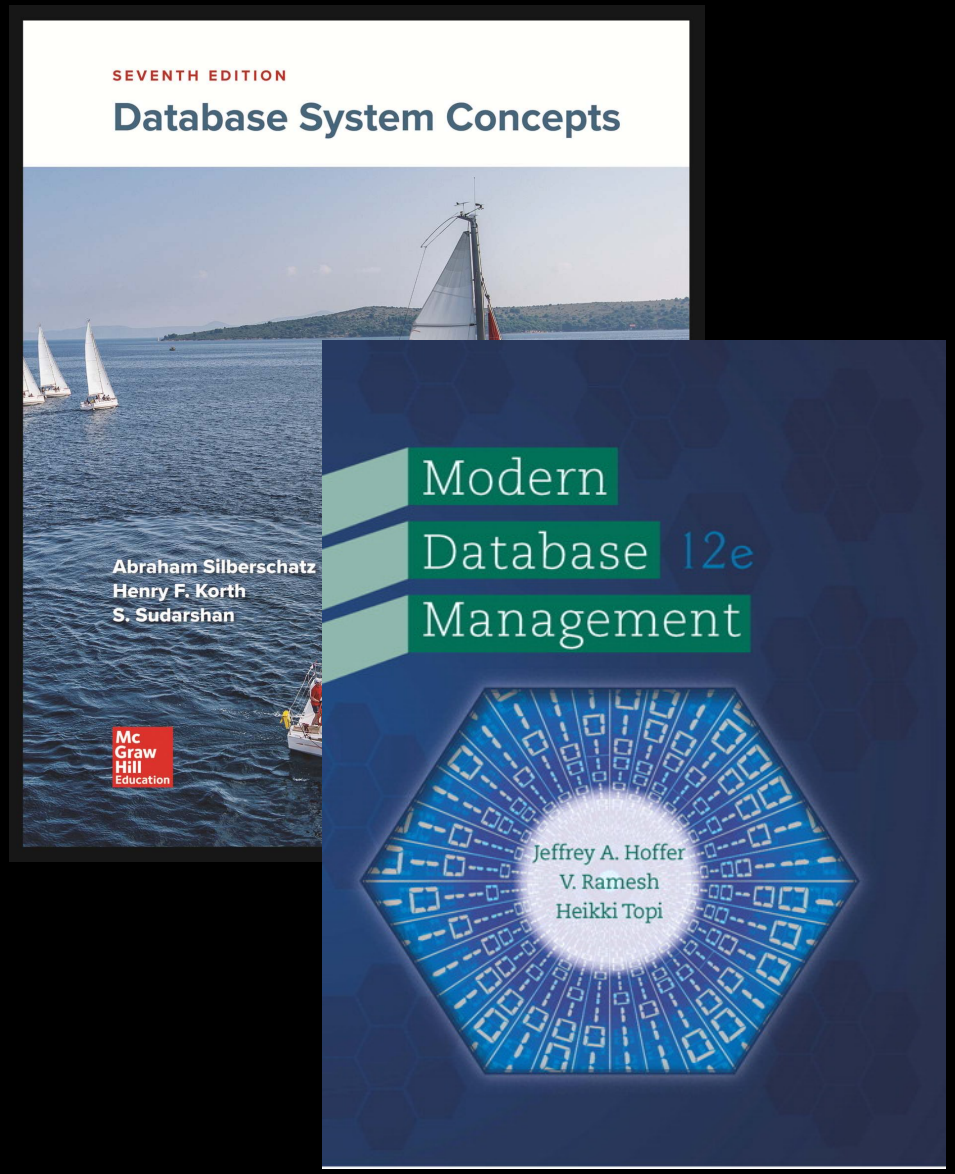
IF2240 – Basis Data

Introduction

KNOWLEDGE & SOFTWARE ENGINEERING



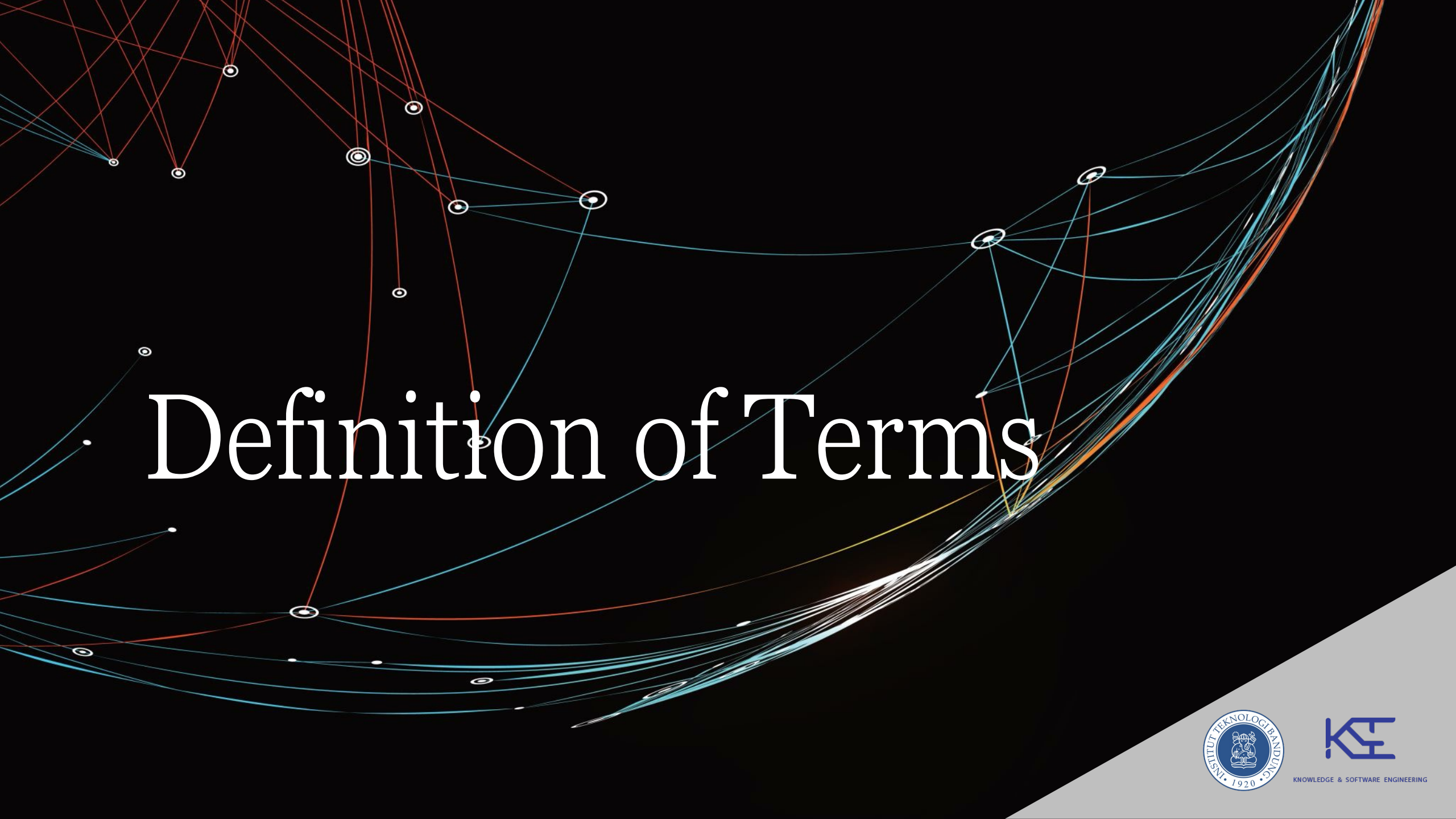
References

Abraham Silberschatz, Henry F. Korth, S. Sudarshan : "**Database System Concepts**", 7th Edition

- Chapter 1: Introduction

Jeffrey A. Hoffer, Mary B. Prescott, Heikki Topi : "**Modern Database Management**", 12th Edition

- Chapter 1: The Database Environment and Development Process



Definition of Terms

KNOWLEDGE & SOFTWARE ENGINEERING

Definition (1/3)

Database	organized collection of logically related data
Data	stored representations of meaningful objects and events Structured: numbers, text, dates Unstructured: images, video, documents

Example:

Baker, Kenneth D.	324917628
Doyle, Joan E.	476193248
Finkle, Clive R.	548429344
Lewis, John C.	551742186
McFerran, Debra R.	409723145

Definition (2/3)

Information	data processed to increase knowledge in the person using the data

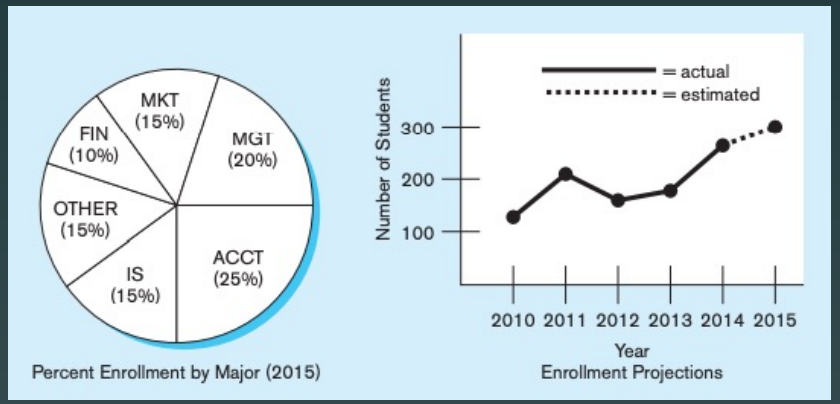
Example:

Class Roster

Course: MGT 500 Business Policy Semester: Spring 2015

Section: 2

Name	ID	Major	GPA
Baker, Kenneth D.	324917628	MGT	2.9
Doyle, Joan E.	476193248	MKT	3.4
Finkle, Clive R.	548429344	PRM	2.8
Lewis, John C.	551742186	MGT	3.7
McFerran, Debra R.	409723145	IS	2.9
Sisneros, Michael	392416582	ACCT	3.3



Summarized data - Graphical displays turn data into useful information that managers can use for decision making and interpretation

Data in context - Context helps users understand data

Definition (3/3)

Metadata	data that describes the properties and context of user data

Example:

TABLE 1-1 Example Metadata for Class Roster

Data Item	Metadata					
Name	**Type**	**Length**	**Min**	**Max**	**Description**	**Source**
Course	Alphanumeric	30			Course ID and name	Academic Unit
Section	Integer	1	1	9	Section number	Registrar
Semester	Alphanumeric	10			Semester and year	Registrar
Name	Alphanumeric	30			Student name	Student IS
ID	Integer	9			Student ID (SSN)	Student IS
Major	Alphanumeric	4			Student major	Student IS
GPA	Decimal	3	0.0	4.0	Student grade point average	Academic Unit

Descriptions of the properties or characteristics of the data, including data types, field sizes, allowable values, and data context

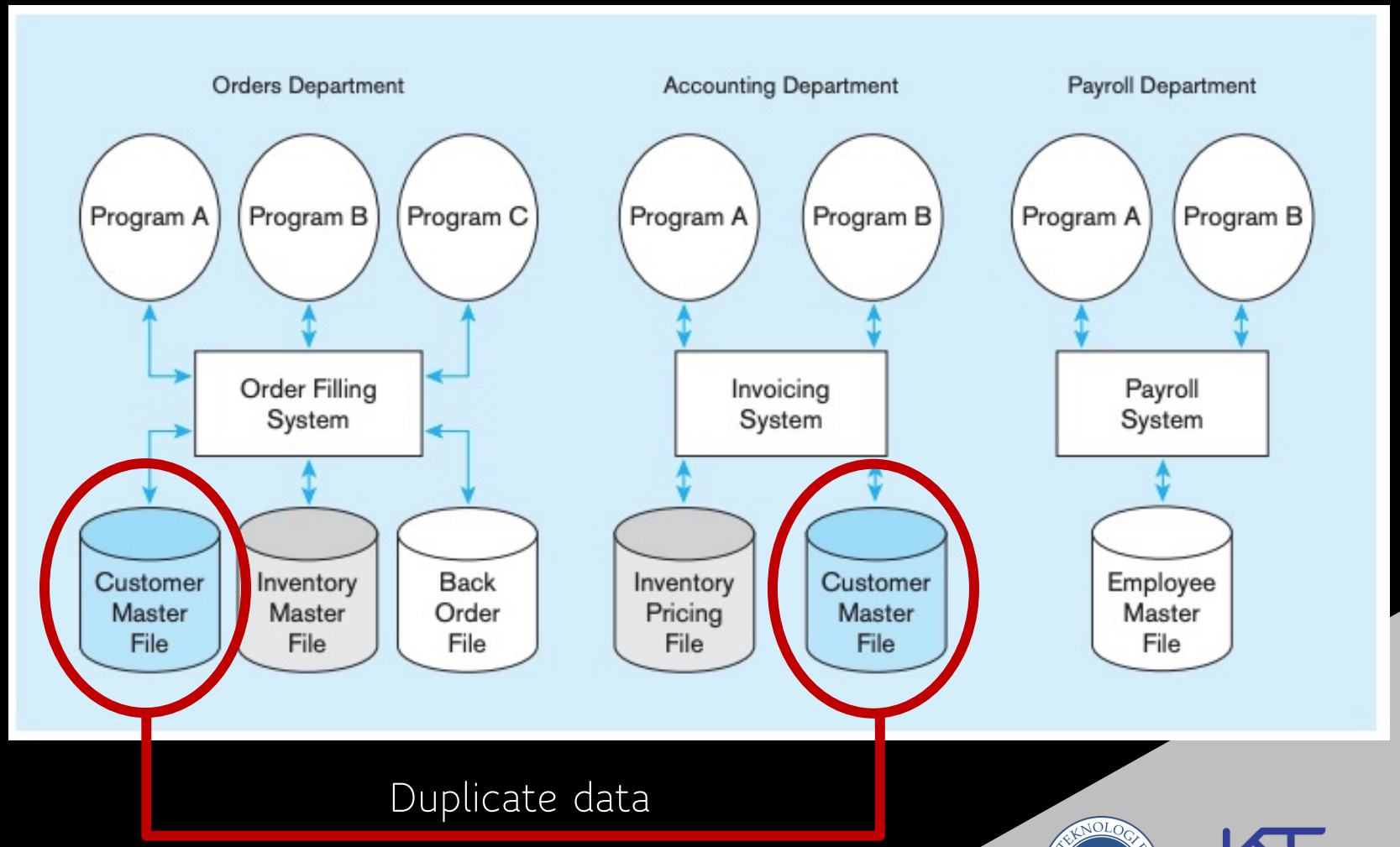
File Processing System vs The Database Approach

KNOWLEDGE & SOFTWARE ENGINEERING

Traditional File Processing System

When computer-based data processing was first available, there were no databases. To be useful for business applications, computers had to store, manipulate, and retrieve large files of data.

File processing systems at Pine Valley Furniture Company



Disadvantages of File Processing (1)

Program-Data Dependence	Data Redundancy (Duplication of data)	Limited Data Sharing	Lengthy Development Times	Excessive Program Maintenance
· All programs maintain metadata for each file they use	· Different systems/ programs have separate copies of the same data	· No centralized control of data	· Programmers must design their own file formats	· 80% of information systems budget

Disadvantages of File Processing (2)

Difficulty in accessing data	Data isolation	Integrity problems	Atomicity of updates	Concurrent access by multiple users	Security problems
· Need to write a new program to carry out each new task	· Multiple files and formats	· 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	· Failures may leave database in an inconsistent state with partial updates carried out	· Concurrent access needed for performance · Uncontrolled concurrent accesses can lead to inconsistencies	· Hard to provide user access to some, but not all, data

Problems with **Data Dependency**

Each application programmer must maintain their own data

Each application program needs to include code for the metadata of each file

Each application program must have its own processing routines for reading, inserting, updating and deleting data

Lack of coordination and central control

Non-standard file formats

Problems with **Data Redundancy**

Waste of space to have duplicate data

Causes more maintenance headaches

The biggest Problem:

- When data changes in one file, could cause inconsistencies
- Compromises *data integrity*

Solution : The Database Approach

CENTRAL REPOSITORY OF SHARED DATA

DATA IS MANAGED BY A CONTROLLING AGENT

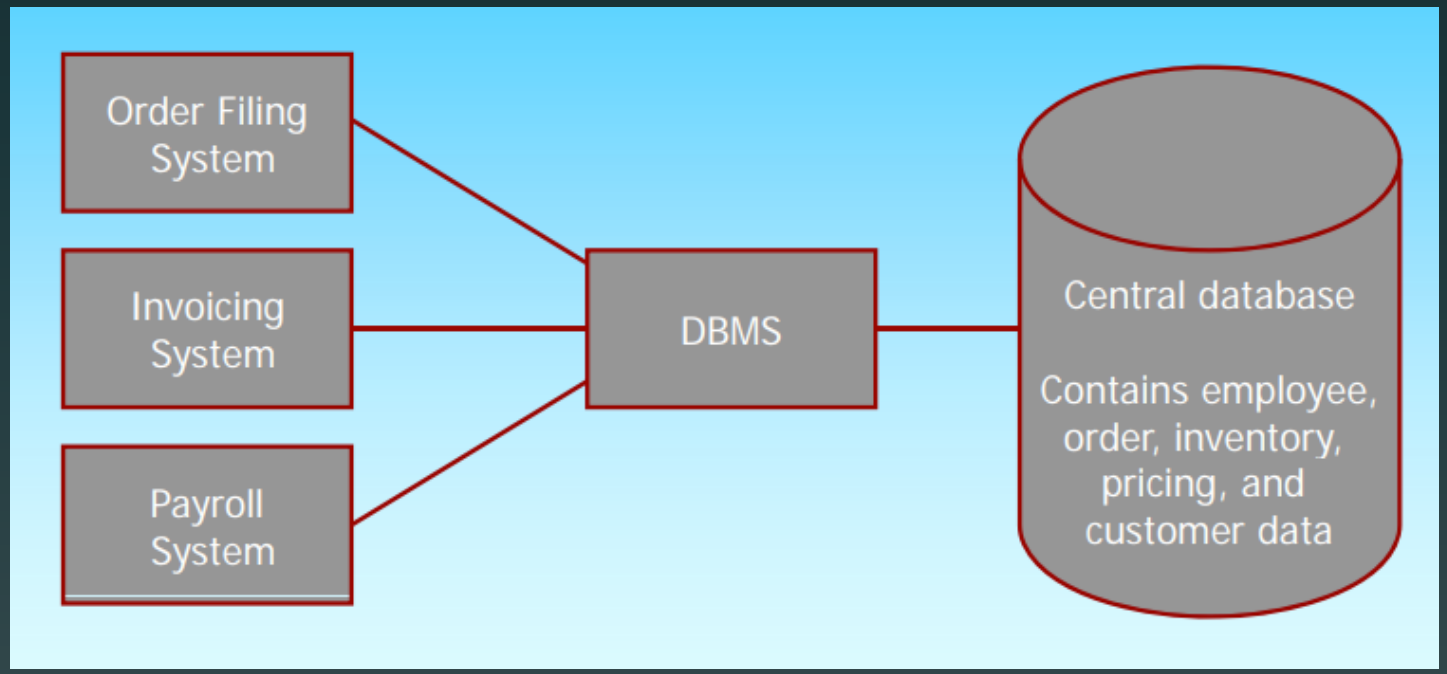
STORED IN A STANDARDIZED, CONVENIENT FORM

Requires a **Database Management System (DBMS)**

Database Management System (DBMS)

A software system that is used to create, maintain, and provide controlled access to user databases

DBMS manages data resources like an operating system manages hardware resources



Advantages of Database Approach

Improved Data Sharing	Enforcement of Standards	Improved Data Quality	Better Data Accessibility/ Responsiveness	Security, Backup/Recovery, Concurrency
· Different users get different views of the data	· All data access is done in the same way	· Constraints, data validation rules	· Use of standard data query language (SQL)	· Disaster recovery is easier

Costs and Risks of the Database Approach

Up-front costs

- Installation Management Cost and Complexity
- Conversion Costs

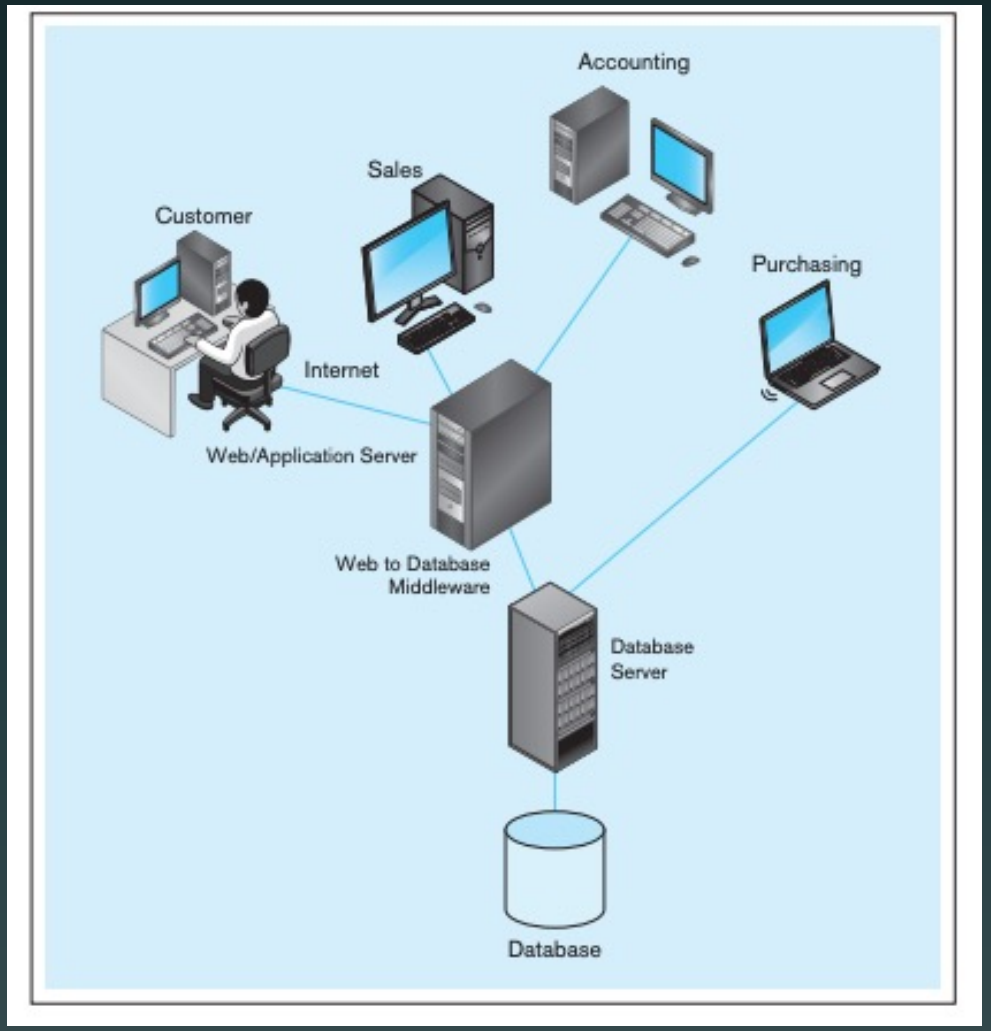
Ongoing Costs

- Requires New, Specialized Personnel
- Need for Explicit Backup and Recovery

Organizational Conflict

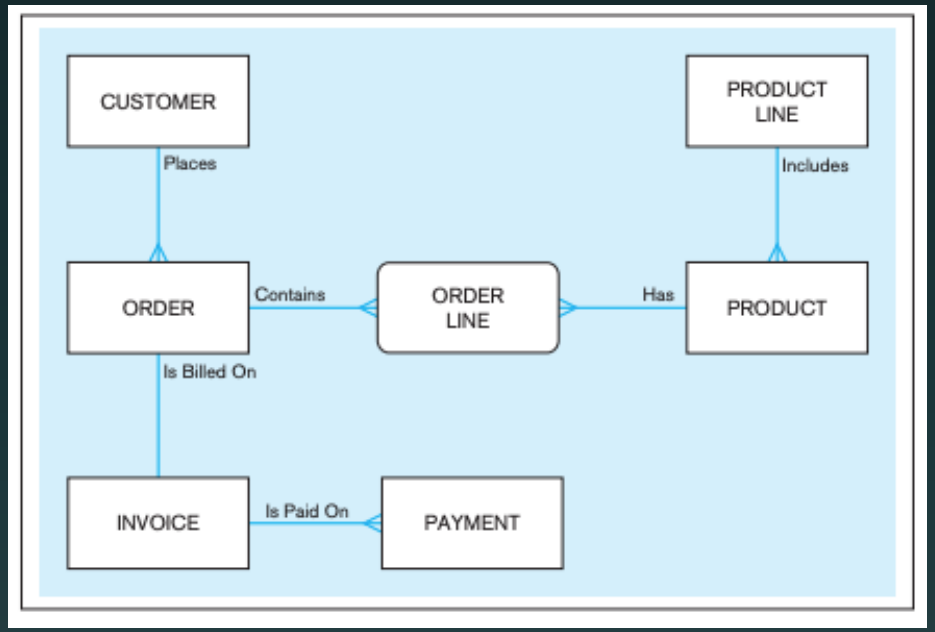
- Old habits die hard

Developing a **database** application for Pine Valley Furniture Company (1/4)

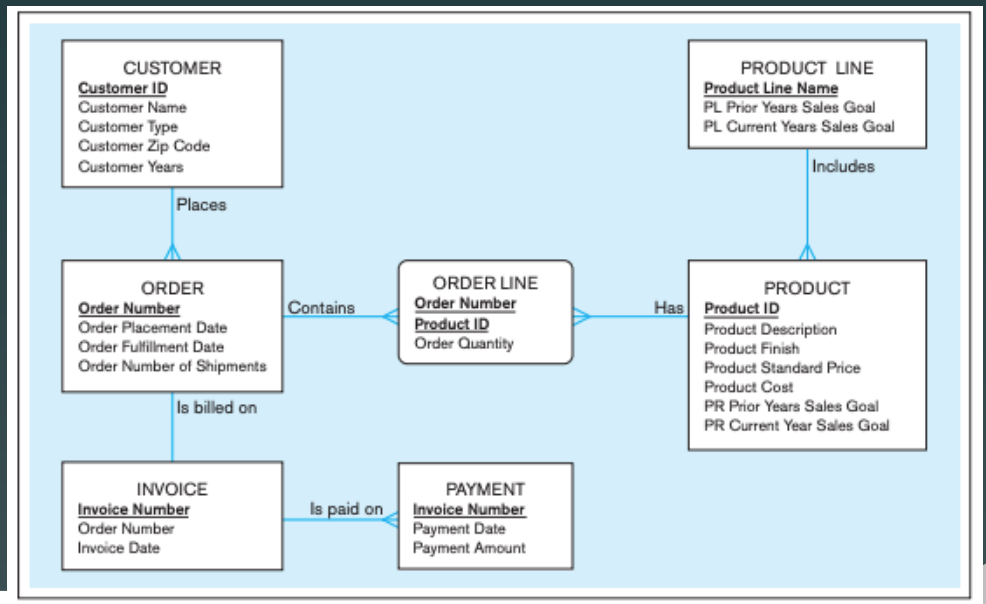


Computer System for Pine Valley Furniture Company

Developing a **database** application for Pine Valley Furniture Company (2/4)



Preliminary data model for Home Office product line marketing support system

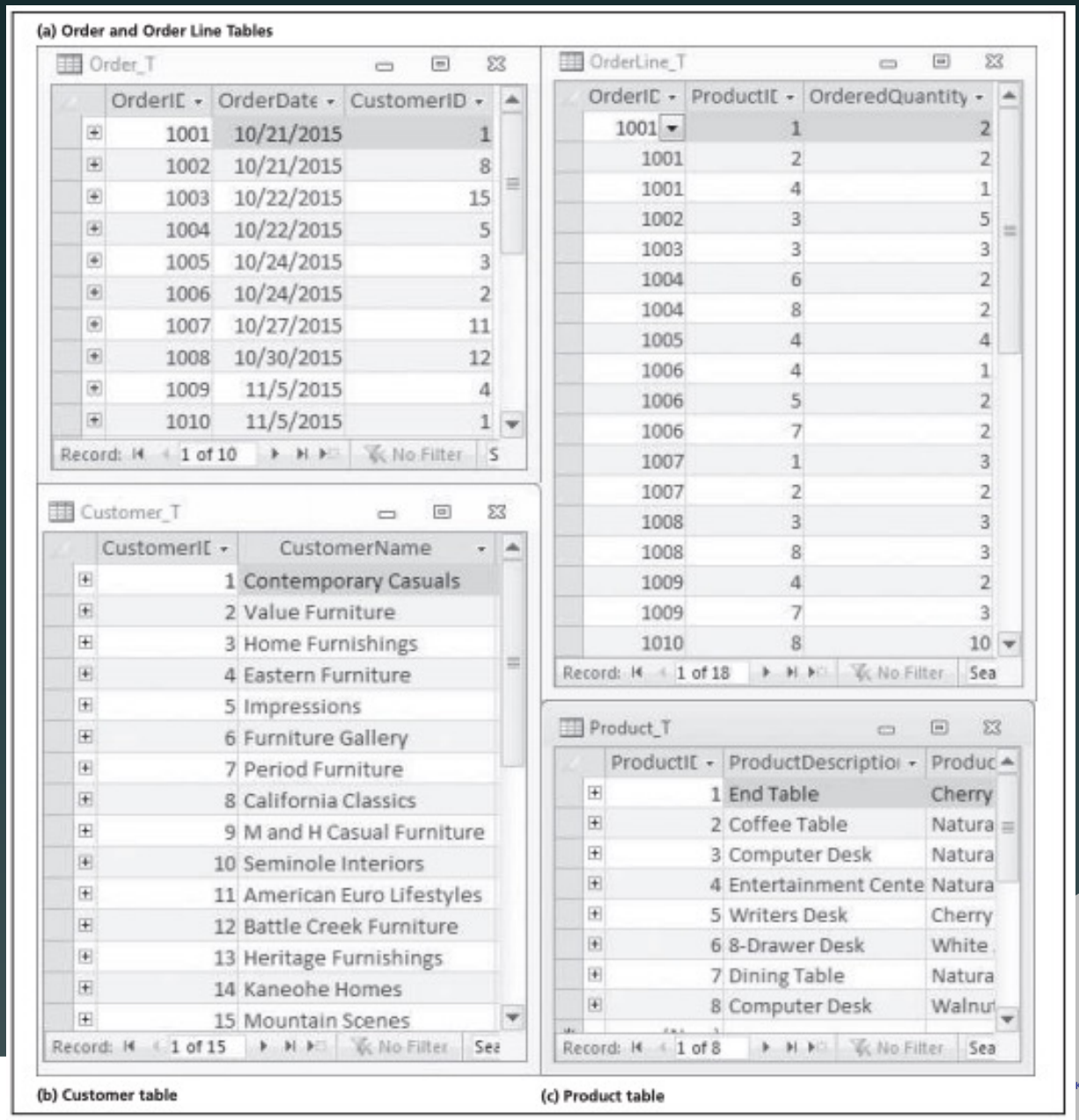


Project data model for Home Office product line marketing support system

KNOWLEDGE & SOFTWARE ENGINEERING

Developing a **database** application for Pine Valley Furniture Company (3/4)

Four **relations** (Pine Valley Furniture Company)



(a) Order and Order Line Tables

Order_T

OrderID	OrderDate	CustomerID
1001	10/21/2015	1
1002	10/21/2015	8
1003	10/22/2015	15
1004	10/22/2015	5
1005	10/24/2015	3
1006	10/24/2015	2
1007	10/27/2015	11
1008	10/30/2015	12
1009	11/5/2015	4
1010	11/5/2015	1

Record: 1 of 10 No Filter

OrderLine_T

OrderID	ProductID	OrderedQuantity
1001	1	2
1001	2	2
1001	4	1
1002	3	5
1003	3	3
1004	6	2
1004	8	2
1005	4	4
1006	4	1
1006	5	2
1006	7	2
1007	1	3
1007	2	2
1008	3	3
1008	8	3
1009	4	2
1009	7	3
1010	8	10

Record: 1 of 18 No Filter

Customer_T

CustomerID	CustomerName
1	Contemporary Casuals
2	Value Furniture
3	Home Furnishings
4	Eastern Furniture
5	Impressions
6	Furniture Gallery
7	Period Furniture
8	California Classics
9	M and H Casual Furniture
10	Seminole Interiors
11	American Euro Lifestyles
12	Battle Creek Furniture
13	Heritage Furnishings
14	Kaneohe Homes
15	Mountain Scenes

Record: 1 of 15 No Filter

Product_T

ProductID	ProductDescriptio	Produc
1	End Table	Cherry
2	Coffee Table	Natura
3	Computer Desk	Natura
4	Entertainment Cente	Natura
5	Writers Desk	Cherry
6	8-Drawer Desk	White
7	Dining Table	Natura
8	Computer Desk	Walnu

Record: 1 of 8 No Filter

(b) Customer table

(c) Product table

Developing a **database** application for Pine Valley Furniture Company (4/4)

```
SELECT Product.ProductID, Product.ProductDescription, Product.PRCurrentYearSalesGoal,
       (OrderQuantity * ProductPrice) AS SalesToDate
FROM Order.OrderLine, Product.ProductLine
WHERE Order.OrderNumber = OrderLine.OrderNumber
AND Product.ProductID = OrderedProduct.ProductID
AND Product.ProductID = ProductLine.ProductID
AND Product.ProductLineName = "Home Office";
```

SQL query for Home Office sales-to-goal comparison

Home Office Sales to Date : Select Query

Product ID	Product Description	PR Current Year Sales Goal	Sales to Date
3	Computer Desk	$23,500.00	5625
10	96" Bookcase	$22,500.00	4400
5	Writer's Desk	$26,500.00	650
3	Computer Desk	$23,500.00	3750
7	48" Bookcase	$17,000.00	2250
5	Writer's Desk	$26,500.00	3900

Home Office product line sales comparison

Customer invoice (Pine Valley Furniture Company)

PVFC Customer Invoice

Customer ID 2
Customer Name Value Furniture
Address 15145 S.W. 17th St.
Plano, TX 75094

Order Number 1006
Order Date 10/24/2000

Product_ID	Product Description	Finish	Quantity	Unit Price	Extended Price:
7	Dining Table	Natural As	2	$800.00	$1,600.00
5	Writer's Desk	Cherry	2	$325.00	$650.00
4	Entertainment Center	Natural M	1	$650.00	$650.00
				Total	$2,900.00

Database Applications

KNOWLEDGE & SOFTWARE ENGINEERING

Components of the Database Environment

CASE Tools – computer-aided software engineering

Repository – centralized storehouse of metadata

Database Management System (DBMS) – software for managing the database

Database – storehouse of the data

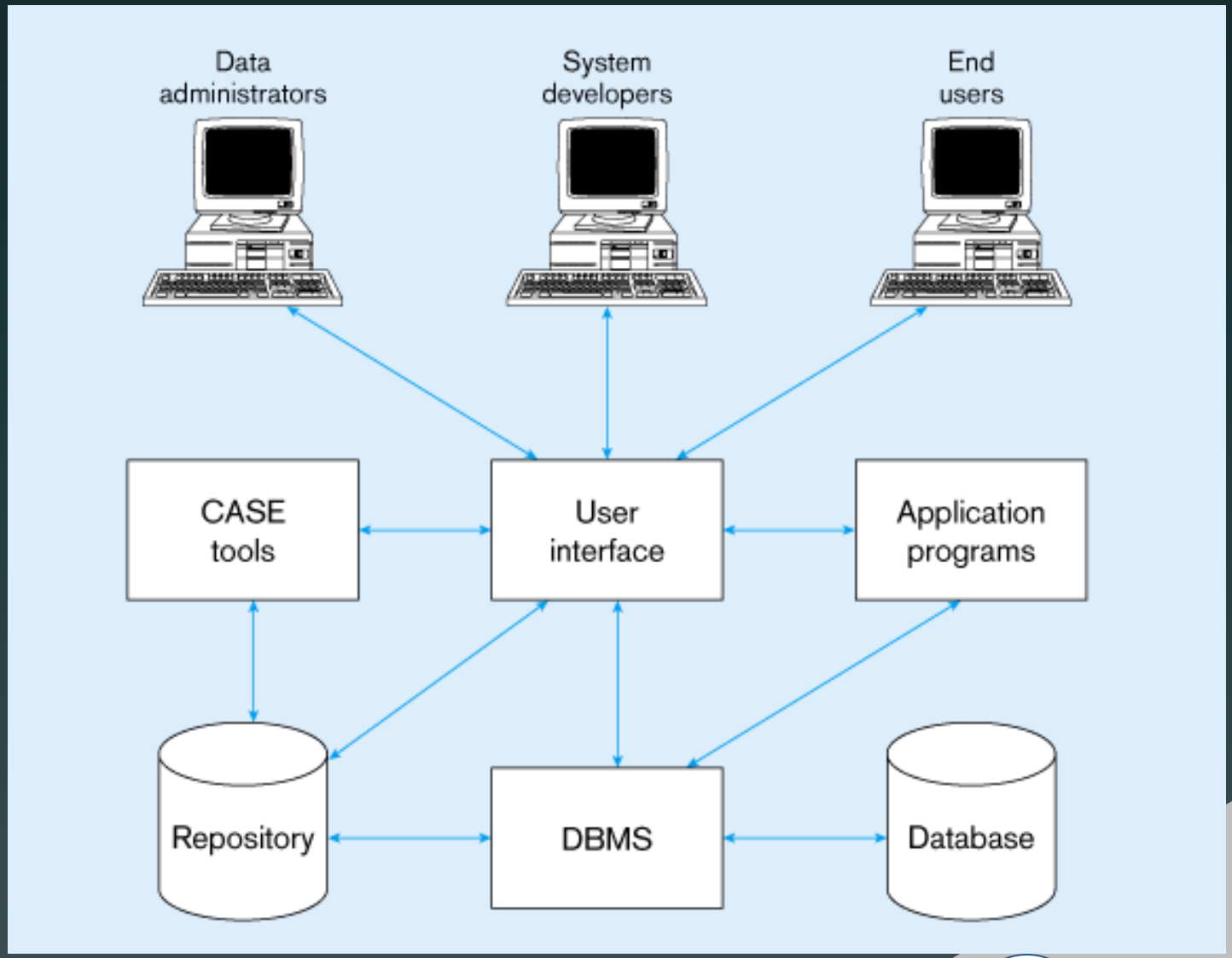
Application Programs – software using the data

User Interface – text and graphical displays to users

Data Administrators – personnel responsible for maintaining the database

System Developers – personnel responsible for designing databases and software

End Users – people who use the applications and databases



The Range of Database Applications

Personal databases

Workgroup databases

Departmental/ divisional databases

Enterprise database

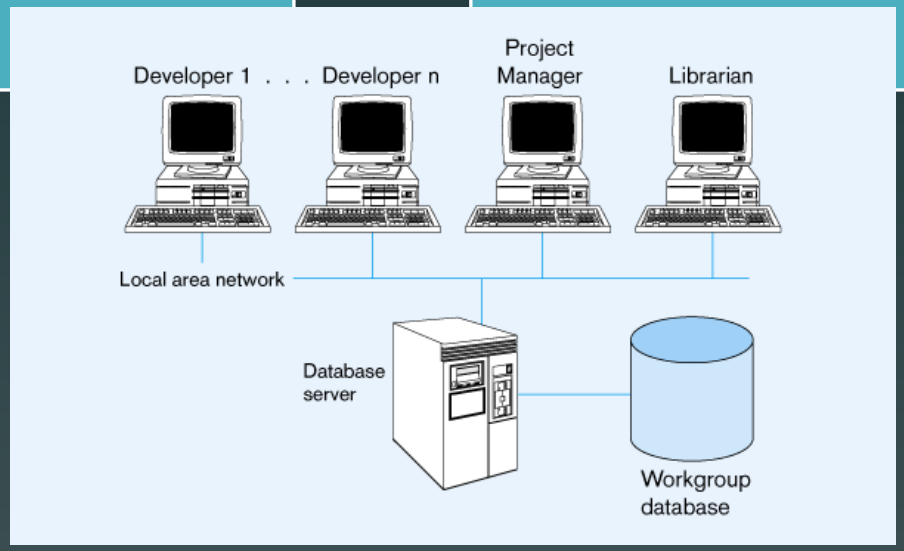
- Enterprise resource planning (ERP) systems
- Data warehousing implementations

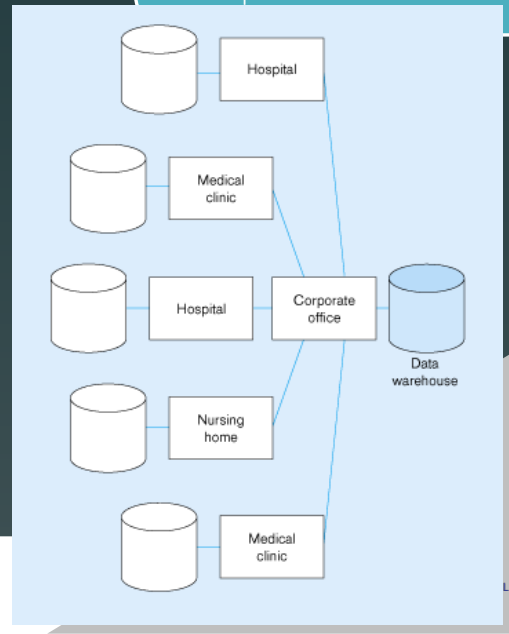
Customer

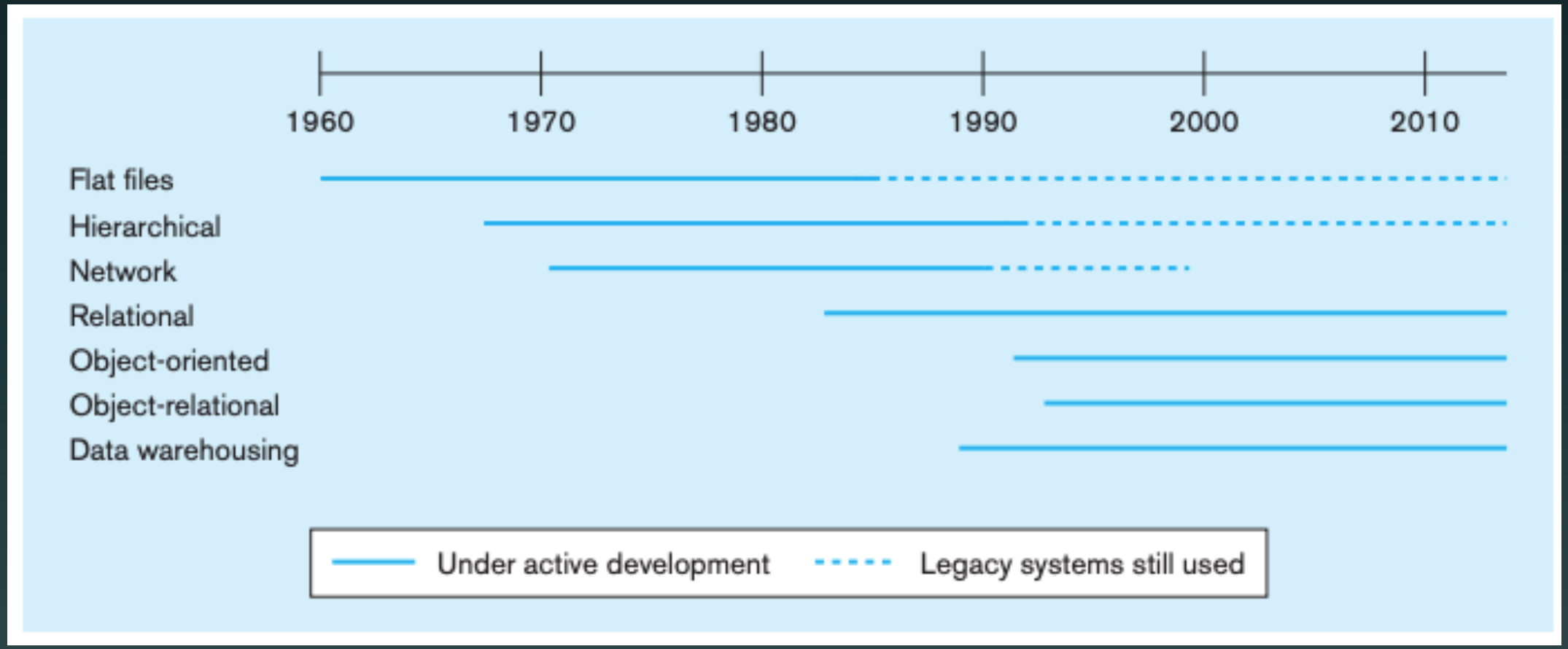
Customer Name: **Multi Media, Inc.**
Address: **1000 River Road**
City: **San Antonio**
State: **TX**
Zip: **76235**
Phone: **(219) 864-2000**
Next Contact Date: **10/17/2000** / Time: **10:30 AM**

Contact History for Customer

Date	Time	Contact	Comments
08/04/2000	10:00 AM	Roberts	Review proposal
08/19/2000	08:00 AM	Roberts	Revise schedule
09/10/2000	09:00 AM	Pearson	Sign contract
09/21/2000	02:00 PM	Roberts	Follow up







Evolution of Database Technologies

More on Database Approach

Modified form Silberschatz's slides, Database System Concept, 7th edition

KNOWLEDGE & SOFTWARE ENGINEERING

Levels of Abstraction

Physical level: describes how a record is stored.

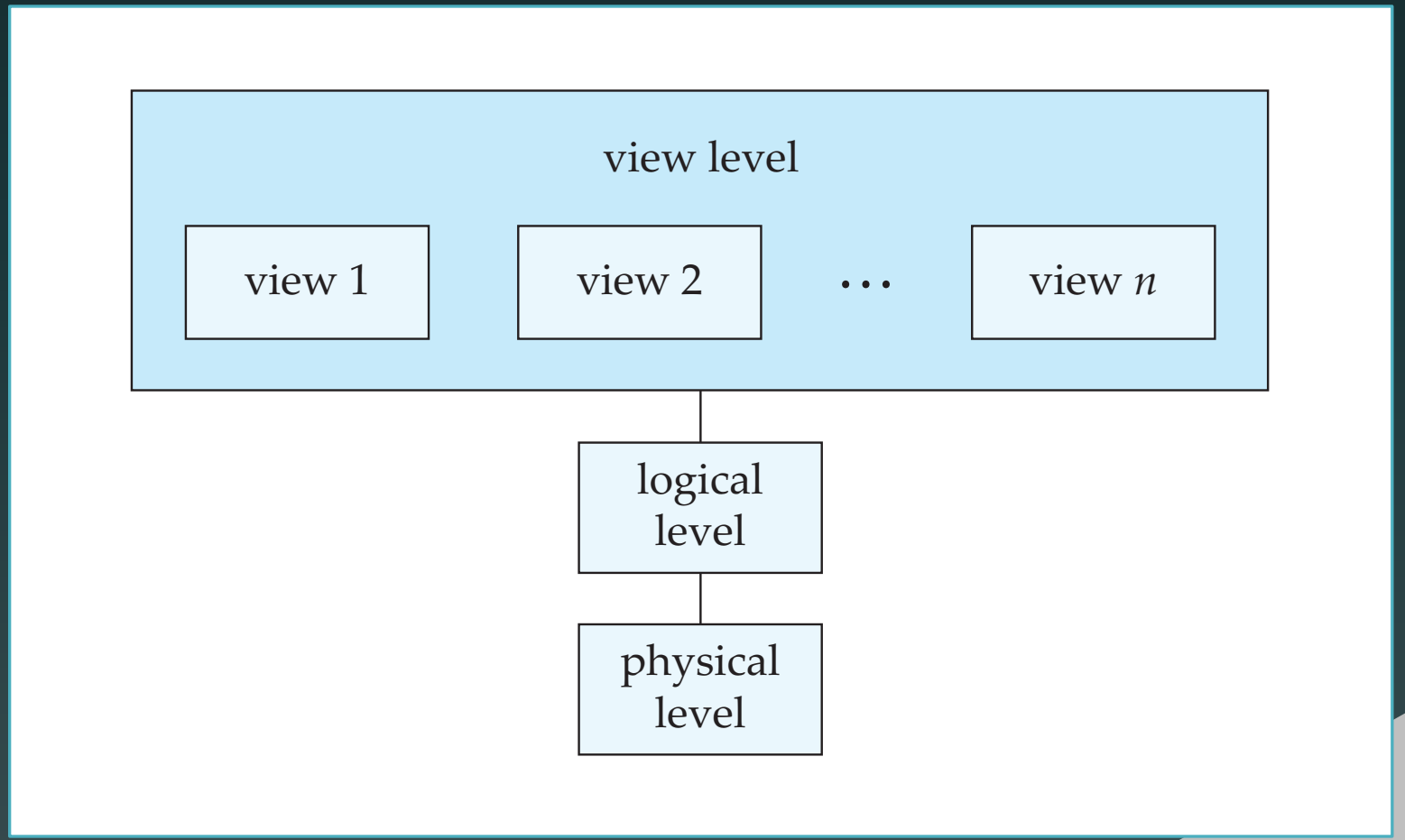
Logical level: describes data stored in database, and the relationships among the data.

```
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 (ANSI/SPARC Architecture)



Physical Data Independence

Physical Data Independence – the ability to modify the physical schema without changing the logical schema

- Applications depend on the logical schema
- In general, the interfaces between the various levels and components should be well defined so that changes in some parts do not seriously influence others.

Instances and Schemas

Schema : structure of the database

Logical Schema – the overall logical structure of the database

Physical Schema – the overall physical structure of the database

Instance : the actual content of the database at a particular point in time

Similar to types and variables in programming languages

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*** - contains metadata (i.e., data about data)

- Database schema
- Integrity constraints
  - Primary key (ID uniquely identifies instructors)
- Authorization
  - Who can access what

Data Manipulation Language (DML)

Language for accessing and updating the data organized by the appropriate data model

There are basically two types of data-manipulation language

- **Procedural DML** -- require a user to specify what data are needed and how to get those data.
- **Declarative DML** -- require a user to specify what data are needed without specifying how to get those data.

SQL Query Language

SQL query language is nonprocedural. A query takes as input several tables (possibly only one) and always returns a single table.

Example to find all instructors in Comp. Sci. dept

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

Application programs generally access databases through one of

- Language extensions to allow embedded SQL
- Application program interface (e.g., ODBC/JDBC) which allow SQL queries to be sent to a database

Database Design

Logical Design – Deciding on the database schema. Database design requires that we find a "good" collection of relation schemas.

Business decision – What attributes should we record?

Computer Science decision – What relation schemas should we have?

Physical Design – Deciding on the physical layout of the database

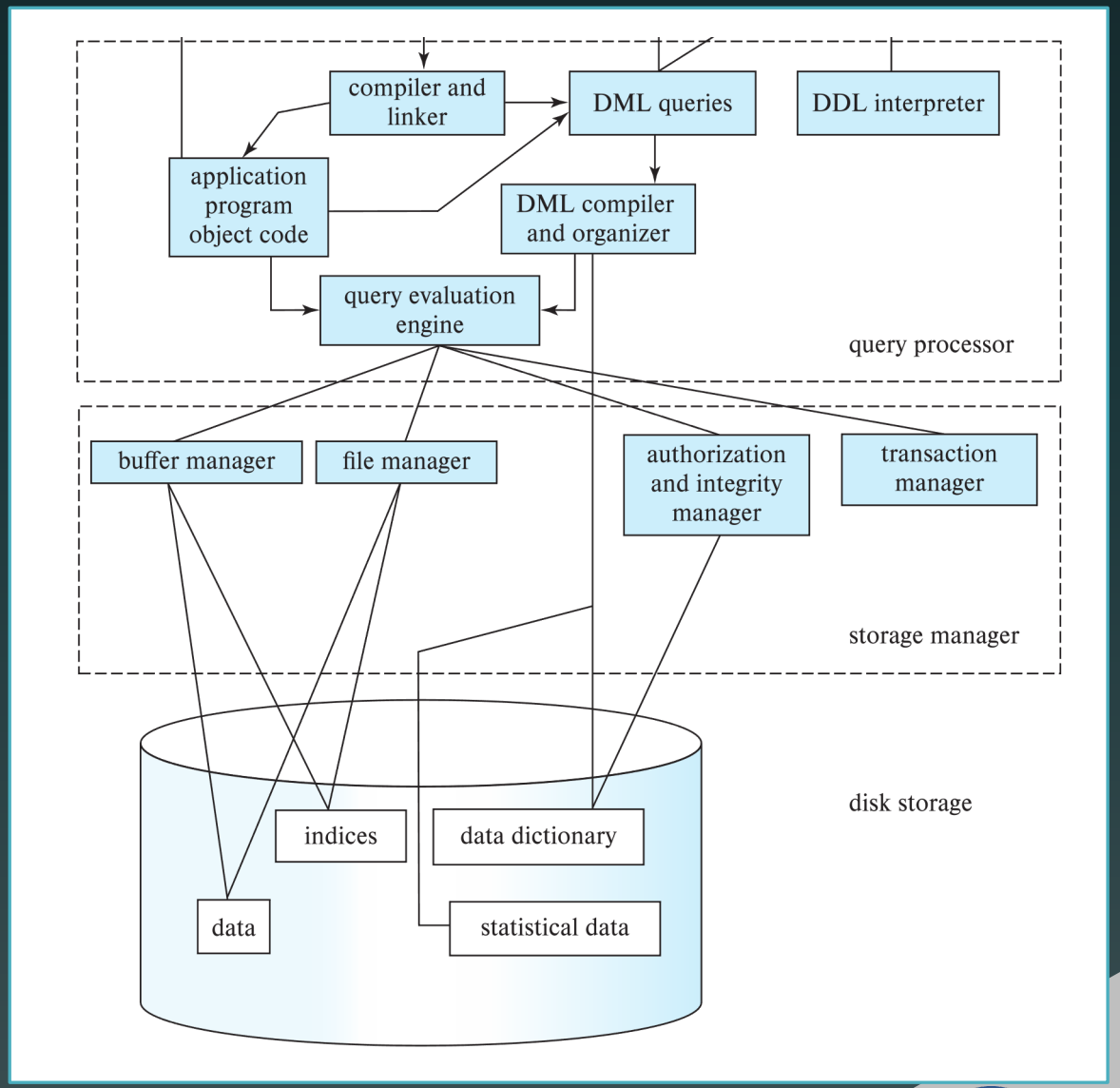
Database Management System

- A database management system is partitioned into modules that deal with each of the responsibilities of the overall system.
- The **functional components** of a database management system can be divided into:

The storage manager	The query processor component	The transaction management component

A Database Management System Architecture

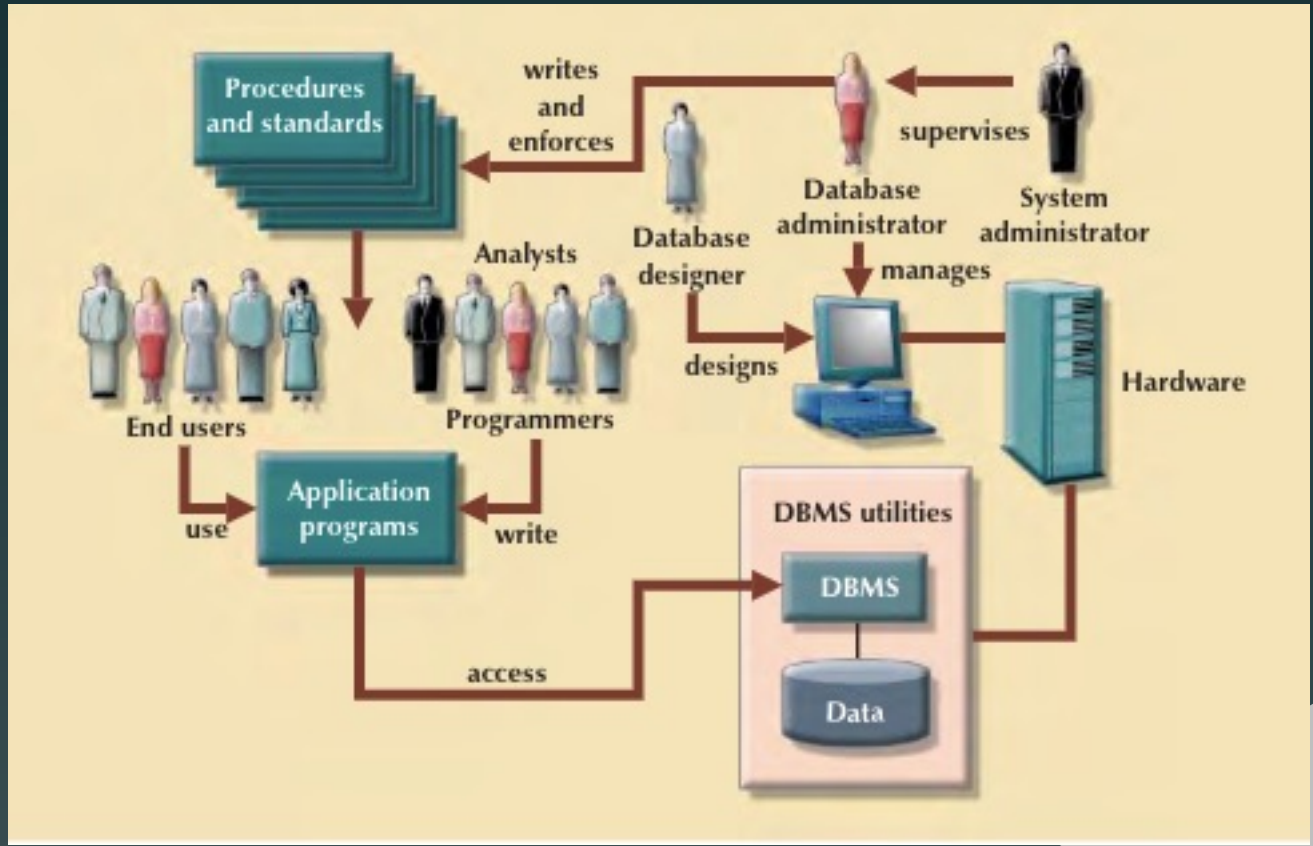
on a centralized server machine



The Database System Environment

Database System refers to an organization of components that define and regulate the collection, storage, management, and use of data within a database environment.

The database system is composed of the five major parts: hardware, software, people, procedures, and data.

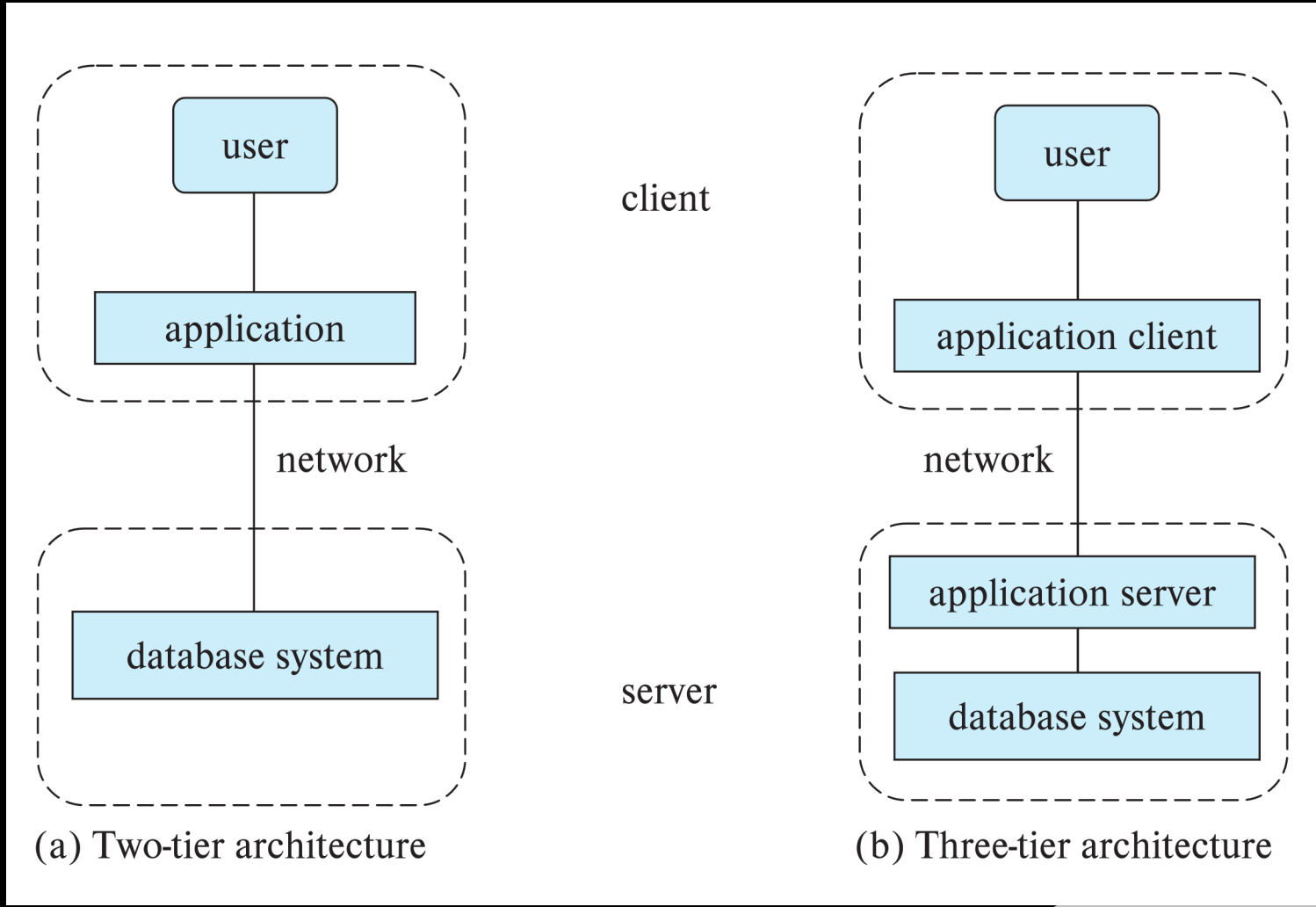


C. Coronel, S. Morris, P. Rob, 'Database System: Design, Implementation, and Management', Ninth Edition, Cengage Learning

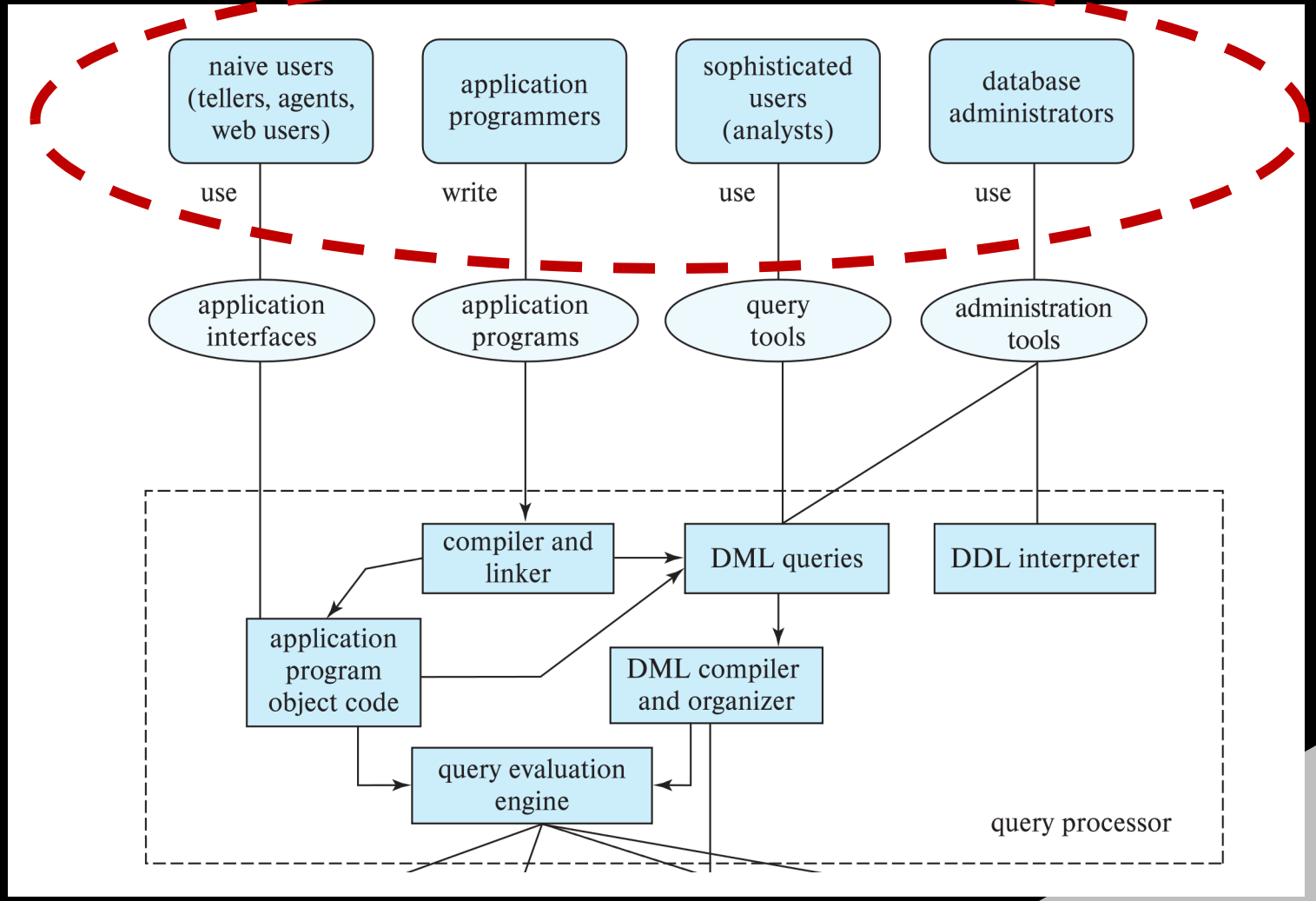
Database Applications Architecture

Two-tier architecture -- the application resides at the client machine, where it invokes database system functionality at the server machine

Three-tier architecture -- the client machine acts as a front end and does not contain any direct database calls.



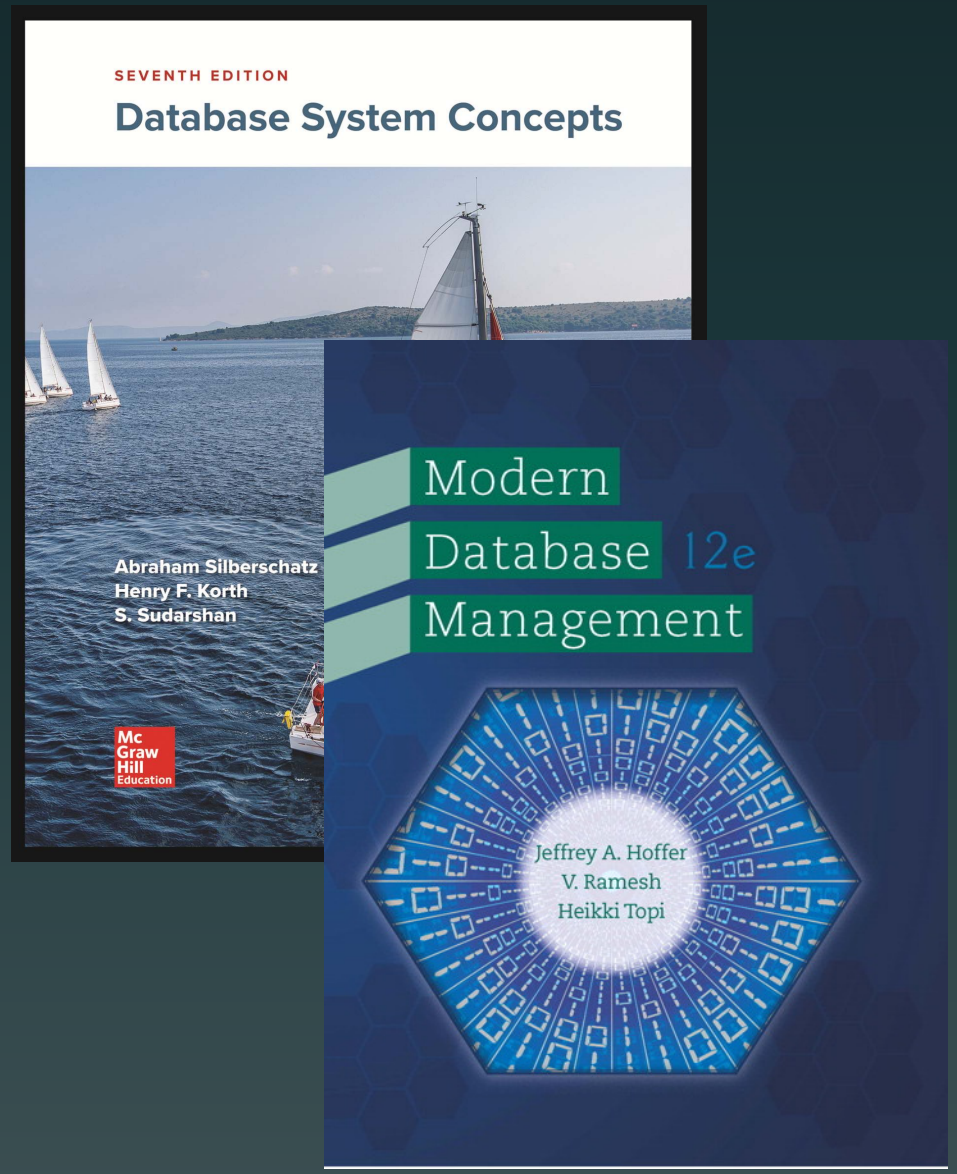
Database Users



IF2240 – Basis Data

Data Model

KNOWLEDGE & SOFTWARE ENGINEERING



References

Abraham Silberschatz, Henry F. Korth, S. Sudarshan : "**Database System Concepts**", 7th Edition

- Chapter 1: Introduction

Jeffrey A. Hoffer, Mary B. Prescott, Heikki Topi : "**Modern Database Management**", 12th Edition

- Chapter 1: The Database Environment and Development Process

Data Modeling (1/2)

- A technique aimed at optimizing the way that information is stored and used within an organization
  - Begins with the **identification of the main data groups**, continues by **defining the detailed content** of each of these groups.
  - Result: **structured definitions** for all of the information that is stored and used within a given system.
- **An essential precursor** to analysis, design, maintenance & documentation and improving the performance of an existing system.

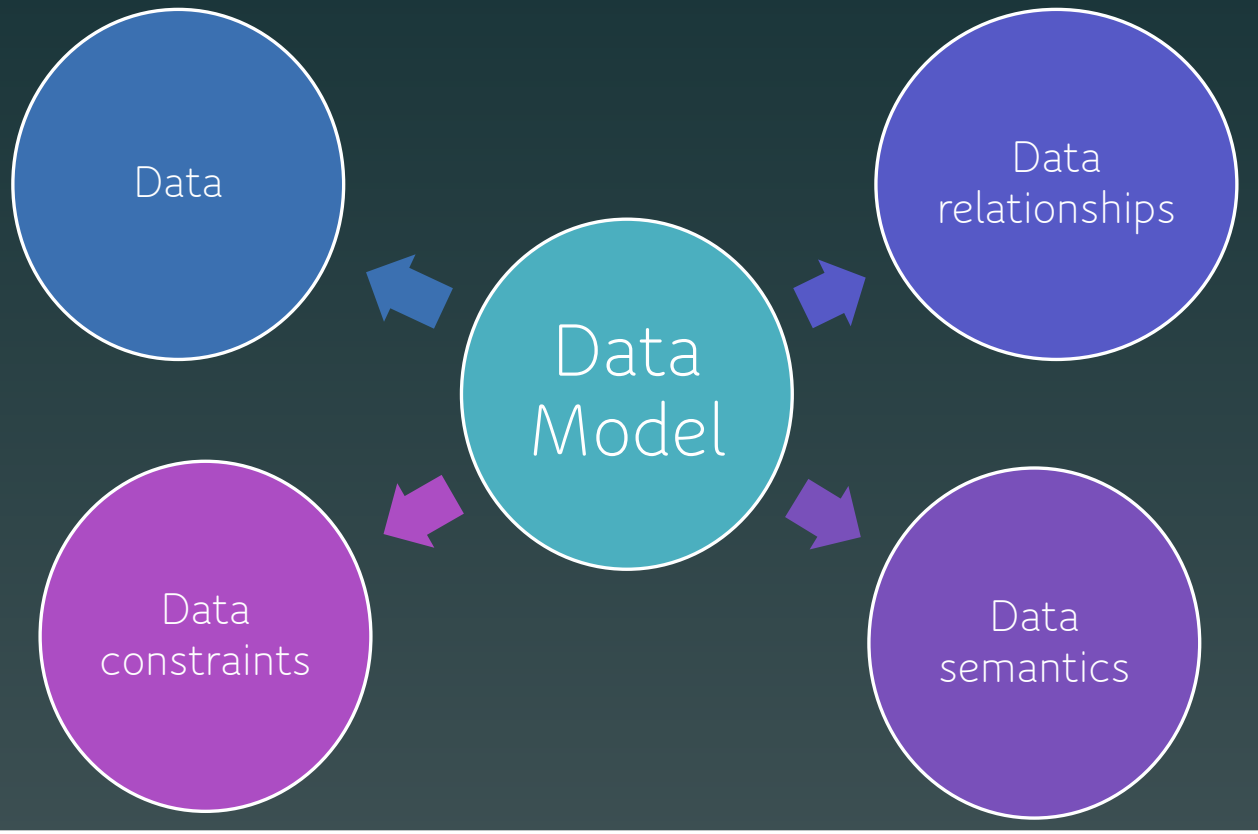
Data Modeling (2/2)

There are three different types of data models produced while progressing **from requirements to the actual database**.

- Conceptual data model: a set of technology independent specifications about the data and is used to discuss initial requirements with the business stakeholders.
- Logical data model: the structures of the data that can be implemented in databases.
- Physical data model: that organizes the data into tables, and accounts for access, performance and storage details.

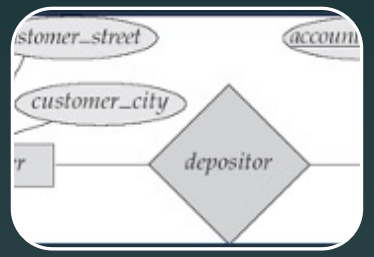
Data Model (1/2)

A collection of tools for describing:

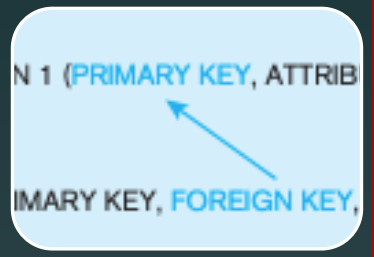


Data Model (2/2)

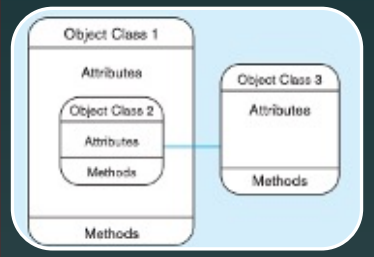
Types of data model:



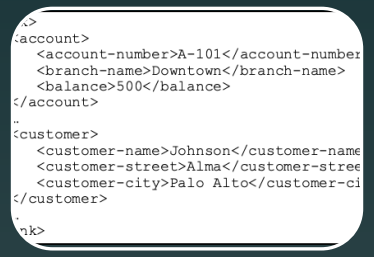
Entity-relationship model



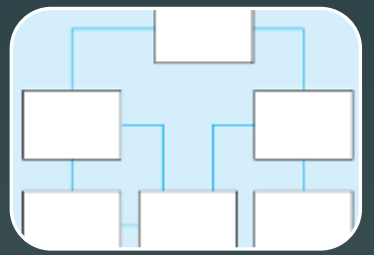
Relational model



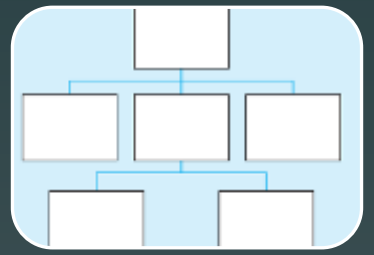
Object-based data model (object-oriented and object relational)



Semi-structured data model (XML)



Network model

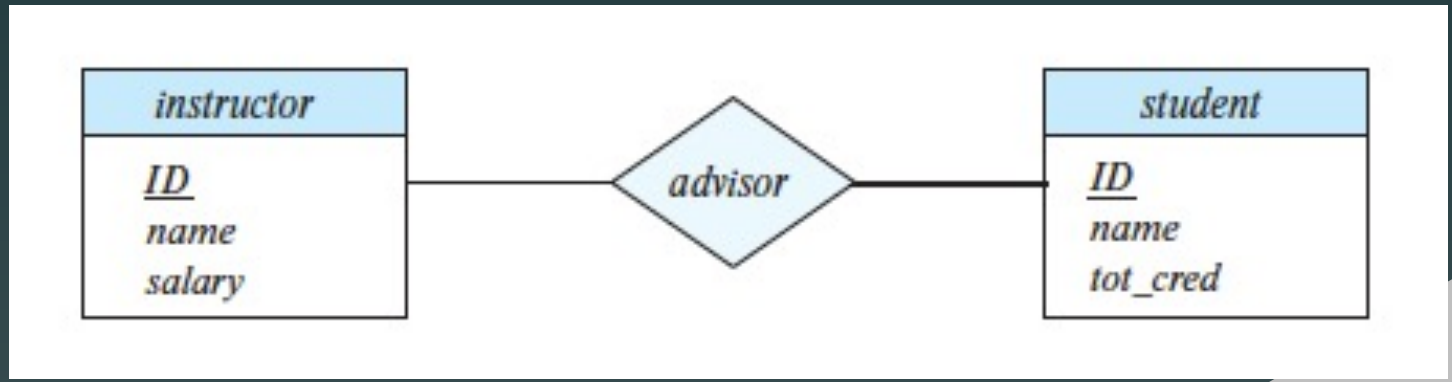


Hierarchical model

Older data models

Entity Relationship Model

- Widely used for database design
  - Database design in E-R model usually converted into design in Relational Model
- Models an enterprise as a collection of *entities* and *relationships*
  - **Entity**: a "thing" or "object" in the enterprise that is distinguishable from other objects
  - **Relationship**: an association among several entities
- Represented diagrammatically by an *entity-relationship diagram* (ERD)



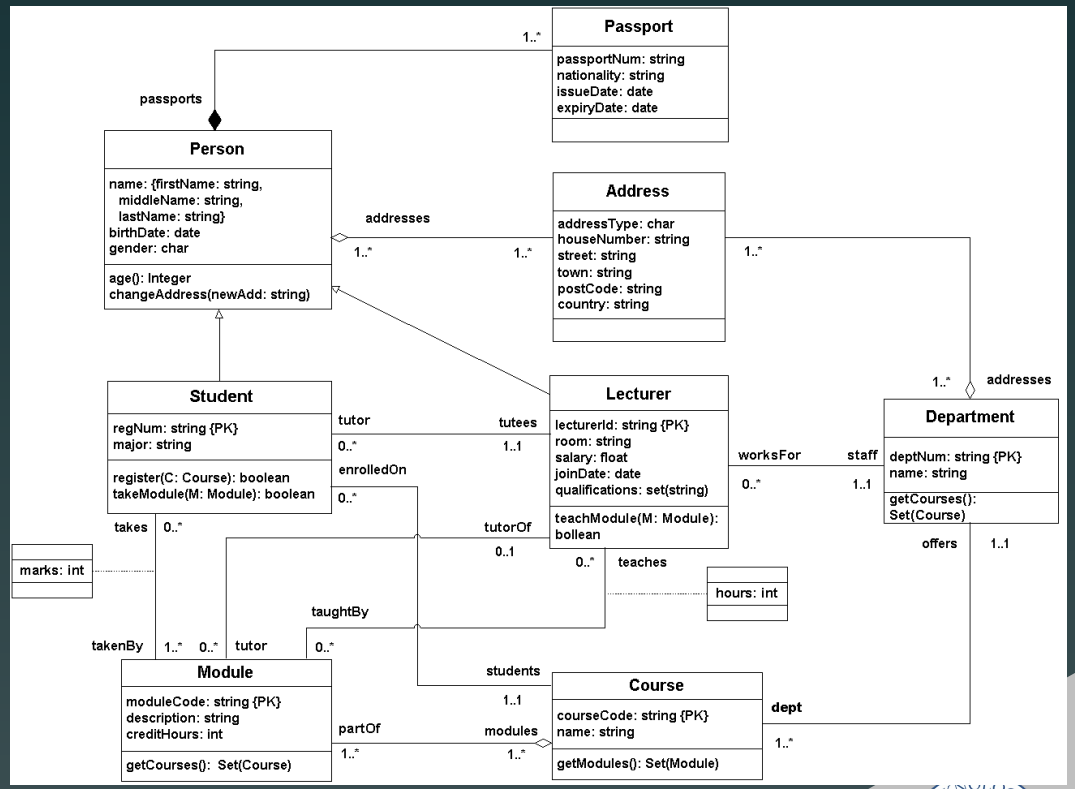
Relational Model

- Consist of collection of tables to represent data and the relationship among those data
- Example of tabular data in the relational model

customer_id	*customer_name*	*customer_street*	*customer_city*	*account_number*
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-101
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-201
677-89-9011	Hayes	3 Main St.	Harrison	A-102
182-73-6091	Turner	123 Putnam St.	Stamford	A-305
321-12-3123	Jones	100 Main St.	Harrison	A-217
336-66-9999	Lindsay	175 Park Ave.	Pittsfield	A-222
019-28-3746	Smith	72 North St.	Rye	A-201

Object-Oriented Data Model

- Adaptation of the object-oriented programming paradigm (e.g. Smalltalk, C++) to database systems
- The *object-oriented paradigm* is based on *encapsulating* code and data related to an object into single unit



Object Structure

```
class Person {
   /* variable */
      Name        name;
      date        birthDate;
      string      address;
      char        gender;
   /* messages */
      int         age();
      int         changeAddress(string newAdd);
      Name        getName();
      date        getBirthDate();
      string      getAddress();
      char        getGender();

};

string get-address() {
   return address;
}
```

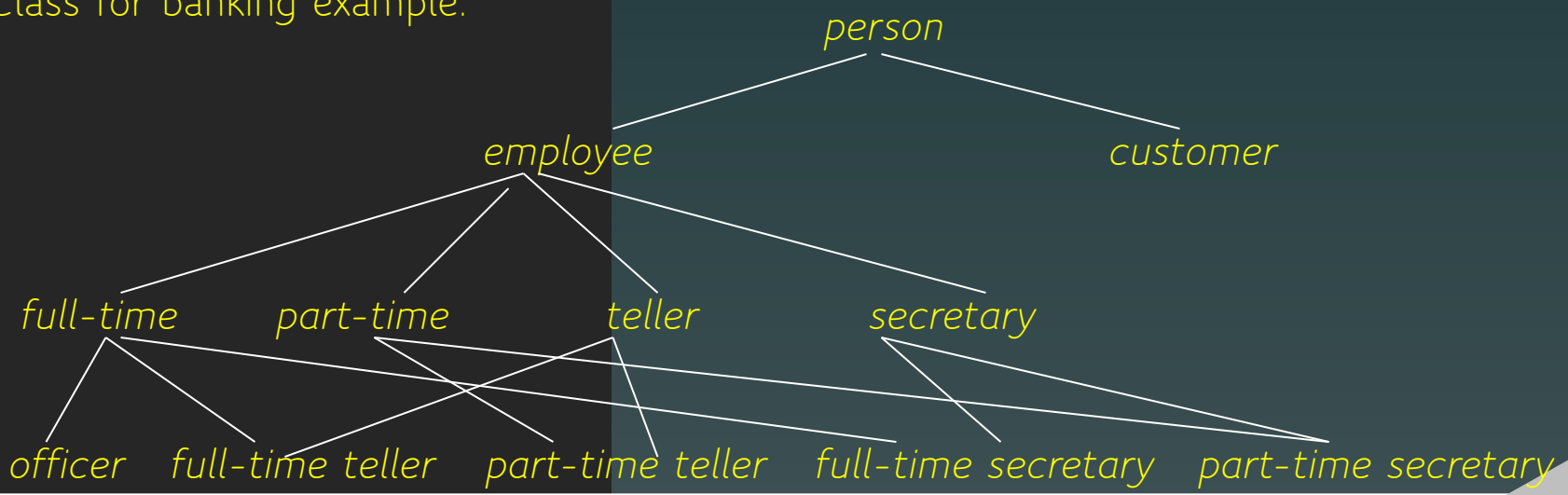
An object has associated with it:

- A set of variables that contain the data for the object. The value of each variable is itself an object.
- A set of messages to which the object responds; each message may have zero, one, or more *parameters*.
- A set of methods, each of which is a body of code to implement a message; a method returns a value as the *response* to the message

Object-Relational Data Model

- Extend the relational data model by including object orientation and a richer type system including collection types
  - Object orientation provides inheritance with subtypes and sub tables
  - Collection types include nested relations, sets, multisets, and arrays
- Preserve relational foundations, in particular the declarative access to data, while extending modeling power

Class for banking example:



Semi Structured Data Model (ex: XML)

- The ability to specify new tags, and to create nested tag structures made XML a great way to exchange **data**, not just documents
- XML has become the basis for all new generation data interchange formats.
- A wide variety of tools is available for parsing, browsing and querying XML documents/data

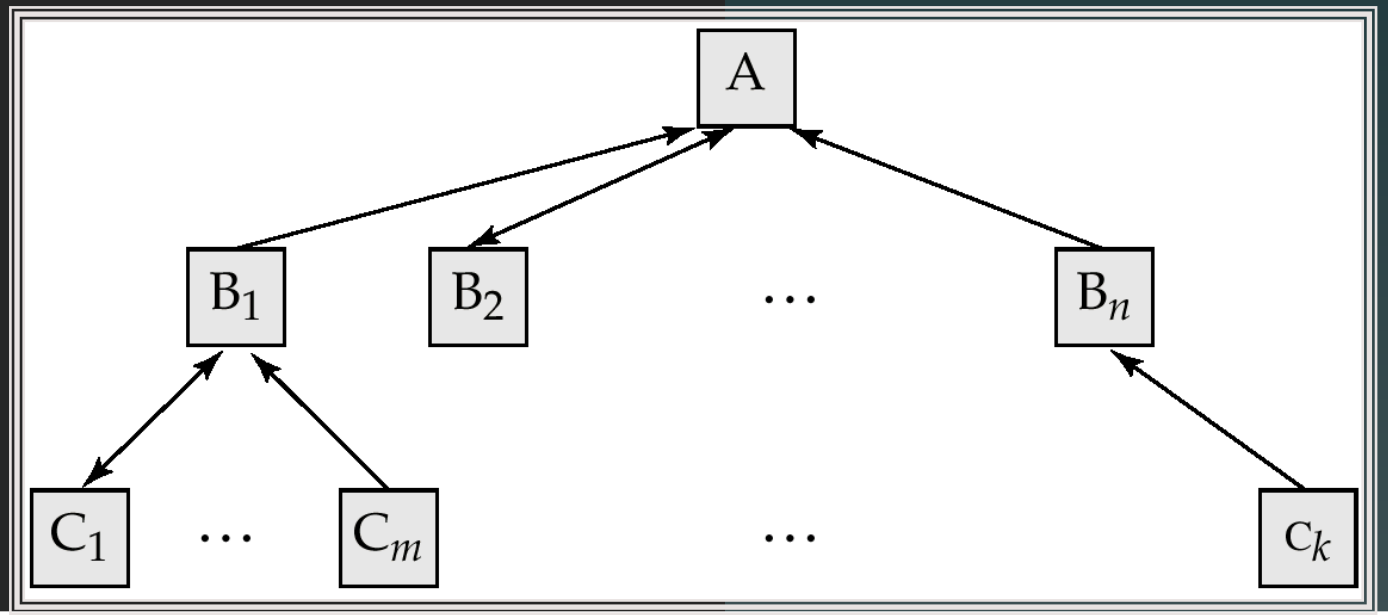
- Characteristics:
  - organized in semantic entities
  - similar entities are grouped together
  - entities in same group may not have same attributes
  - order of attributes not necessarily important
  - not all attributes may be required
  - size of same attributes in a group may differ
  - type of same attributes in a group may differ

→Self-describing, irregular data, no a priori structure

```
<bank>
   <account>
      <account-number>A-101</account-number>
      <branch-name>Downtown</branch-name>
      <balance>500</balance>
   </account>
   …
   <customer>
      <customer-name>Johnson</customer-name>
      <customer-street>Alma</customer-street>
      <customer-city>Palo Alto</customer-city>
   </customer>
   …
</bank>
```

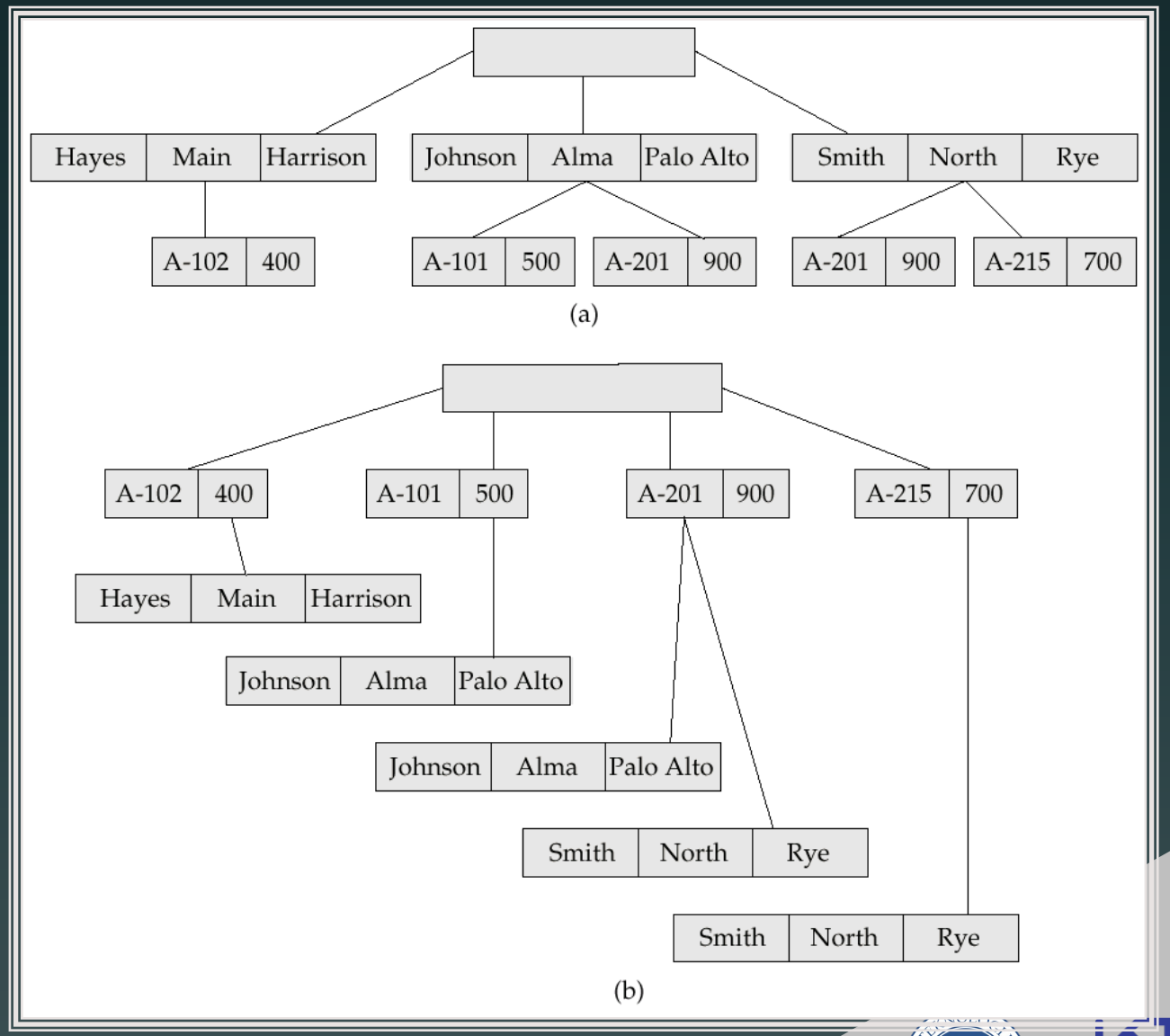
Hierarchical Model (1/2)

- World's leading mainframe hierarchical database system in the 1970s and early 1980s
- Database schema is represented as a collection of tree-structure diagrams
  - *Single* instance of a database tree
  - The root of this tree is a dummy node
  - The children of that node are actual instances of the appropriate record type
- The schema for a hierarchical database consists of
  - *boxes*, which correspond to record types
  - *lines*, which correspond to links
- Record types are organized in the form of a *rooted tree*



A parent *may* have an arrow pointing to a child, but a child *must* have an arrow pointing to its parent

Hierarchical Model (2/2)



Network Model (1/2)

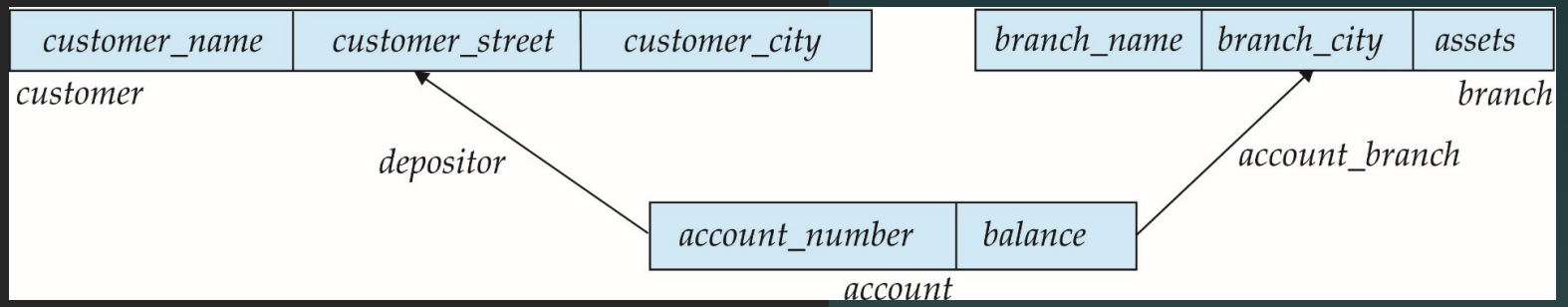
- Data are represented by collections of *records*.
  - similar to an entity in the E-R model
  - Records and their fields are represented as *record type*
- Relationships among data are represented by *links*
  - similar to a restricted (binary) form of an E-R relationship
  - restrictions on links depend on whether the relationship is many-many, many-to-one, or one-to-one.

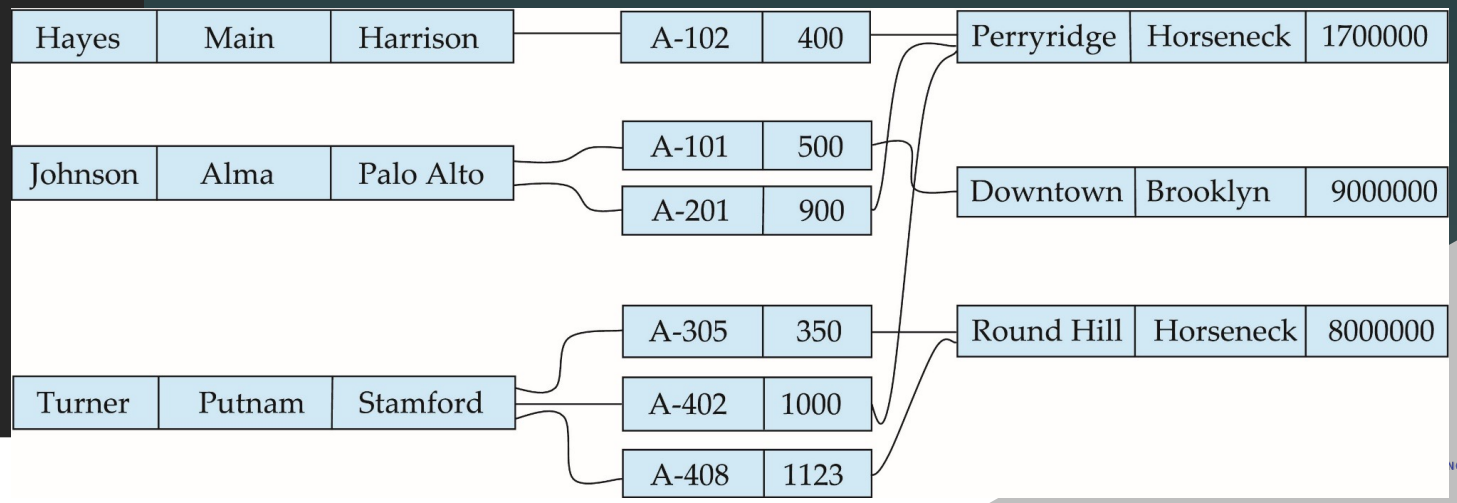
```
type customer = record
     customer-name: string;
     customer-street: string;
     customer-city: string;
end;
```

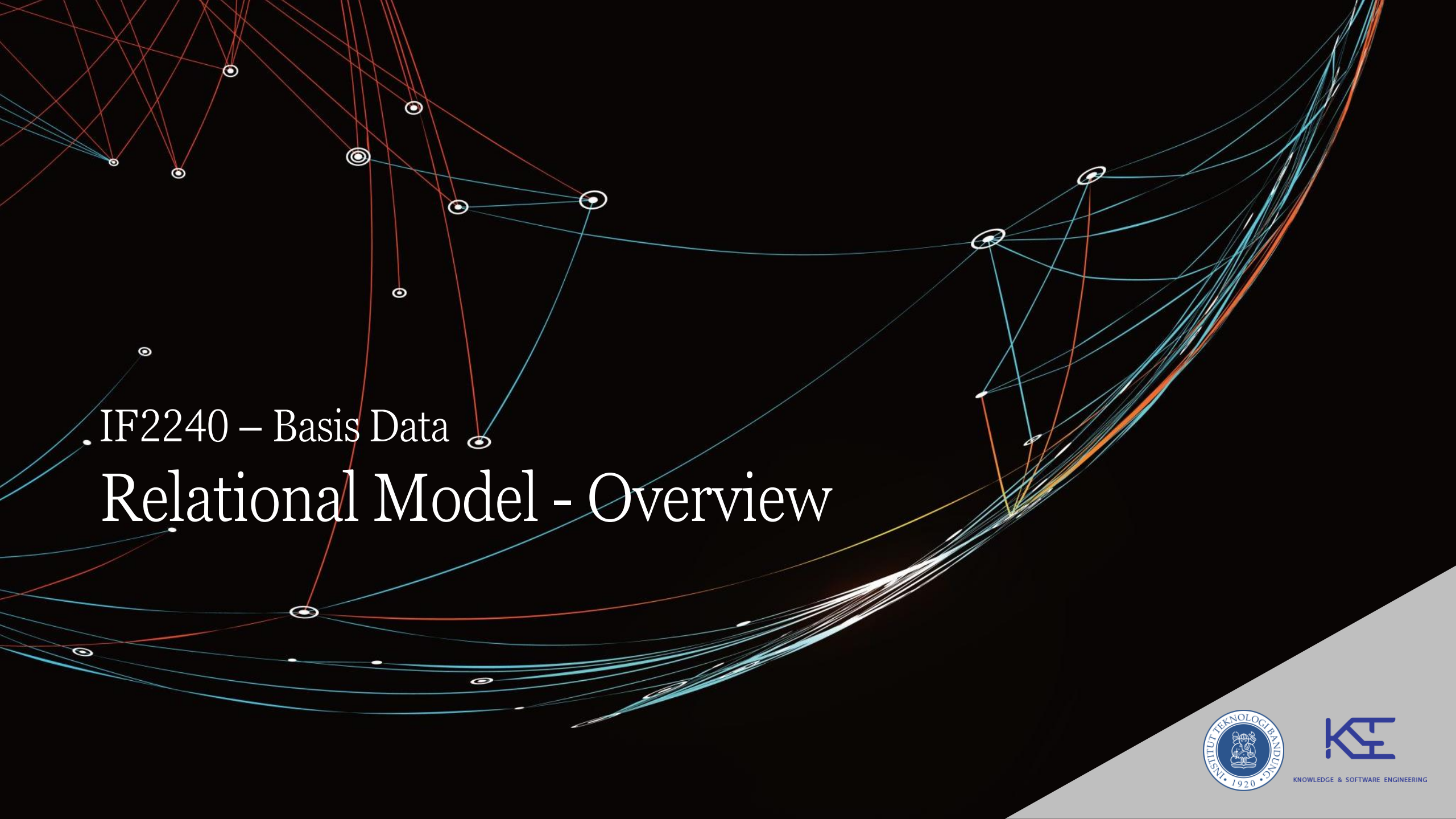
```
type account = record
     account-number: integer;
     balance: integer;
end;
```

Network Model (2/2)

- Graph like structure
  - Child may have multiple parent
- A data-structure diagram consists of two basic components:
  - **Boxes**, which correspond to record types
  - **Lines**, which correspond to links
- Specifies the overall logical structure of the database



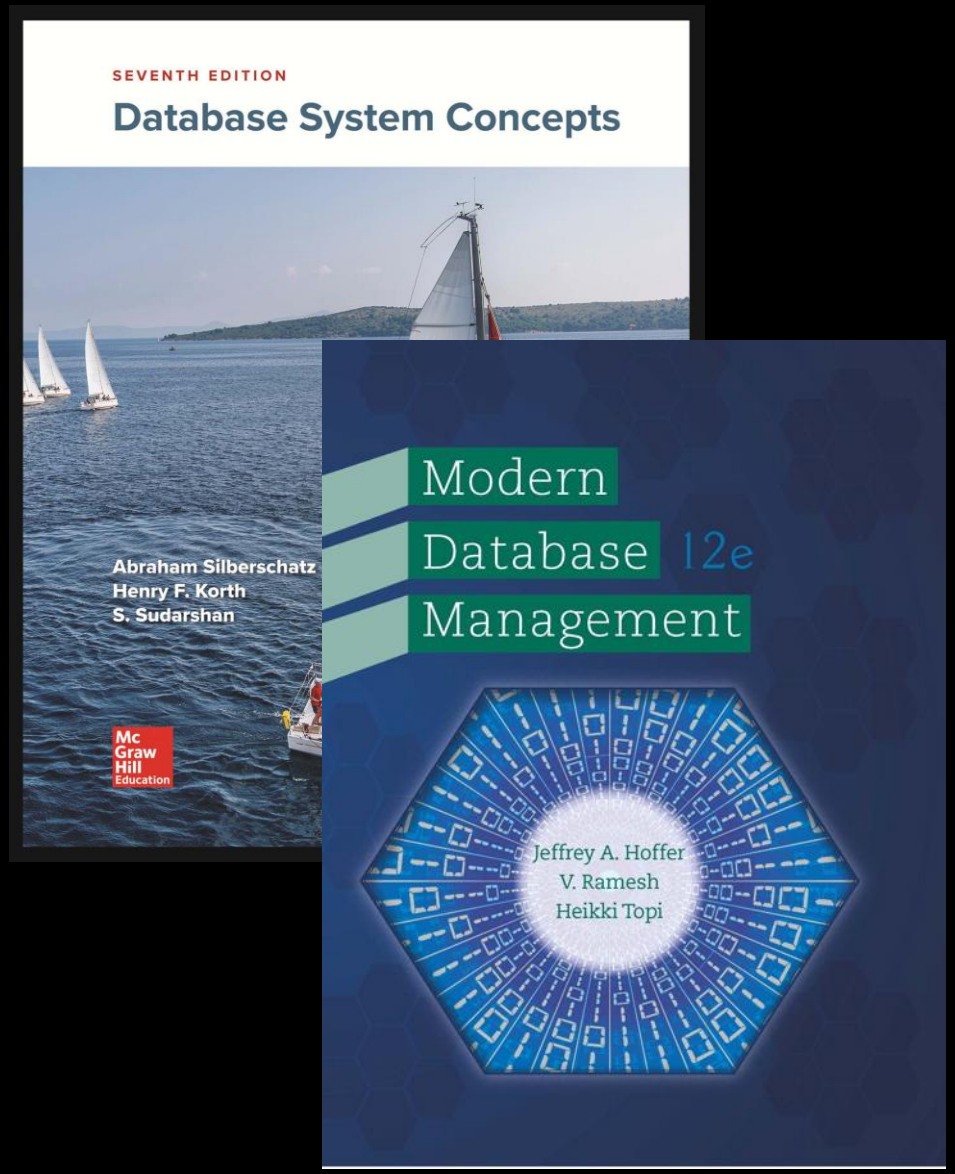




IF2240 – Basis Data

Relational Model - Overview

KNOWLEDGE & SOFTWARE ENGINEERING



References

Abraham Silberschatz, Henry F. Korth, S. Sudarshan : "**Database System Concepts**", 7th Edition

- Chapter 2: Introduction to the Relational Model

Jeffrey A. Hoffer, Mary B. Prescott, Heikki Topi : "**Modern Database Management**", 12th Edition

- Chapter 4: Logical Database Design and the Relational Model

Relational Model Concept

- A Relation is a mathematical concept based on the **ideas of sets**
- The model was first proposed by Dr. E.F. Codd of IBM Research in 1970 in the following paper:
  - "A Relational Model for Large Shared Data Banks," Communications of the ACM, June 1970
- The above paper caused a major revolution in the field of database management and earned Dr. Codd the coveted ACM Turing Award
- Why study this? **Most widely used model**
  - Today, RDBMSs have become the dominant technology for database management, and there are literally hundreds of RDBMS products for computers ranging from smartphones and personal computers to mainframes

Relation

- Definition: A relation is a named, two-dimensional table of data
- Table consists of **rows (records)** and **columns (attribute or field)**
- Requirements for a table to qualify as a relation:
  - It must have a unique name
  - Every attribute value must be atomic (not multivalued, not composite)
  - Every row must be unique (can't have two rows with exactly the same values for all their fields)
  - Attributes (columns) in tables must have unique names
  - The order of the columns may be unordered
  - The order of the rows may be unordered

NOTE: all *relations* are in ***1st Normal form***

Example : Relation Instructor

attributes
(or columns)

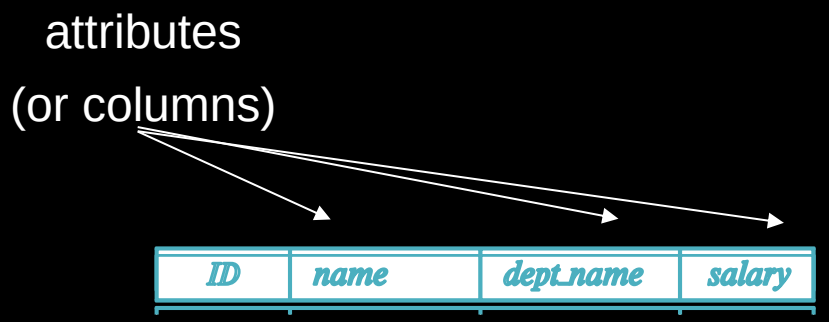
ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821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

tuples
(or rows)

KNOWLEDGE & SOFTWARE ENGINEERING

Attributes

- Each attribute of a relation has a name
- The set of allowed values for each attribute is called the **domain** of the attribute
- Attribute values are (normally) required to be **atomic**; that is, indivisible
  - E.g. the value of an attribute can be an account number, but cannot be a set of account numbers
- The special value *null* is a member of every domain
- The null value causes complications in the definition of many operations
  - We shall ignore the effect of null values in our main presentation and consider their effect later
- Meanings for NULL values
  - Value unknown
  - Value exists but is not available
  - Attribute does not apply to this tuple (also known as value undefined)
- IMPORTANT: **NULL ≠ NULL**



Relation Schema & Instance

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

  Example:

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

- The current values a relation are specified by a table
- An element ***t*** of relation ***r*** is called a *tuple* and is represented by a *row* in a table

Database Schema

Database schema -- is the logical structure of the database.

Database instance -- is a snapshot of the data in the database at a given instant in time.

Example:

- schema: *instructor* (*ID, name, dept_name, salary*)
- Instance:

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

Why Split Information Across Relations?

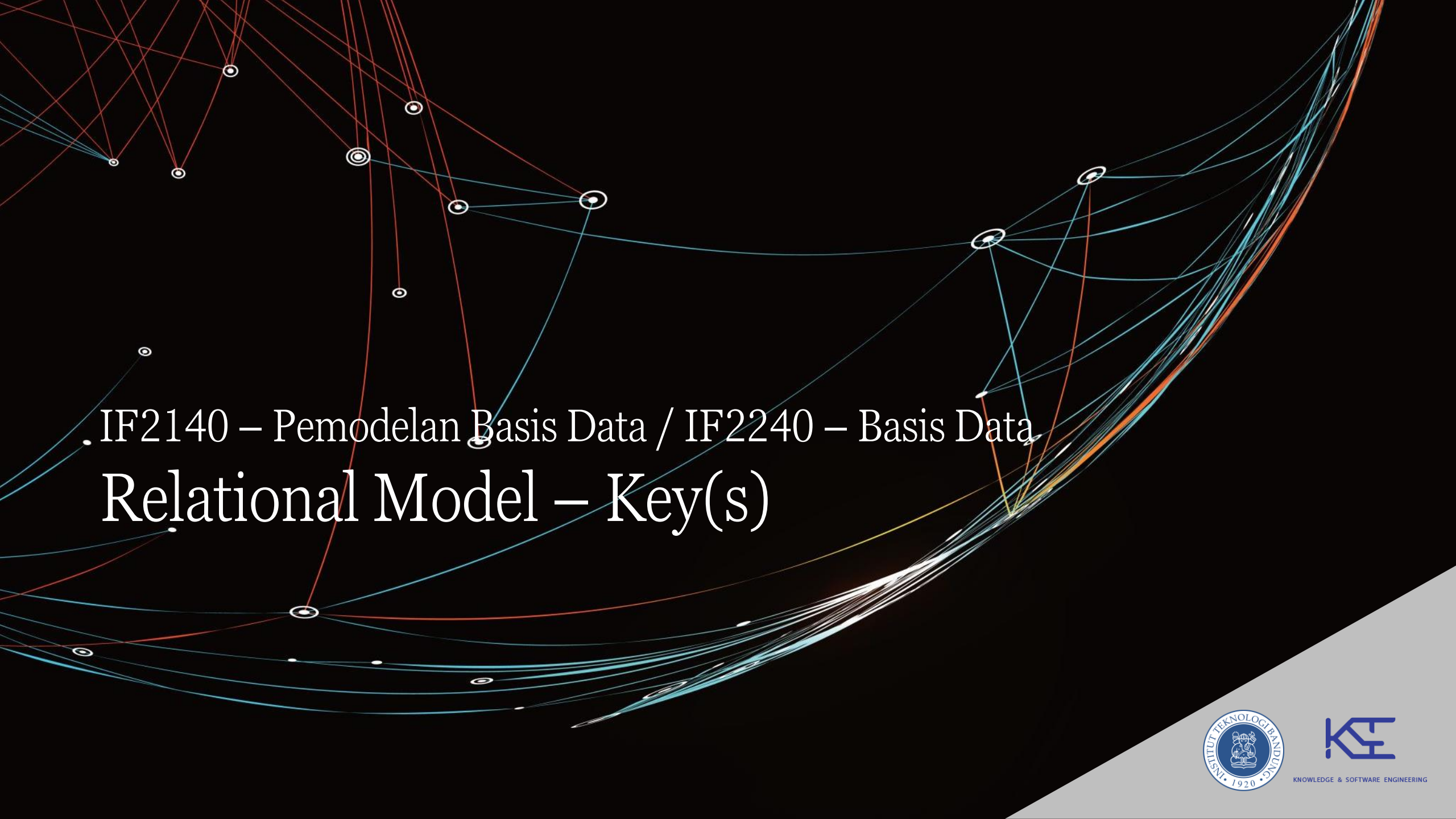
Storing all information as a single relation such as

bank(*account_number, balance, customer_name*, ..)

results in :

- repetition of information
  - e.g.,if two customers own an account (What gets repeated?)
- the need for null values
  - e.g., to represent a customer without an account

Normalization theory (we'll come back later) deals with how to design relational schemas



IF2140 – Pemodelan Basis Data / IF2240 – Basis Data

Relational Model – Key(s)

KNOWLEDGE & SOFTWARE ENGINEERING

Keys

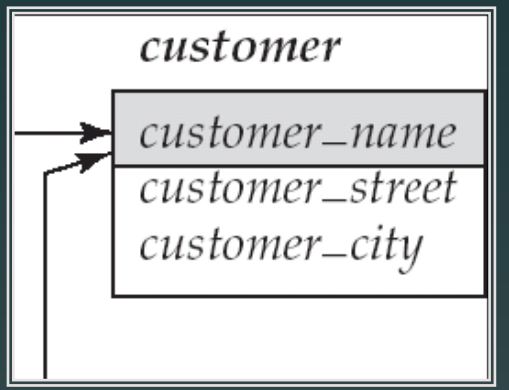
- Superkey
- Candidate Keys
- Primary Key
- Foreign Key

Superkeys

Let $K \subseteq R$

K is a **superkey** of *R* if values for *K* are sufficient to identify a unique tuple of each possible relation *r(R)*

- by "possible *r* " we mean a relation *r* that could exist in the enterprise we are modeling.
- Example: {*customer_name, customer_street*} and
  {*customer_name*}
  are both superkeys of *Customer*, if no two customers can possibly have the same name
  - In real life, an attribute such as *customer_id* would be used instead of *customer_name* to uniquely identify customers, but we omit it to keep our examples small, and instead assume customer names are unique.



Candidate and Primary Key

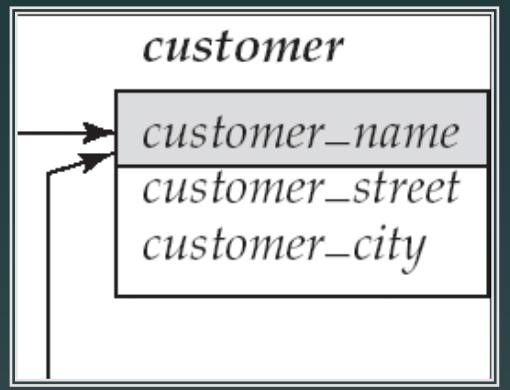
K is a **candidate key** if *K* is minimal
Example: {*customer_name*} is a candidate key for *Customer*, since it is a superkey and no subset of it is a superkey.

Primary key: a candidate key chosen as the principal means of identifying tuples within a relation

- Should choose an attribute whose value never, or very rarely, changes.
- E.g. email address is unique, but may change

Keys can be ***simple*** (a single field) or ***composite*** (more than one field)

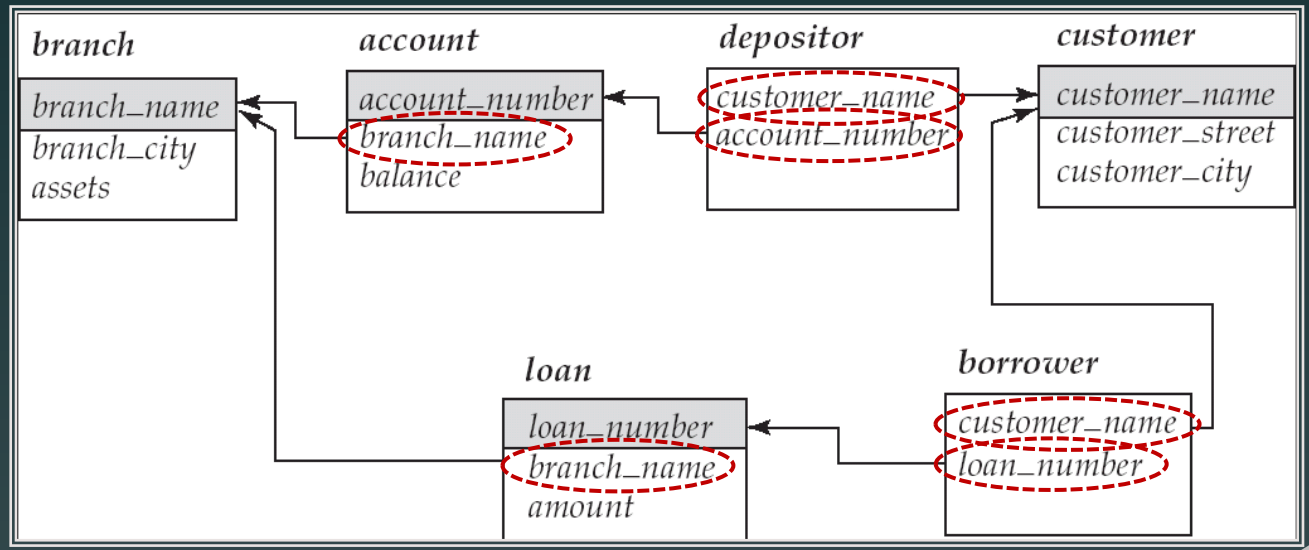
Keys usually are used as indexes to speed up the response to user queries



Foreign Keys

A relation schema may have an attribute that corresponds to the primary key of another relation. The attribute is called a **foreign key**.

- E.g. *customer_name* and *account_number* attributes of *depositor* are foreign keys to *customer* and *account* respectively.
- Only values occurring in the primary key attribute of the **referenced relation** may occur in the foreign key attribute of the **referencing relation**.

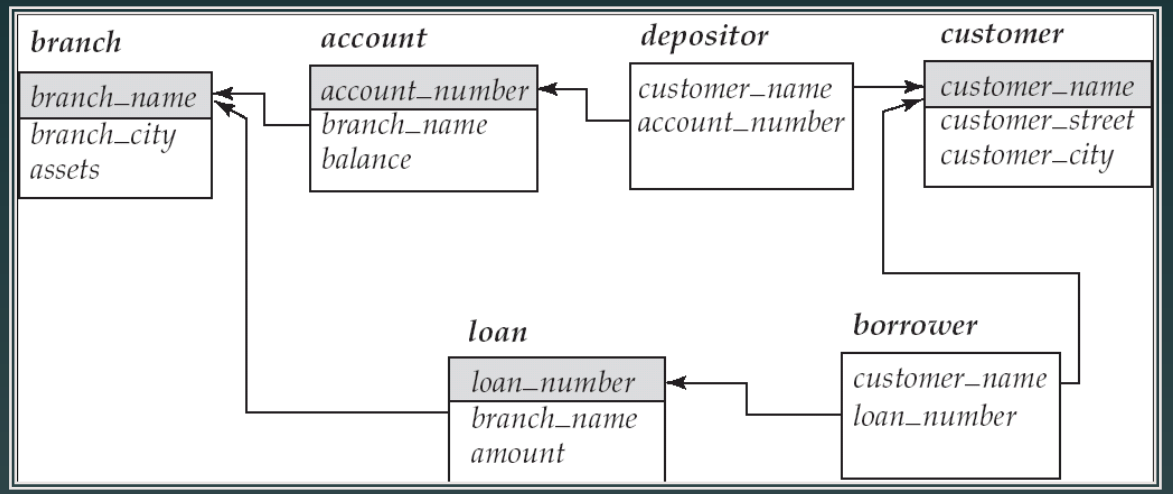


Primary Keys & Foreign Keys (Example)

branch = (branch_name, branch_city, assets)
account = (account_number, branch_name, balance)
customer = (customer_name, customer_street, customer_city
loan = (loan_number, branch_name, amount)
depositor = (customer_name, account_number)
borrower = (customer_name, loan_number)

FK's: account(branch_name) → branch(branch_name)
loan(branch_name) → branch(branch_name)
depositor(customer_name) → customer(customer_name)
depositor(account_number → account(account_number)
borrower(customer_name) → customer(customer_name)
borrower(loan_number) → loan(loan_number)

Note:
underlined : primary key

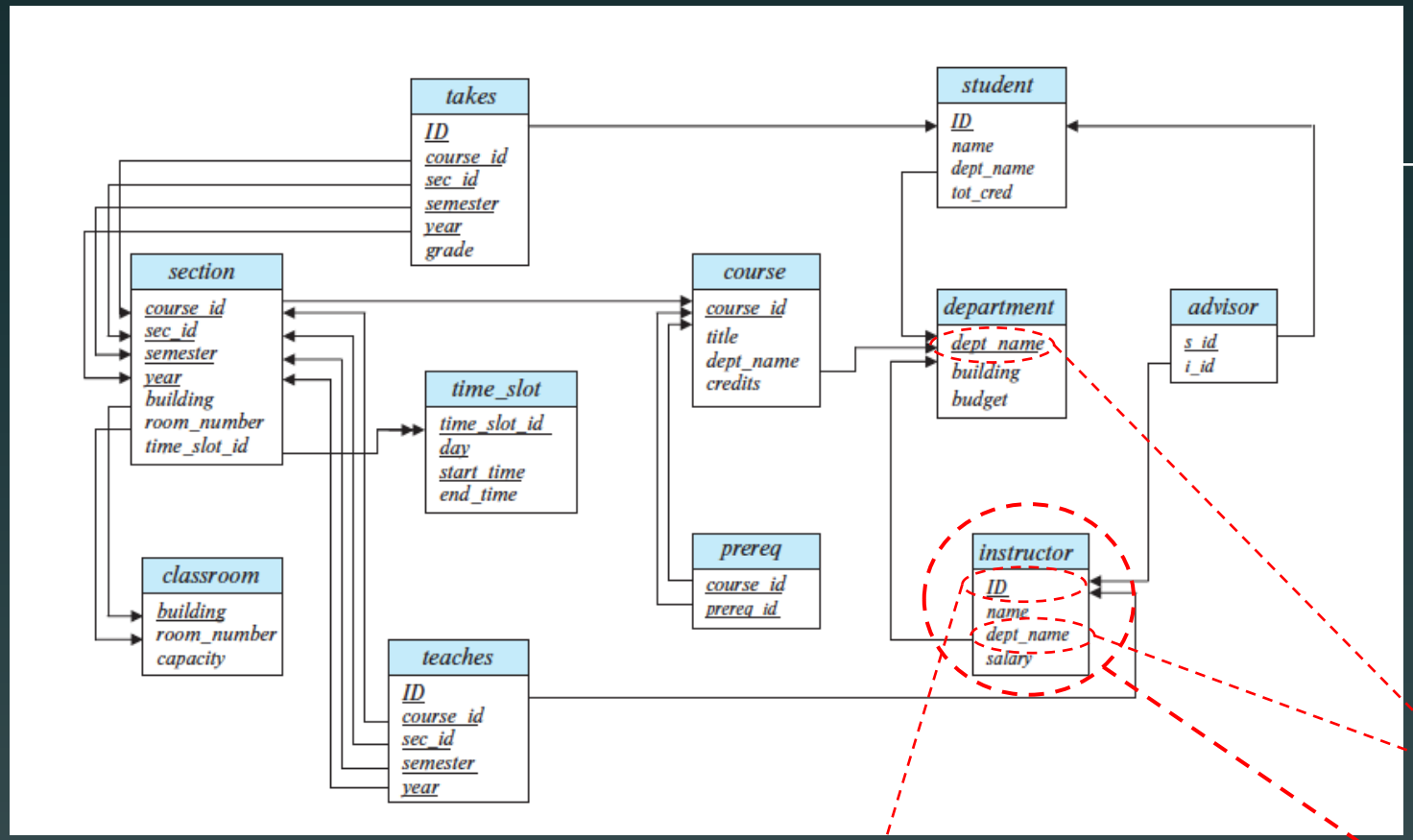


IF2140 – Pemodelan Basis Data / IF2240 – Basis Data

Relational Model – Schema Diagram & Integrity Constraint

KNOWLEDGE & SOFTWARE ENGINEERING

Schema Diagrams



Foreign-key constraints appear as arrows from the foreign-key attributes of the referencing relation to the primary key of the referenced relation

Primary-key shown underlined

Relation appears as box
Name at the top in the blue, attributes listed inside the box

Integrity Constraint

Domain Constraints	Entity Integrity	Referential Integrity
• All of the values that appear in a column of a relation must be from the same domain.	• The entity integrity rule is designed to ensure that every relation has a primary key and that the data values for that primary key are all valid.	• Rule that maintains consistency among the rows of two relations

Domain Constraint

A domain definition usually consists of the following components: domain name, meaning, data type, size (or length), and allowable values or allowable range (if applicable).

TABLE 4-1 Domain Definitions for INVOICE Attributes

Attribute	Domain Name	Description	Domain
CustomerID	Customer IDs	Set of all possible customer IDs	character: size 5
CustomerName	Customer Names	Set of all possible customer names	character: size 25
CustomerAddress	Customer Addresses	Set of all possible customer addresses	character: size 30
CustomerCity	Cities	Set of all possible cities	character: size 20
CustomerState	States	Set of all possible states	character: size 2
CustomerPostalCode	Postal Codes	Set of all possible postal zip codes	character: size 10
OrderID	Order IDs	Set of all possible order IDs	character: size 5
OrderDate	Order Dates	Set of all possible order dates	date: format mm/dd/yy
ProductID	Product IDs	Set of all possible product IDs	character: size 5
ProductDescription	Product Descriptions	Set of all possible product descriptions	character: size 25
ProductFinish	Product Finishes	Set of all possible product finishes	character: size 15
ProductStandardPrice	Unit Prices	Set of all possible unit prices	monetary: 6 digits
ProductLineID	Product Line IDs	Set of all possible product line IDs	integer: 3 digits
OrderedQuantity	Quantities	Set of all possible ordered quantities	integer: 3 digits

Entity Integrity

A rule that states that **no primary key attribute** (or component of a primary key attribute) may be null.

If we choose this one as a primary key, so there's no null CustomerID

TABLE 4-1 Domain Definitions for INVOICE Attributes

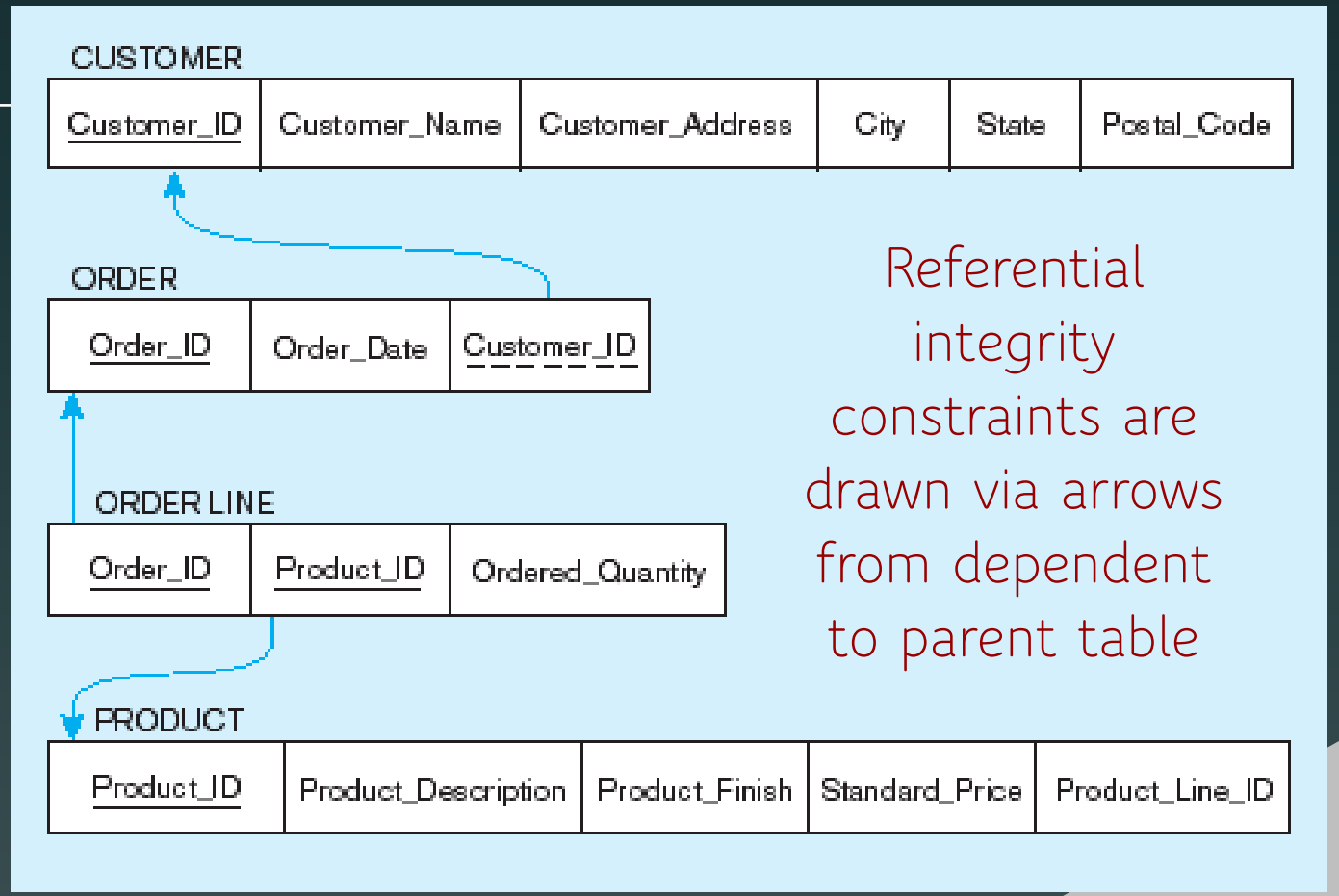
Attribute	Domain Name	Description	Domain
CustomerID	Customer IDs	Set of all possible customer IDs	character: size 5
CustomerName	Customer Names	Set of all possible customer names	character: size 25
CustomerAddress	Customer Addresses	Set of all possible customer addresses	character: size 30
CustomerCity	Cities	Set of all possible cities	character: size 20
CustomerState	States	Set of all possible states	character: size 2
CustomerPostalCode	Postal Codes	Set of all possible postal zip codes	character: size 10
OrderID	Order IDs	Set of all possible order IDs	character: size 5
OrderDate	Order Dates	Set of all possible order dates	date: format mm/dd/yy
ProductID	Product IDs	Set of all possible product IDs	character: size 5
ProductDescription	Product Descriptions	Set of all possible product descriptions	character: size 25
ProductFinish	Product Finishes	Set of all possible product finishes	character: size 15
ProductStandardPrice	Unit Prices	Set of all possible unit prices	monetary: 6 digits
ProductLineID	Product Line IDs	Set of all possible product line IDs	integer: 3 digits
OrderedQuantity	Quantities	Set of all possible ordered quantities	integer: 3 digits

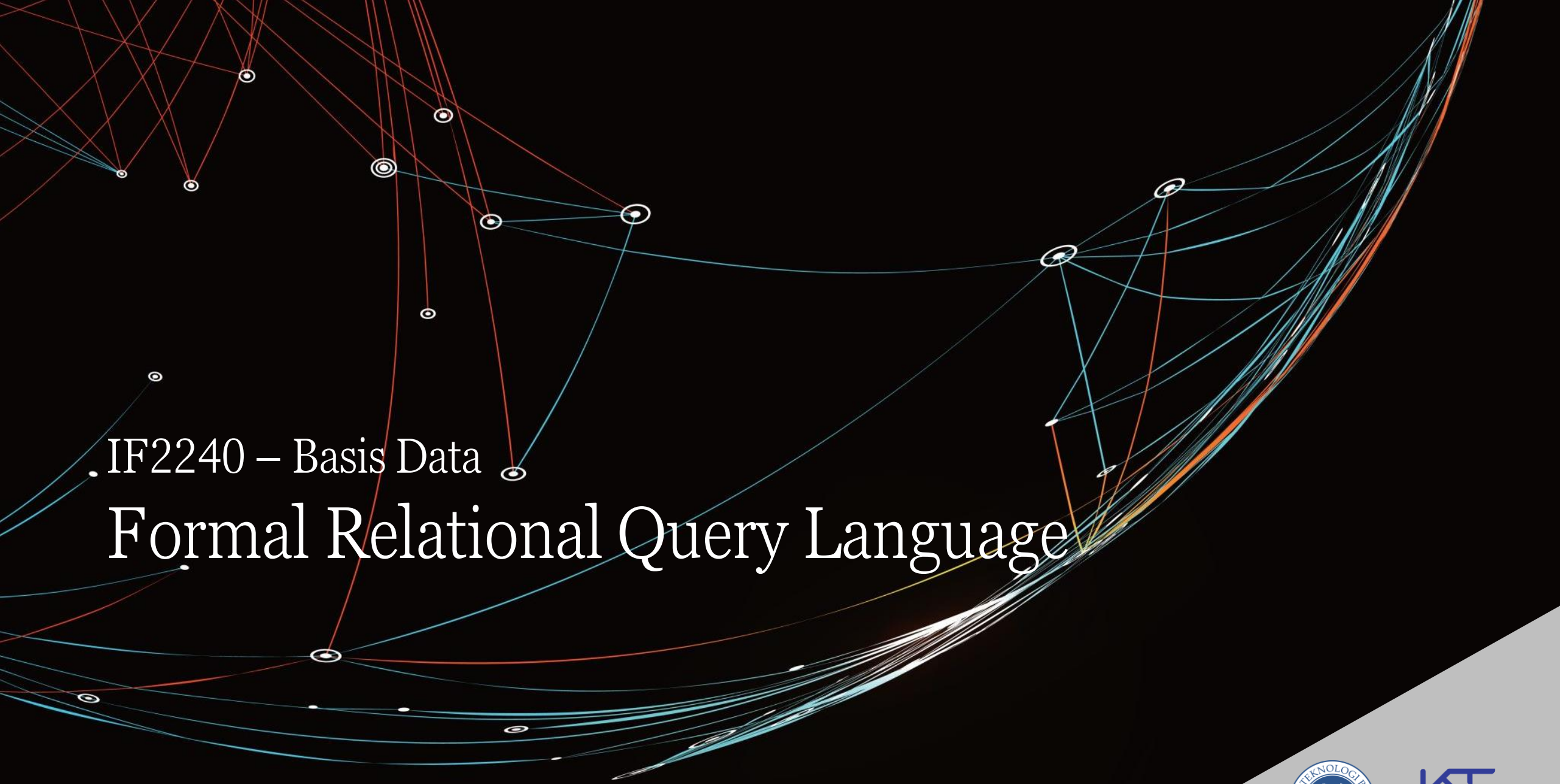
Referential Integrity

Rule states that any foreign key value MUST match a primary key value in the referenced relation **(Or the foreign key can be null)**.

For example: Delete Rules

- **Restrict**–don't allow delete of "parent" side if related rows exist in "dependent" side
- **Cascade**–automatically delete "dependent" side rows that correspond with the "parent" side row to be deleted
- **Set-to-Null**–set the foreign key in the dependent side to null if deleting from the parent side (not allowed for weak entities)

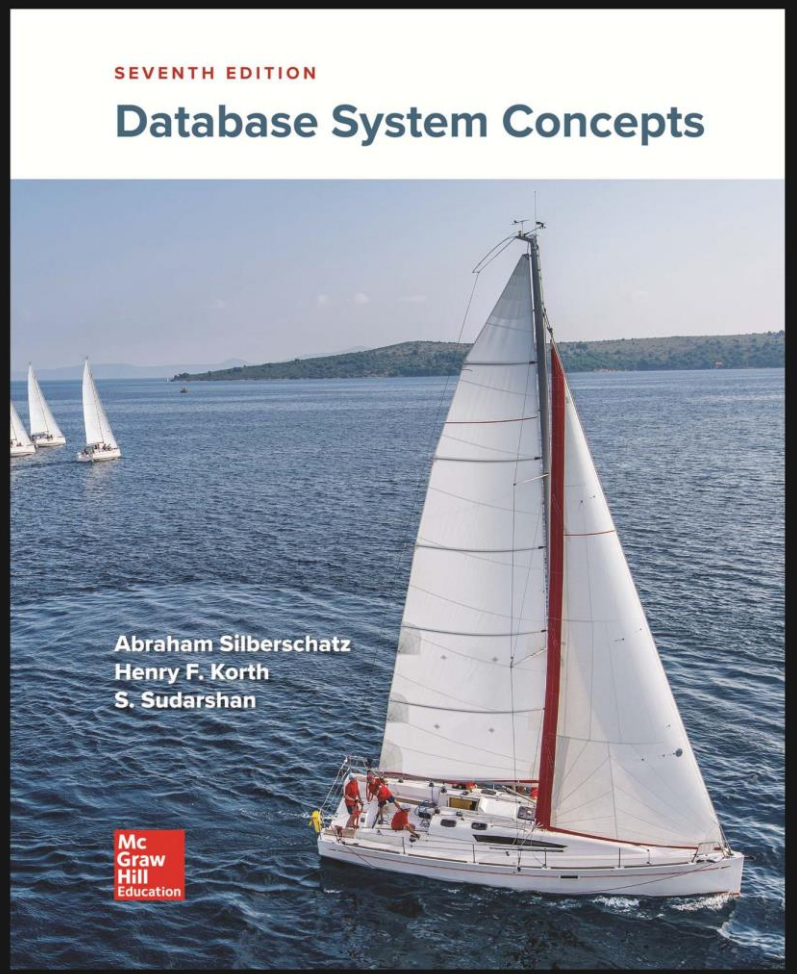




IF2240 – Basis Data

Formal Relational Query Language

KNOWLEDGE & SOFTWARE ENGINEERING



Sumber

Silberschatz, Korth, Sudarshan: "Database System Concepts", 7th Edition

- Chapter 2: Relational Model
  - Section 2.5. Relational Query Languages
  - Section 2.6. The Relational Algebra
  - Page 47 – 58
- Chapter 27: Formal-Relational Query Languages (*online chapter*)
  - Section 27.1. The Tuple Relational Calculus
  - Section 27.2. The Domain Relational Calculus

Objective

- Demonstrate use of the relational algebra operations from mathematical set theory (union, intersection, difference, and Cartesian product) and the relational algebra operations developed specifically for relational databases (select (restrict), project, join, and division)
- Demonstrate queries in the relational algebra
- Demonstrate queries in the domain relational calculus
- Demonstrate queries in the tuple relational calculus

Query Languages

Categories of languages

1

Functional/Procedural

- **Relational algebra**

HOW?

2

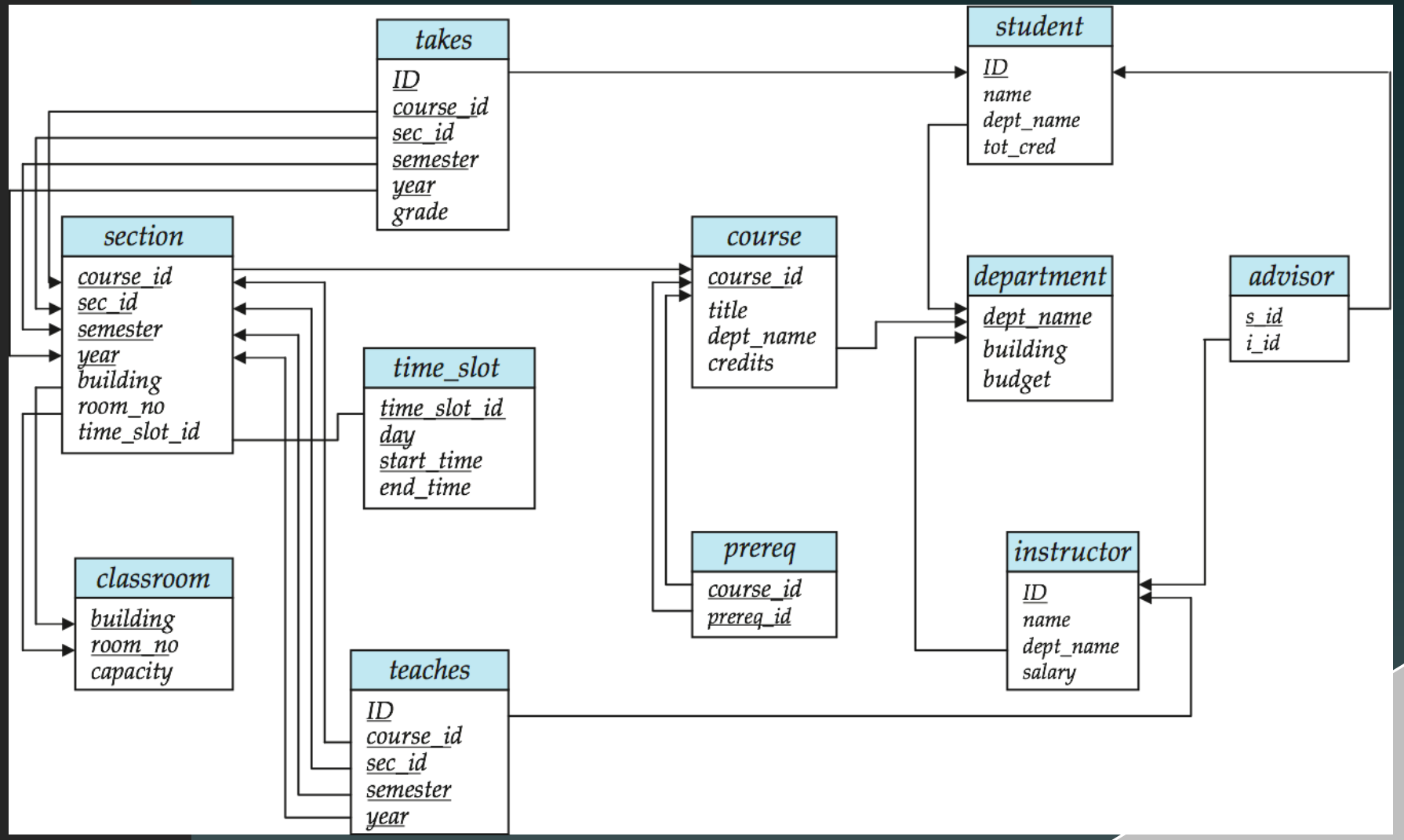
Non-procedural, or declarative

- Tuple relational calculus
- Domain relational calculus

WHAT?

Pure languages form underlying basis of query languages that people use

Schema Diagram for University Database



KNOWLEDGE & SOFTWARE ENGINEERING

RELATIONAL ALGEBRA

Relational Algebra

Procedural language

Six basic operators

- select: σ
- project: Π
- union: $\cup$
- set difference: $-$
- cartessian product: x
- rename: ρ

The operators take one or two relations as inputs and produce a new relation as a result.

- Additional Operators
  - intersection
  - natural join
  - assignment
  - outer join
  - division
- Extended Operators
  - generalized projection
  - aggregation

Basic Operators

Select Operation

Notation

$$\sigma_p(r)$$

p = selection predicate

Defined as

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

- Where p is a formula in propositional calculus: terms connected by $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**)
- Term = <attribute> *op* <attribute> OR <constant>

Select Operation – Example 1

INSTRUCTOR (ID, NAME, DEPT_NAME, SALARY)

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

$\sigma_{DEPT\_NAME= \text{“PHYSICS”}}(INSTRUCTOR)$

ID	name	dept_name	salary
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

Select Operation – Example 2

RELATION R

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

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

A	B	C	D
α	α	1	7
β	β	23	10

Project Operation

Notation

$$\Pi_{A1, A2, \ldots, Ak} (r)$$

where A_1, A_2, … are attribute names and r is a relation name.

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

Project Operation – Example 1

INSTRUCTOR (ID, NAME, DEPT_NAME, SALARY)

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

$\Pi_{ID,\ NAME,\ DEPT\_NAME}\ (INSTRUCTOR)$

ID	name	dept_name
22222	Einstein	Physics
12121	Wu	Finance
32343	El Said	History
45565	Katz	Comp. Sci.
98345	Kim	Elec. Eng.
76766	Crick	Biology
10101	Srinivasan	Comp. Sci.
58583	Califieri	History
83821	Brandt	Comp. Sci.
15151	Mozart	Music
33456	Gold	Physics
76543	Singh	Finance

Project Operation – Example 2

RELATION *R*:

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

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

A	C
α	1
α	1
β	1
β	2

A	C
α	1
β	1
β	2

Composition of Relational Operations

The result of a relational-algebra operation is a relation and therefore some of relational-algebra operations can be composed together into a **relational-algebra expression**.

Consider the query -- Find the names of all instructors in the Physics department.

$$\Pi_{name}(\sigma_{dept\_name = \text{“Physics”}}(instructor))$$

Union Operation

- Notation

$$r \cup s$$

- Defined as

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

- Requirement
  1. *r, s* must have the *same* *arity*
  2. The attribute domains must be compatible

Union Operation – Example 1

RELATIONS

R

A	B
α	1
α	2
β	1

S

A	B
α	2
β	3

R ∪ S:

A	B
α	1
α	2
β	1
β	3

Union Operation – Example 2

section
course_id
sec_id
semester
year
building
room_no
time_slot_id

Find all courses taught in the Fall 2017 semester, or in the Spring 2018 semester, or in both

$$\Pi_{course\_id}\,(\sigma_{semester=\text{“Fall”}\ \wedge\ year=2017}\,(section))\ \cup$$
$$\Pi_{course\_id}\,(\sigma_{semester=\text{“Spring”}\ \wedge\ year=2018}\,(section))$$

Set Difference Operation

- Notation

$$r - s$$

- Defined as

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

- Requirement

  Set differences must be taken between *compatible* relations

Set Difference Operation – Example 1

RELATIONS

R

A	B
α	1
α	2
β	1

S

A	B
α	2
β	3

R - S:

A	B
α	1
β	1

Set Difference Operation – Example 2

section
course_id
sec_id
semester
year
building
room_no
time_slot_id

Find all courses taught in the Fall 2017 semester, but not in the Spring 2018 semester

$$\Pi_{course\_id}\,(\sigma_{semester=\text{“Fall”}\ \wedge\ year=2017}\,(section)) \;-\; \Pi_{course\_id}\,(\sigma_{semester=\text{“Spring”}\ \wedge\ year=2018}\,(section))$$

Cartesian-Product Operation

Notation

$$r \times s$$

Defined as

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

Requirement

- Attributes of r(R) and s(S) are disjoint
- If attributes of *r(R)* and *s(S)* are not disjoint, then:
  renaming OR attach its source relation

instructor.ID

Cartesian-Product Operation – Example 1

RELATIONS

R

A	B
α	1
β	2

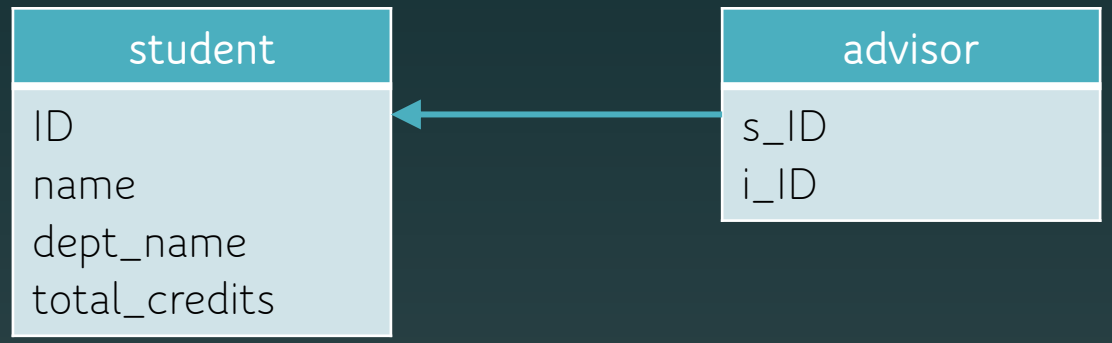
S

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

R X S:

A	B	C	D	E
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

Cartesian-Product Operation – Example 2



Find the names of all students whose advisor's id is 22222

$\Pi_{name}\ (\sigma_{i\_ID=22222}\ (\sigma_{ID=\ s\_ID}\ (student\ x\ advisor)))$

Rename Operation

- Usage
  - To name the results of relational-algebra expressions.
  - To refer to a relation by more than one name
- Notation (relation)

$$\rho_{X}(E)$$

returns the result of expression E under the name X

- Notation (attributes)

$$\rho_{X(A1, A2, \ldots, An)}(E)$$

returns the result of expression E under the name X, and with the attributes renamed to $A1, A2, \ldots, An$.

Rename Operation - Example

instructor
ID
name
dept_name
salary

Find the ID and name of those instructors who earn more than the instructor whose ID is 12121

$$\Pi_{z.ID,\ z.name} \left(\sigma_{z.salary > w.salary} \left(\rho_{z} (instructor) \times \left(\sigma_{ID=12121} (\rho_{w} (instructor)) \right) \right) \right)$$

Formal Definition

A basic expression in the relational algebra consists of either one of the following:

- A relation in the database
- A constant relation

Let E_1 and E_2 be relational-algebra expressions; the following are all relational-algebra expressions:

- $E_1 \cup E_2$
- $E_1 - E_2$
- $E_1 \times E_2$
- $\sigma_P (E_1)$, P is a predicate on attributes in E_1
- $\prod_S (E_1)$, S is a list consisting of some of the attributes in E_1
- $\rho_x (E_1)$, x is the new name for the result of E_1

Example Queries

Find all instructors with salary over $80000

Find the names of all instructors in Comp. Sci. dept with salary > 80000

Find the names of all person in the university

schema

Example Queries

Find the title of all courses offered in Fall 2017 semester.

Find the title of all courses offered in Fall 2017 semester that do not have any prerequsites.

schema

Example Queries

Find the names of all instructors who are located at Archway Bldg.

schema

Example Queries

Find the names of all instructors in the Physics department, along with the *course_id* of all courses they have taught

Example Query

Find the largest salary in the university

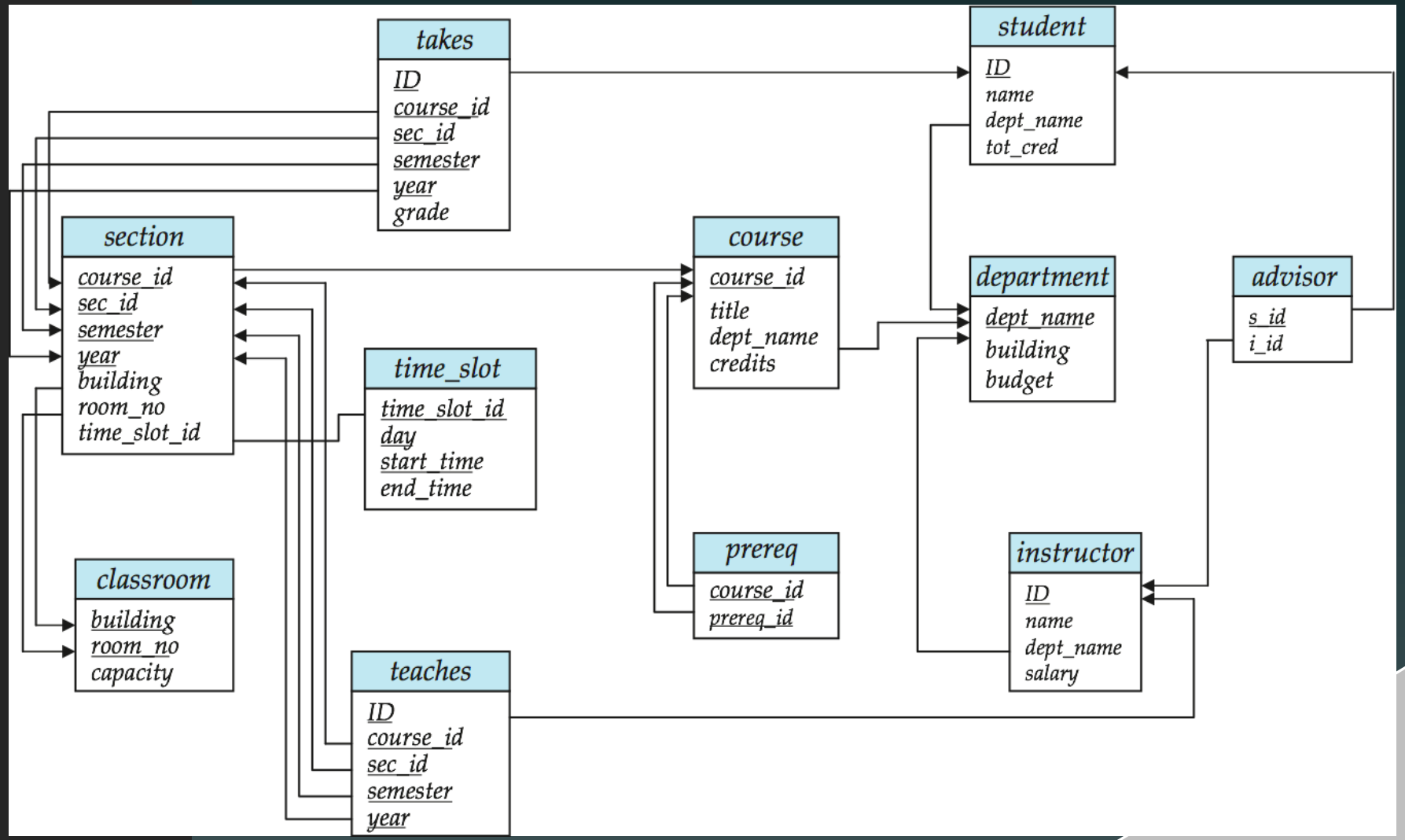
schema

IF2240 – Basis Data

Formal Relational Query Language (2)

KNOWLEDGE & SOFTWARE ENGINEERING

Schema Diagram for University Database



Relational Algebra

Procedural language

Six basic operators
- select: σ
- project: Π
- union: $\cup$
- set difference: $-$
- cartessian product: $\times$
- rename: ρ

The operators take one or two relations as inputs and produce a new relation as a result.

- Additional Operators
  - Intersection: $\cap$
  - natural join: $\bowtie$
  - assignment: $\leftarrow$
  - outer join: $⟗$
  - division: $\div$
- Extended Operators
  - generalized projection
  - aggregation

Additional Operators

Additional Operations

We define additional operations that do not add any power to the relational algebra, but that simplify common queries.

- Set intersection
- Natural join
- Assignment
- Outer join
- Division

Set-Intersection Operation

Notation

$$r \cap s$$

Defined as

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

Requirement

Set intersection must be taken between *compatible* relations.

Set-Intersection Operation – Example 1

RELATIONS

R

A	B
α	1
α	2
β	1

S

A	B
α	2
β	3

R ∩ S:

A	B
α	2

Set-Intersection Operation

Define set-intersection by using basic operators

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

Natural-Join Operation

Notation

$$r \bowtie s$$

Defined as

- Let r and s be relations on schemas R and S respectively.
  Then, $r \bowtie s$ is a relation on schema $R \cup S$ obtained as follows:
  - If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result (concatenation of t_r and t_s)

Natural Join Operation – Example 1

RELATIONS

R

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

S

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ε

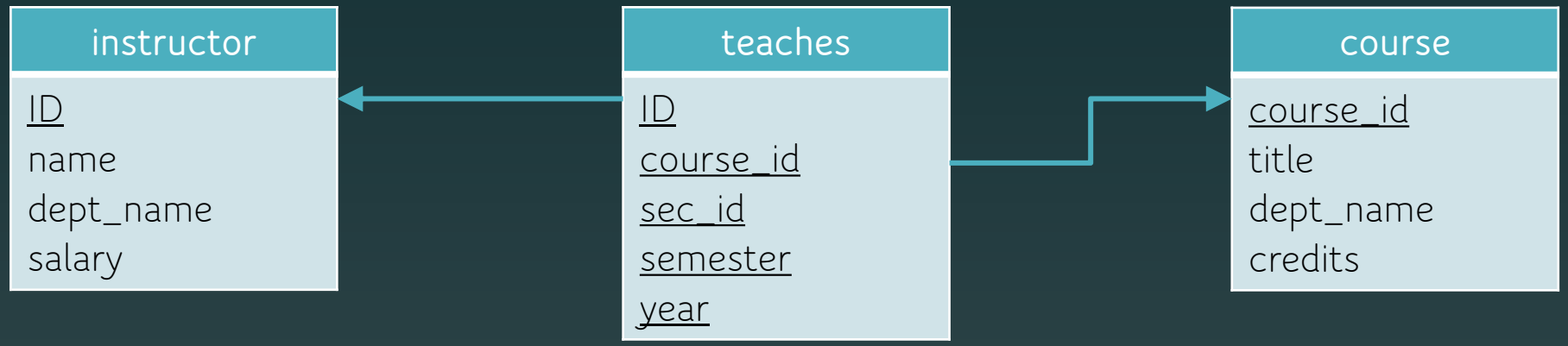
R ⋈ S

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

Note: natural join can be expressed using basic operations.
e.g. $r \bowtie s$ can be written as

$$\Pi_{r.A,\ r.B,\ r.C,\ r.D,\ s.E}\ (\sigma_{r.B = s.B \wedge r.D = s.D}\ (r \times s))$$

Natural Join Operation – Example 2



Find the names of all instructors in the Comp. Sci. department together with the course titles of all the courses that the instructors teach

$\Pi_{name,\ title}\ (\sigma_{dept\_name=\text{“Comp. Sci.”}}\ (instructor \bowtie teaches \bowtie course))$

$\Pi_{name,\ title}\ (\sigma_{dept\_name=\text{“Comp. Sci.”}}\ (instructor \bowtie teaches \bowtie \Pi_{course\_id,\ title}\ (course)))$

Natural Join and Theta Join

Natural join is associative

$(instructor \bowtie teaches) \bowtie course$ is equivalent to
$instructor \bowtie (teaches \bowtie course)$

Natural join is commutative

$instructor \bowtie teaches$ is equivalent to
$teaches \bowtie instructor$

The **theta join** operation

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

Assignment Operation

The assignment operation (←) provides a convenient way to express complex queries.

- Write query as a sequential program consisting of
  - a series of assignments
  - followed by an expression whose value is displayed as a result of the query.
- Assignment must always be made to a temporary relation variable.

Example: Find all instructor in the "Physics" and "Music" department.

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

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

$Physics \cup Music$

Outer Join

Notation:

- Left outer join: ⟕
- Right outer join: ⟖
- Full outer join: ⟗

Defined as

- 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

- *null* signifies that the value is unknown or does not exist
- All comparisons involving *null* are (roughly speaking) **false** by definition.

Outer Join Operation

Express outer join (e.g. r ⟖ s) using basic [and join – to simplify] operations

$$(r \bowtie s) \cup ((r - \prod_R (r \bowtie s)) \times \{(null, \ldots, null)\})$$

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*.

Aggregate functions simply ignore null values (as in SQL)

For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same (as in SQL)

Null Values

Comparisons with null values return the special truth value: *unknown*

- If *false* was used instead of *unknown*, then *not (A < 5)*
  would not be equivalent to *A >= 5*

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*
- In SQL "*P* **is unknown**" evaluates to true if predicate *P* evaluates to *unknown*

Result of select predicate is treated as *false* if it evaluates to *unknown*

Division Operation

Notation

$$r \div s$$

Defined as

Given relations r(R) and s(S), such that $S \subset R$,
r ÷ s is the largest relation t(R-S) such that $\mathbf{t \times s \subseteq r}$

E.g.

let $r(ID, course\_id) = \prod_{ID,\ course\_id}(takes)$ and
$s(course\_id) = \prod_{course\_id}(\sigma_{dept\_name="Biology"}(course))$
then **r ÷ s** = students who have taken all courses in the Biology Dept.

Division Operation

r

A	B
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
ε	6
ε	1
β	2

s

B
1
2

$r \div s$

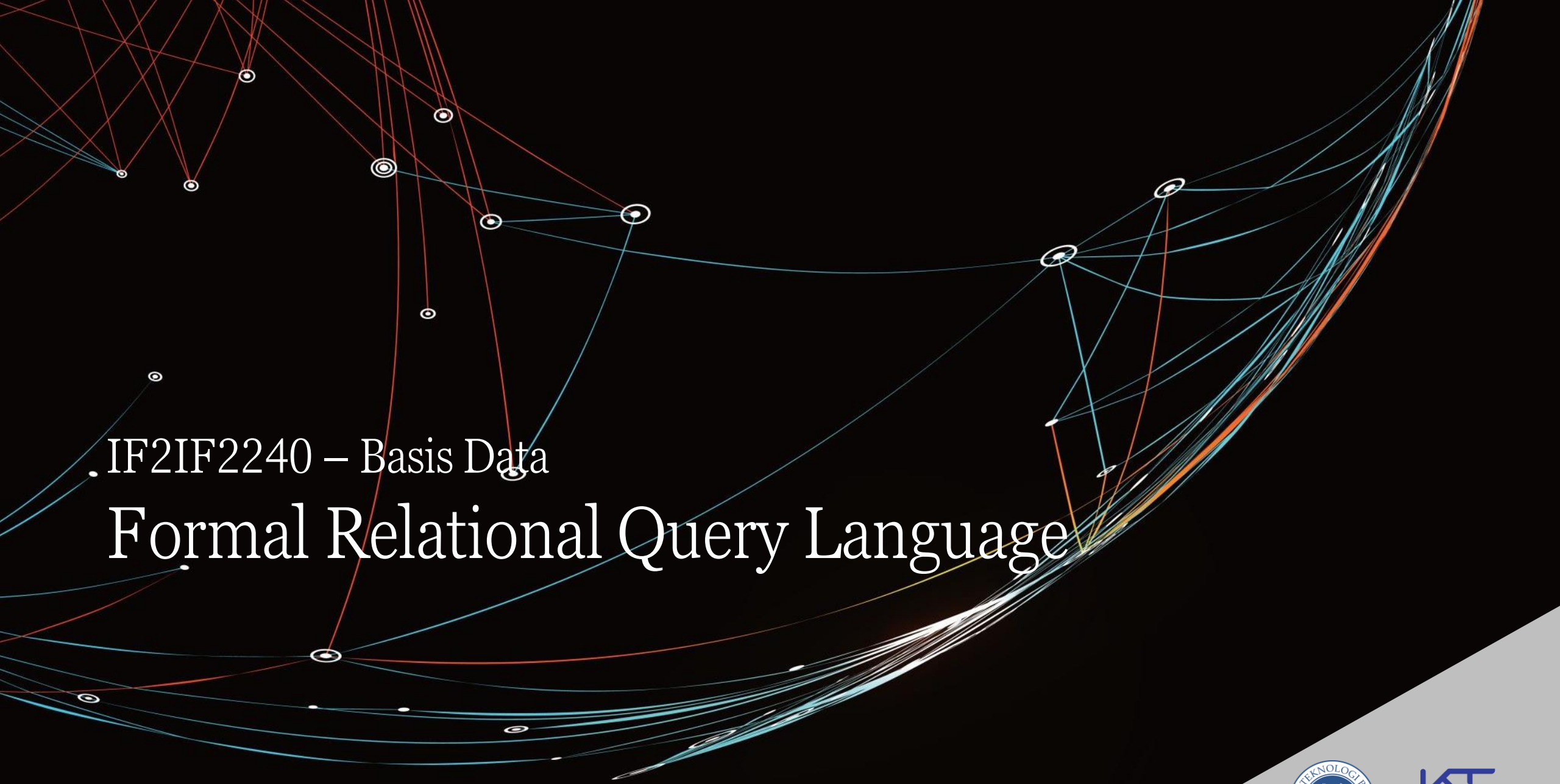
A
α
β

Express r $\div s$ using basic operations

$temp1 \leftarrow \prod_{R-S} (r)$

$temp2 \leftarrow \prod_{R-S} ((temp1 \times s) - \prod_{R-S,S} (r))$

$temp1 - temp2$

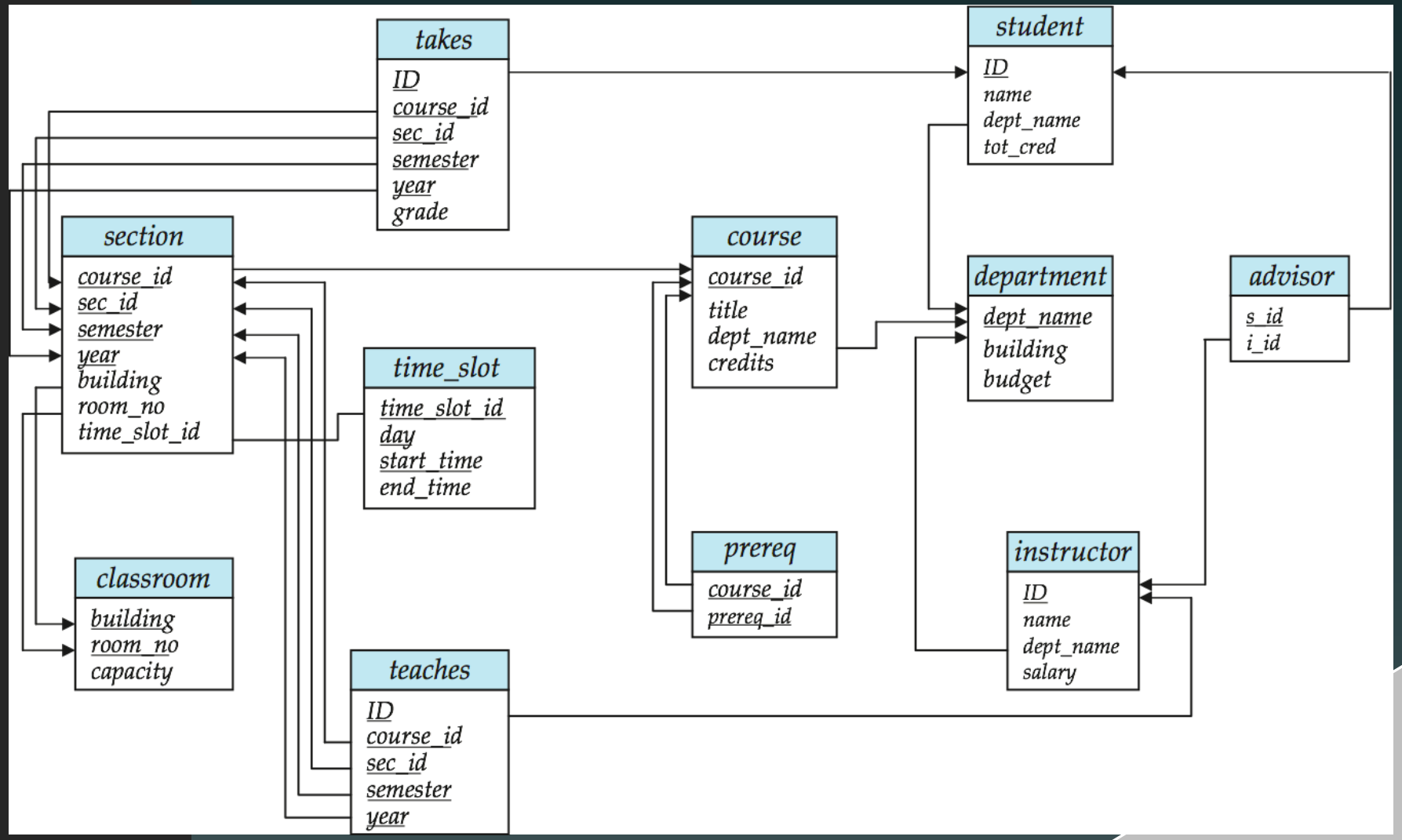


IF2IF2240 – Basis Data

Formal Relational Query Language

KNOWLEDGE & SOFTWARE ENGINEERING

Schema Diagram for University Database



KNOWLEDGE & SOFTWARE ENGINEERING

Relational Algebra

Procedural language

Six basic operators

- select: σ
- project: Π
- union: $\cup$
- set difference: $-$
- cartessian product: x
- rename: ρ

The operators take one or two relations as inputs and produce a new relation as a result.

- Additional Operators
  - Intersection: $\cap$
  - natural join:
  - assignment: $\leftarrow$
  - outer join: $⟗$
  - division: $\div$
- Extended Operators
  - generalized projection
  - aggregation

KNOWLEDGE & SOFTWARE ENGINEERING

Extended Operations

Generalized Projection

- Notation

$$\Pi_{F1, F2, \ldots, Fn}(E)$$

E is any relational-algebra expression
Each of F_1, F_2, …, F_n are arithmetic expressions

- E.g.

Given relation *credit-info(customer-name, limit, credit-balance)*
find how much more each person can spend

$$\Pi_{customer\text{-}name,\ limit\ -\ credit\text{-}balance}\ (credit\text{-}info)$$

Aggregate Functions and Operations (1)

Aggregate function

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

Notation

$$G1, G2, \ldots, Gn\ \mathcal{G}\ F1(A1), F2(A2), \ldots, Fn(An)\ (E)$$

The aggregated values do not have an attribute name; they can be given a name either by using the rename operator **ρ** or for convenience using the following syntax: $\mathcal{G}$ F1(A1) as <atr-name>

Aggregate Functions and Operations – Example 1

R

A	B	C
α	α	7
α	β	7
β	β	3
β	β	10

$\mathcal{G}_{\text{SUM(C)}}(R)$

sum-C
27

Aggregate Functions and Operations – Example 2

INSTRUCTOR (ID, NAME, DEPT_NAME, SALARY)

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

DEPT_NAME $\mathcal{G}$ AVG(SALARY) (INSTRUCTOR)

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

Data Modifications

Deletion

Notation

$$r \leftarrow r - E$$

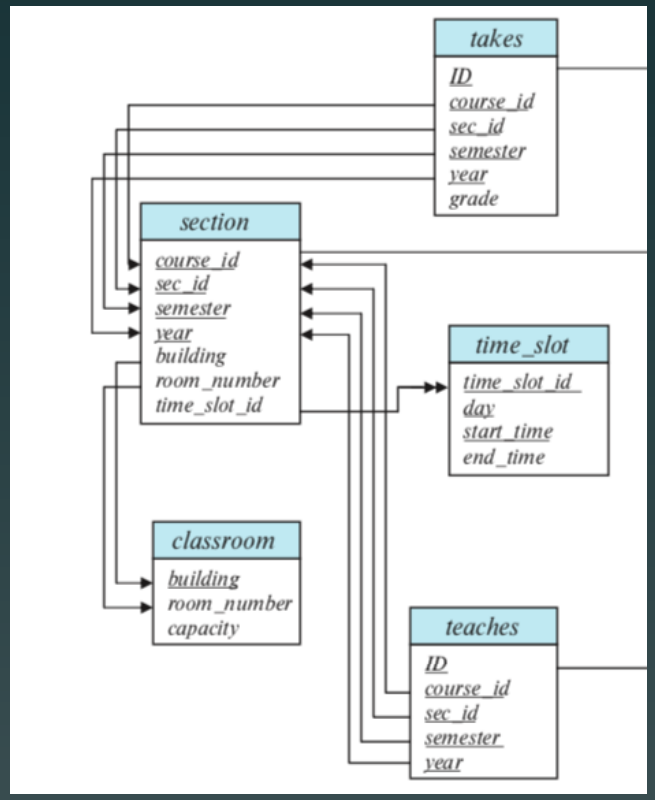
Example

prereq (<u>course_id</u>, <u>prereq_id</u>)

Delete all prerequisites of course "IF2240"

$$prereq \leftarrow prereq - \sigma_{course\_id = "IF2240"} (prereq)$$

Deletion Examples



@Silberschatz et.al. (2020)

Delete the study plan of student with ID "13518000" for 1-2019 semester.

$takes \leftarrow takes - \sigma_{ID="13518000" \wedge sem=1 \wedge year=2019} (takes)$

Delete all sections that was taught by instructor with ID "132132132" for 2-2019 semester.

$r_1 \leftarrow \sigma_{ID = "132132132" \wedge sem=2 \wedge year=2019} (teaches)$

$r_2 \leftarrow \Pi_{course\_id, sec\_id, sem, year} (r_1) \bowtie takes$

$r_3 \leftarrow \Pi_{course\_id, sec\_id, sem, year} (r_1) \bowtie section$

$teaches \leftarrow teaches - r_1$

$takes \leftarrow takes - r_2$

$section \leftarrow section - r_3$

Insertion

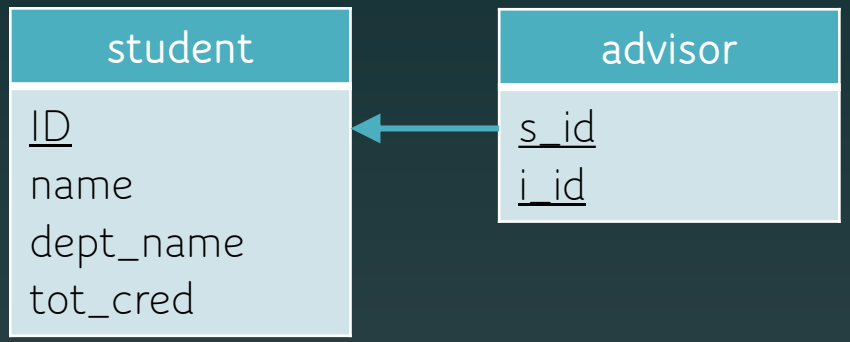
Types

1. specify a tuple to be inserted
2. write a query whose result is a set of tuples to be inserted

Notation

$$r \leftarrow r \cup E$$

Insertion – Example 1

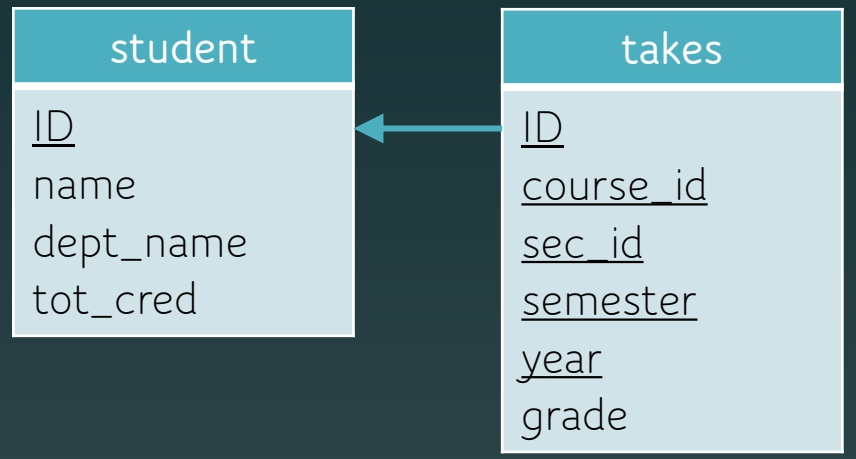


Insert information in the database specifying that a transfer student, Abdul, with ID 13518600 was enrolled to Comp. Sci. department with 36 total credit transfer and instructor 132132132 as his advisor.

$$student \leftarrow student \cup \{(13518600, \text{“Abdul”}, \text{“Comp. Sci.”}, 36)\}$$

$$advisor \leftarrow advisor \cup \{(13518600, 132132132)\}$$

Insertion – Example 2



All students from Comp. Sci. dept with less than 130 total credits are automatically enrolled to course IF4000 in 2-2019 semester (evenly distributed to 3 available section IDs: 1, 2, 3)

$$r_1 \leftarrow \sigma_{dept\_name="Comp.Sci." \wedge tot\_cred<130}(student)$$

$$takes \leftarrow takes \cup \prod_{ID,\ "IF4000",\ ((ID-1)\ mod\ 3)+1,\ 2,\ 2019,\ null}(r_1)$$

Updating

Use the generalized projection operator to do this task

$$r \leftarrow \Pi_{F1, F2, \ldots, Fn} (r)$$

Each F_i is either

- the i^{th} attribute of r, if the i^{th} attribute is not updated, or,
- if the attribute is to be updated F_i is an expression, involving only constants and the attributes of r, which gives the new value for the attribute

Update Examples

instructor
ID
name
dept_name
salary

Give a 5% salary raise to all instructors.

$$instructor \leftarrow \prod_{ID,\ name,\ dept\_name,\ salary\ *\ 1.05} (instructor)$$

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

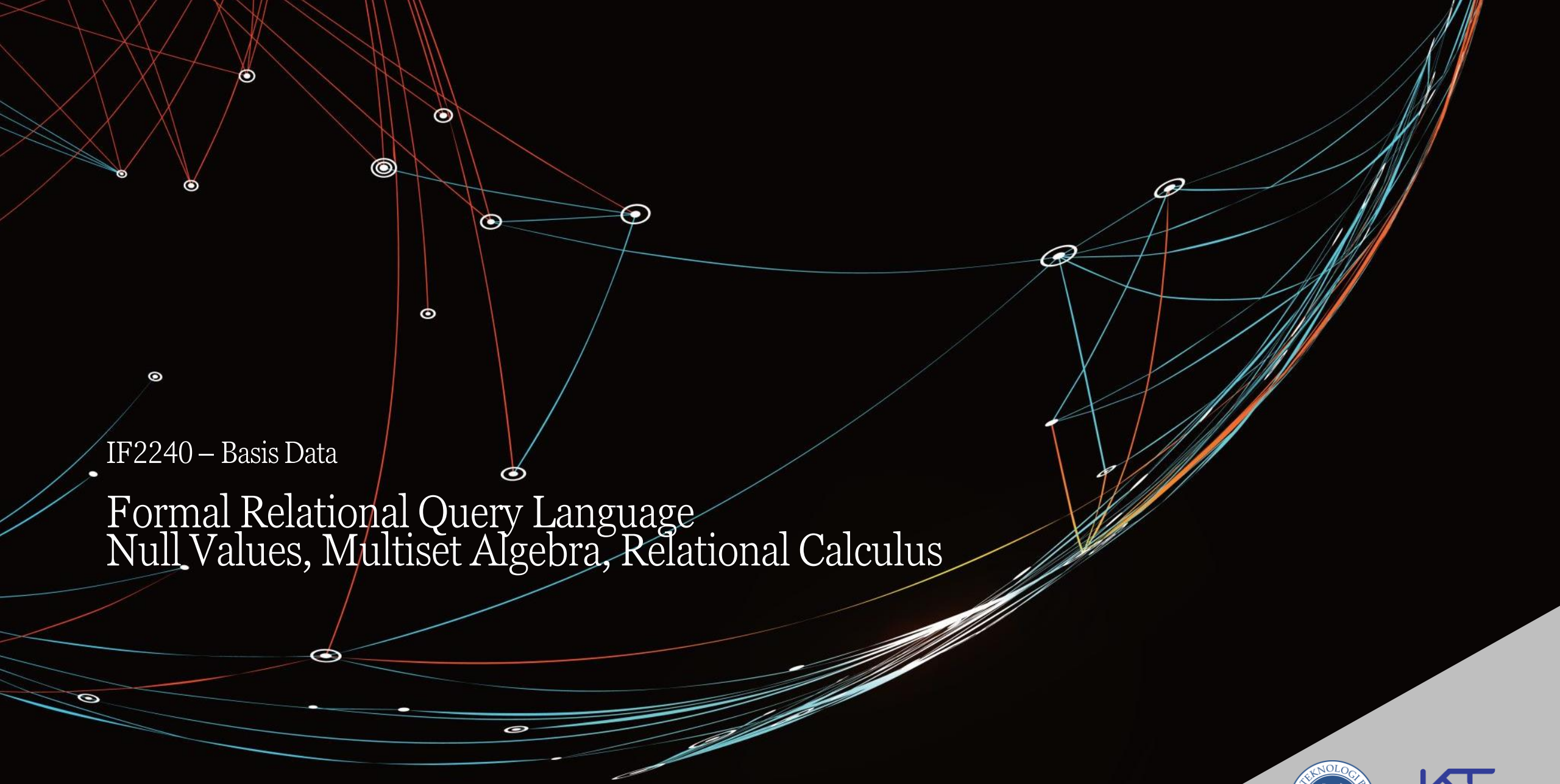
$$instructor \leftarrow \prod_{ID,\ name,\ dept\_name,\ salary\ *\ 1.05} (\sigma_{salary\ <\ 70000} (instructor))$$

$$\cup\ \sigma_{salary\ \geq\ 70000} (instructor)$$

Increase salaries of instructors whose salary is over $70,000 by 3%, and all others receive a 5% raise.

$$instructor \leftarrow \prod_{ID,\ name,\ dept\_name,\ salary\ *\ 1.05} (\sigma_{salary\ \leq\ 70000} (instructor))$$

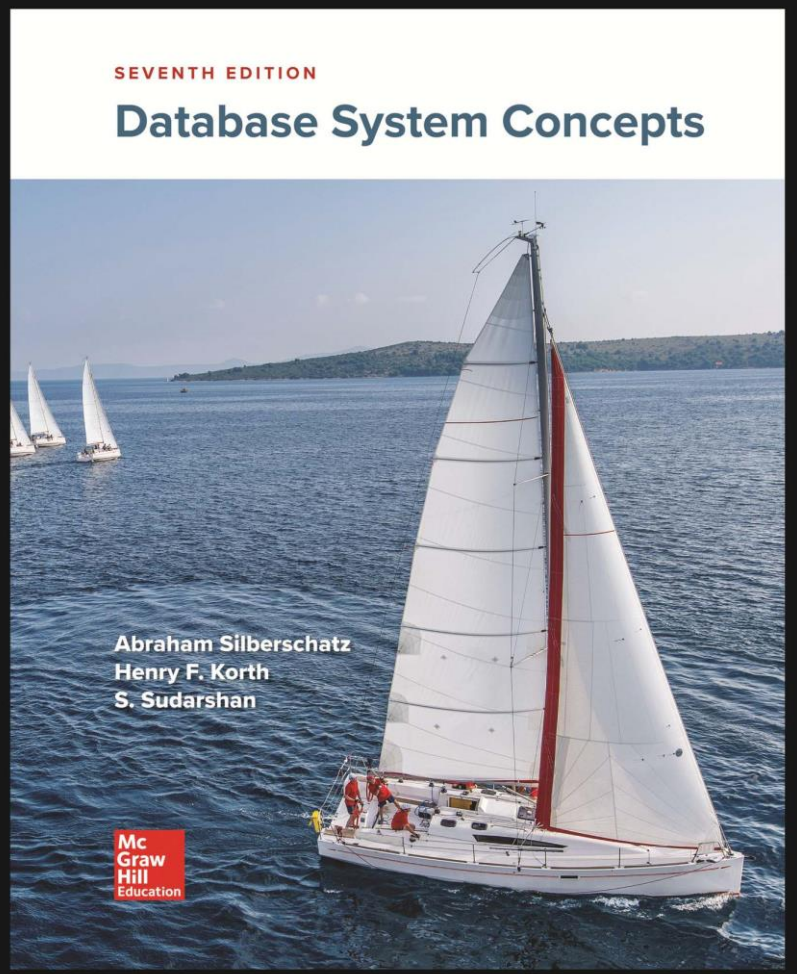
$$\cup \prod_{ID,\ name,\ dept\_name,\ salary\ *\ 1.03} (\sigma_{salary\ >\ 70000} (instructor))$$



IF2240 – Basis Data

Formal Relational Query Language
Null Values, Multiset Algebra, Relational Calculus

KNOWLEDGE & SOFTWARE ENGINEERING



Sumber

Silberschatz, Korth, Sudarshan: "Database System Concepts", 7th Edition

- Chapter 2: Relational Model
  - Section 2.6. The Relational Algebra
- Chapter 27: Formal-Relational Query Languages (*online chapter*)
  - Section 27.1. The Tuple Relational Calculus
  - Section 27.2. The Domain Relational Calculus

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*.

Aggregate functions simply ignore null values (as in SQL)

For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same (as in SQL)

Null Values

Comparisons with null values return the special truth value: *unknown*

- If *false* was used instead of *unknown*, then *not (A < 5)*
  would not be equivalent to *A >= 5*

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*
- In SQL "*P* **is unknown**" evaluates to true if predicate *P* evaluates to *unknown*

Result of select predicate is treated as *false* if it evaluates to *unknown*

Multiset Relational Algebra

Pure relational algebra removes all duplicates

- e.g. after projection

Multiset relational algebra retains duplicates, to match SQL semantics

- SQL duplicate retention was initially for efficiency, but is now a feature

Multiset relational algebra defined as follows

- selection: has as many duplicates of a tuple as in the input, if the tuple satisfies the selection
- projection: one tuple per input tuple, even if it is a duplicate
- cross product: If there are *m* copies of *t1* in *r*, and *n* copies of *t2* in *s*, there are *m* x *n* copies of *t1.t2* in *r* x *s*
- Other operators similarly defined
  - E.g. union: *m* + *n copies,* intersection: min(*m*, *n*) copies
    difference: min(0, *m* – *n*) copies

Relational Calculus

Tuple Relational Calculus

A nonprocedural query language, where each query is of the form

$$\{t \mid P(t)\}$$

It is the set of all tuples t such that predicate P is true for t

t is a *tuple variable*, $t[A]$ denotes the value of tuple t on attribute A

$t \in r$ denotes that tuple t is in relation r

P is a *formula* similar to that of the predicate calculus

Predicate Calculus Formula

1. Set of attributes and constants
2. Set of comparison operators: (e.g., $<$, $\leq$, $=$, $\neq$, $>$, $\geq$)
3. Set of connectives: and ($\wedge$), or ($\vee$), not ($\neg$)
4. Implication ($\Rightarrow$): x $\Rightarrow$ y, if x is true, then y is true

$$x \Rightarrow y \equiv \neg x \vee y$$

5. Set of quantifiers:
   - $\exists\, t \in r\, (Q\,(t\,)) \equiv$ "there exists" a tuple t in relation r such that predicate $Q\,(t\,)$ is true
   - $\forall t \in r\, (Q\,(t\,)) \equiv Q$ is true "for all" tuples t in relation r

Example Queries

Find the *ID, name, dept_name, salary* for instructors whose salary is greater than $80,000

$\{t \mid t \in instructor \wedge t[salary] > 80000\}$

As in the previous query, but output only the *ID* attribute value

$\{t \mid \exists\ s \in \text{instructor}\ (t[ID] = s[ID] \wedge s[salary] > 80000)\}$

Notice that a relation on schema (*ID*) is implicitly defined by the query

schema

Example Queries

Find the names of all instructors whose department is in the Watson building

$$\{t \mid \exists s \in instructor\ (t[name] = s[name] \wedge \exists u \in department\ (u[dept\_name] = s[dept\_name] \wedge u[building] = \text{“Watson”}))\}$$

Find the set of all courses taught in the Fall 2017 semester, or in the Spring 2018 semester, or both

$$\{t \mid \exists s \in section\ (t[course\_id] = s[course\_id] \wedge s[semester] = \text{“Fall”} \wedge s[year] = 2017) \vee \exists u \in section\ (t[course\_id] = u[course\_id] \wedge u[semester] = \text{“Spring”} \wedge u[year] = 2018)\}$$

schema

Example Queries

Find the set of all courses taught in the Fall 2017 semester, and in the Spring 2018 semester

$$\{t \mid \exists s \in section\ (t[course\_id] = s[course\_id] \wedge s[semester] = \text{“Fall”} \wedge s[year] = 2017) \wedge \exists u \in section\ (t[course\_id] = u[course\_id] \wedge u[semester] = \text{“Spring”} \wedge u[year] = 2018)\}$$

Find the set of all courses taught in the Fall 2017 semester, but not in the Spring 2018 semester

$$\{t \mid \exists s \in section\ (t[course\_id] = s[course\_id] \wedge s[semester] = \text{“Fall”} \wedge s[year] = 2017) \wedge \neg \exists u \in section\ (t[course\_id] = u[course\_id] \wedge u[semester] = \text{“Spring”} \wedge u[year] = 2018)\}$$

schema

Safety of Expressions

It is possible to write tuple calculus expressions that generate infinite relations.

For example, { t | $\neg$ $t \in r$ } results in an infinite relation if the domain of any attribute of relation r is infinite

To guard against the problem, we restrict the set of allowable expressions to safe expressions.

An expression $\{t \mid P(t)\}$ in the tuple relational calculus is *safe* if every component of t appears in one of the relations, tuples, or constants that appear in P

- NOTE: this is more than just a syntax condition.
  - E.g. { $t \mid t[A] = 5 \vee$ **true** } is not safe --- it defines an infinite set with attribute values that do not appear in any relation or tuples or constants in P.

Universal Quantification

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

$$\{t \mid \exists\, r \in \mathit{student}\ (t\,[\mathit{ID}] = r\,[\mathit{ID}]) \wedge$$
$$(\forall\, u \in \mathit{course}\ (u\,[\mathit{dept\_name}] = \text{“Biology”} \Rightarrow$$
$$\exists\, s \in \mathit{takes}\ (t\,[\mathit{ID}] = s\,[\mathit{ID}\ ] \wedge$$
$$s\,[\mathit{course\_id}] = u\,[\mathit{course\_id}]))\}$$

- Note that without the existential quantification on student, the above query would be unsafe if the Biology department has not offered any courses.

schema

Domain Relational Calculus

A nonprocedural query language equivalent in power to the tuple relational calculus

Each query is an expression of the form:

$$\{ < x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n) \}$$

- $x_1, x_2, \ldots, x_n$ represent domain variables
- P represents a formula similar to that of the predicate calculus

Example Queries

Find the *ID, name, dept_name, salary* for instructors whose salary is greater than $80,000

$\{< i, n, d, s> \mid < i, n, d, s> \in instructor \wedge s > 80000\}$

As in the previous query, but output only the *ID* attribute value

$\{< i> \mid \exists\, n, d, s\ (< i, n, d, s> \in instructor \wedge s > 80000)\}$

Find the names of all instructors whose department is in the Watson building

$\{< n > \mid \exists\, i, d, s\ (< i, n, d, s > \in instructor$
$\wedge \exists\, b, a\ (< d, b, a> \in department \wedge b = \text{“Watson”}\ ))\}$

schema

Example Queries

Find the set of all courses taught in the Fall 2017 semester, or in the Spring 2018 semester, or both

$$\{<c> \mid \exists\, a, s, y, b, r, t\ (<c, a, s, y, b, r, t> \in section \wedge s = \text{“Fall”} \wedge y = 2017) \vee \exists\, a, s, y, b, r, t\ (<c, a, s, y, b, r, t> \in section] \wedge s = \text{“Spring”} \wedge y = 2018)\}$$

This case can also be written as

$$\{<c> \mid \exists\, a, s, y, b, r, t\ (<c, a, s, y, b, r, t> \in section \wedge ((s = \text{“Fall”} \wedge y = 2017) \vee (s = \text{“Spring”} \wedge y = 2018))\}$$

Find the set of all courses taught in the Fall 2017 semester, and in the Spring 2018 semester

$$\{<c> \mid \exists\, a, s, y, b, r, t\ (<c, a, s, y, b, r, t> \in section \wedge s = \text{“Fall”} \wedge y = 2017) \wedge \exists\, a, s, y, b, r, t\ (<c, a, s, y, b, r, t> \in section] \wedge s = \text{“Spring”} \wedge y = 2018)\}$$

schema

Safety of Expressions

The expression:

$$\{ < x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n) \}$$

is safe if all of the following hold:

1. All values that appear in tuples of the expression are values from *dom* (P) (that is, the values appear either in P or in a tuple of a relation mentioned in P).
2. For every "there exists" subformula of the form $\exists x (P_1(x))$, the subformula is true if and only if there is a value of x in *dom* (P_1) such that $P_1(x)$ is true.
3. For every "for all" subformula of the form $\forall_x (P_1(x))$, the subformula is true if and only if $P_1(x)$ is true for all values x from *dom* (P_1).

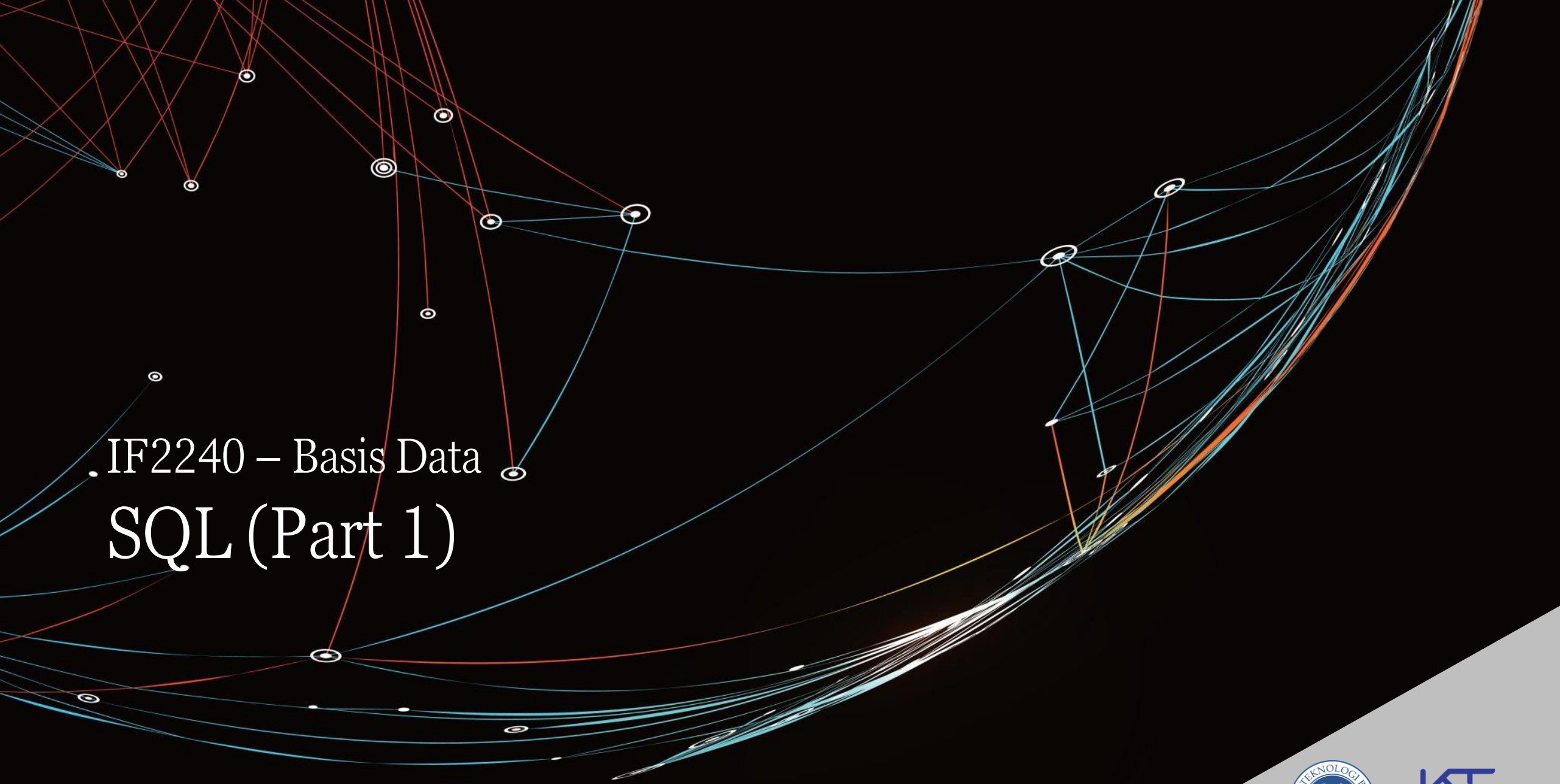
Universal Quantification

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

$$\{< i > \mid \exists\, n, d, tc\ (< i, n, d, tc > \in student \ \wedge$$
$$(\forall\, ci, ti, dn, cr\ (< ci, ti, dn, cr > \in course \wedge dn = \text{“Biology”}$$
$$\Rightarrow \exists\, si, se, y, g\ (< i, ci, si, se, y, g > \in takes\ ))\}$$

- Note that without the existential quantification on student, the above query would be unsafe if the Biology department has not offered any courses.

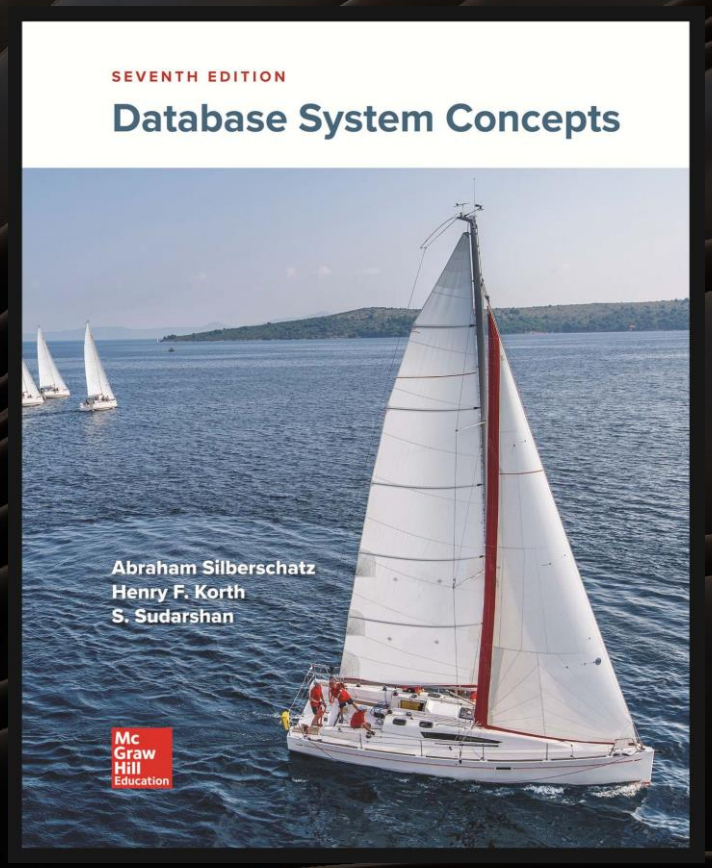
schema



IF2240 – Basis Data

SQL (Part 1)

KNOWLEDGE & SOFTWARE ENGINEERING



Sumber

Silberschatz, Korth, Sudarshan: "Database System Concepts", 7th Edition

- Chapter 3 : Introduction to SQL

History

IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory

Renamed Structured Query Language (SQL)

ANSI and ISO standard SQL:

- SQL-86
- SQL-89
- SQL-92
- SQL:1999 (language name became Y2K compliant!)
- SQL:2003

Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.

- Not all examples here may work on your particular system.

SQL Parts

DML -- provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.

integrity – the DDL includes commands for specifying integrity constraints.

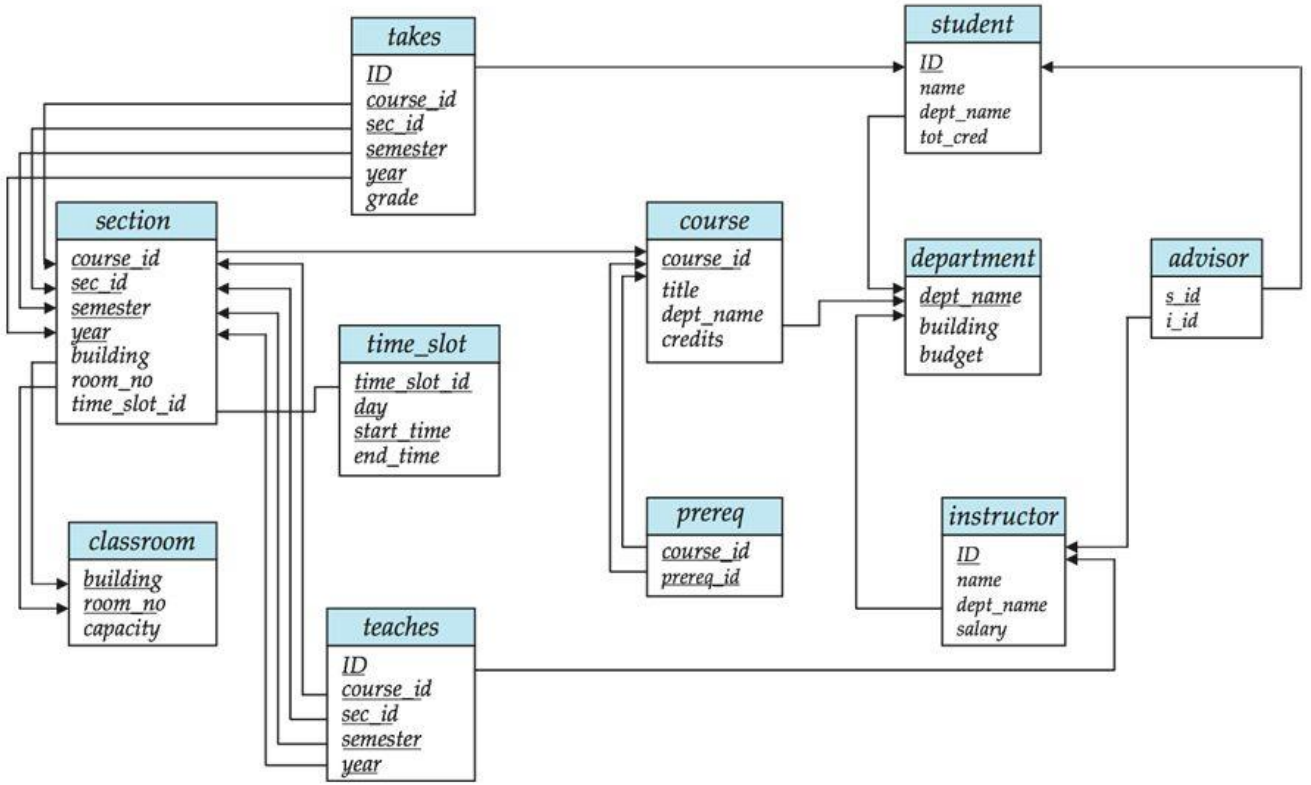
View definition -- The DDL includes commands for defining views.

Transaction control –includes commands for specifying the beginning and ending of transactions.

Embedded SQL and dynamic SQL -- define how SQL statements can be embedded within general-purpose programming languages.

Authorization – includes commands for specifying access rights to relations and views.

Schema Diagram



Basic Query Structure

A typical SQL query has the form:

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

- A_i represents an attribute
- R_i represents a relation
- P is a predicate.

The result of an SQL query is a relation.

The select Clause

The **select** clause lists the attributes desired in the result of a query

- corresponds to the projection operation of the relational algebra

Example: find the names of all instructors:

select *name*
from *instructor*

NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)

- E.g., *Name* ≡ *NAME* ≡ *name*
- Some people use upper case wherever we use bold font.

The select Clause (Cont.)

SQL allows duplicates in relations as well as in query results.

To force the elimination of duplicates, insert the keyword **distinct** after select.

Find the department names of all instructors, and remove duplicates

```
select distinct dept_name
from instructor
```

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

```
select all dept_name
from instructor
```

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

The select Clause (Cont.)

An asterisk in the select clause denotes "all attributes"

```
select *
from instructor
```

An attribute can be a literal with no **from** clause : **select** '437'

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

```
select '437' as FOO
```

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

```
select 'A'
from instructor
```

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

The select Clause (Cont.)

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

- The query:

  select *ID, name, salary/12*
  from *instructor*

  would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.

- Can rename "*salary/12*" using the **as** clause:

  select *ID, name, salary/12* **as** *monthly_salary*

The where Clause

The **where** clause specifies conditions that the result must satisfy

- Corresponds to the selection predicate of the relational algebra.

To find all instructors in Comp. Sci. dept

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

SQL allows the use of the logical connectives **and, or,** and **not.** The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>. Comparisons can be applied to results of arithmetic expressions

To find all instructors in Comp. Sci. dept with salary > 70000

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

name
Katz
Brandt

The from Clause

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

- Corresponds to the Cartesian product operation of the relational algebra.

Find the Cartesian product *instructor X teaches*

```
select *
from instructor, teaches
```

- generates every possible instructor – teaches pair, with all attributes from both relations.
- For common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name (e.g., *instructor.ID*)

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

Examples

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

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

Find the names of all instructors in the Art department who have taught some course and the course_id

- **select** *name, course_id*
  from *instructor , teaches*
  where *instructor.ID = teaches.ID*
  and *instructor. dept_name* = 'Art'

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

The Rename Operation

The SQL allows renaming relations and attributes using the **as** clause:

old-name **as** *new-name*

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

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

Keyword **as** is optional and may be omitted

instructor **as** *T* ≡ *instructor T*

Self "Join" Example

Relation *emp-super*

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

Find the supervisor of "Bob"

Find the supervisor of the supervisor of "Bob"

Can you find ALL the supervisors (direct and indirect) of "Bob"?

String Operations

SQL includes a string-matching operator for comparisons on character strings. The operator **like** uses patterns that are described using two special characters:

- percent (%). The % character matches any substring.
- underscore (_). The _ character matches any character.

Find the names of all instructors whose name includes the substring "dar".

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

Match the string "100%"

```
like '100 \%'  escape  '\'
```

in that above we use backslash (\) as the escape character.

String Operations (Cont.)

Patterns are case sensitive.

Pattern matching examples:

- 'Intro%' matches any string beginning with “Intro”.
- '%Comp%' matches any string containing “Comp” as a substring.
- '_ _ _' matches any string of exactly three characters.
- '_ _ _ %' matches any string of at least three characters.

SQL supports a variety of string operations such as

- concatenation (using “||”)
- converting from upper to lower case (and vice versa)
- finding string length, extracting substrings, etc.

Ordering the Display of Tuples

List in alphabetic order the names of all instructors

```
select distinct name
from    instructor
order by name
```

We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.

- Example: **order by** *name* **desc**

Can sort on multiple attributes

- Example: **order by** *dept_name, name*

Where Clause Predicates

SQL includes a **between** comparison operator

Example: Find the names of all instructors with salary between \$90,000 and \$100,000 (that is, ≥ \$90,000 and ≤ \$100,000)

- **select** *name*
  from *instructor*
  where *salary* **between** 90000 **and** 100000

Tuple comparison

- **select** *name*, *course_id*
  from *instructor*, *teaches*
  where (*instructor.ID*, *dept_name*) = (*teaches.ID*, 'Biology');

Set Operations

- Find courses that ran in Fall 2017 **or** in Spring 2018

```
(select course_id  from section where sem = 'Fall' and year = 2017)
union
 (select course_id  from section where sem = 'Spring' and year = 2018)
```

- Find courses that ran in Fall 2017 **and** in Spring 2018

```
(select course_id  from section where sem = 'Fall' and year = 2017)
intersect
 (select course_id  from section where sem = 'Spring' and year = 2018)
```

- Find courses that ran in Fall 2017 **but not in** Spring 2018

```
(select course_id  from section where sem = 'Fall' and year = 2017)
except
 (select course_id  from section where sem = 'Spring' and year = 2018)
```

Set Operations (Cont.)

Set operations **union**, **intersect**, and **except**

- Each of the above operations automatically eliminates duplicates

To retain all duplicates use the

- **union all**,
- **intersect all**
- **except all**.

Null Values

It is possible for tuples to have a null value, denoted by **null**, for some of their attributes

null signifies an unknown value or that a value does not exist.

The result of any arithmetic expression involving **null** is **null**

- Example: 5 + **null** returns **null**

The predicate **is null** can be used to check for null values.

- Example: Find all instructors whose salary is null.

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

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

Null Values (Cont.)

SQL treats as **unknown** the result of any comparison involving a null value (other than predicates **is null** and **is not null**).

- Example: *5* < **null** or **null** <> **null** or **null** = **null**

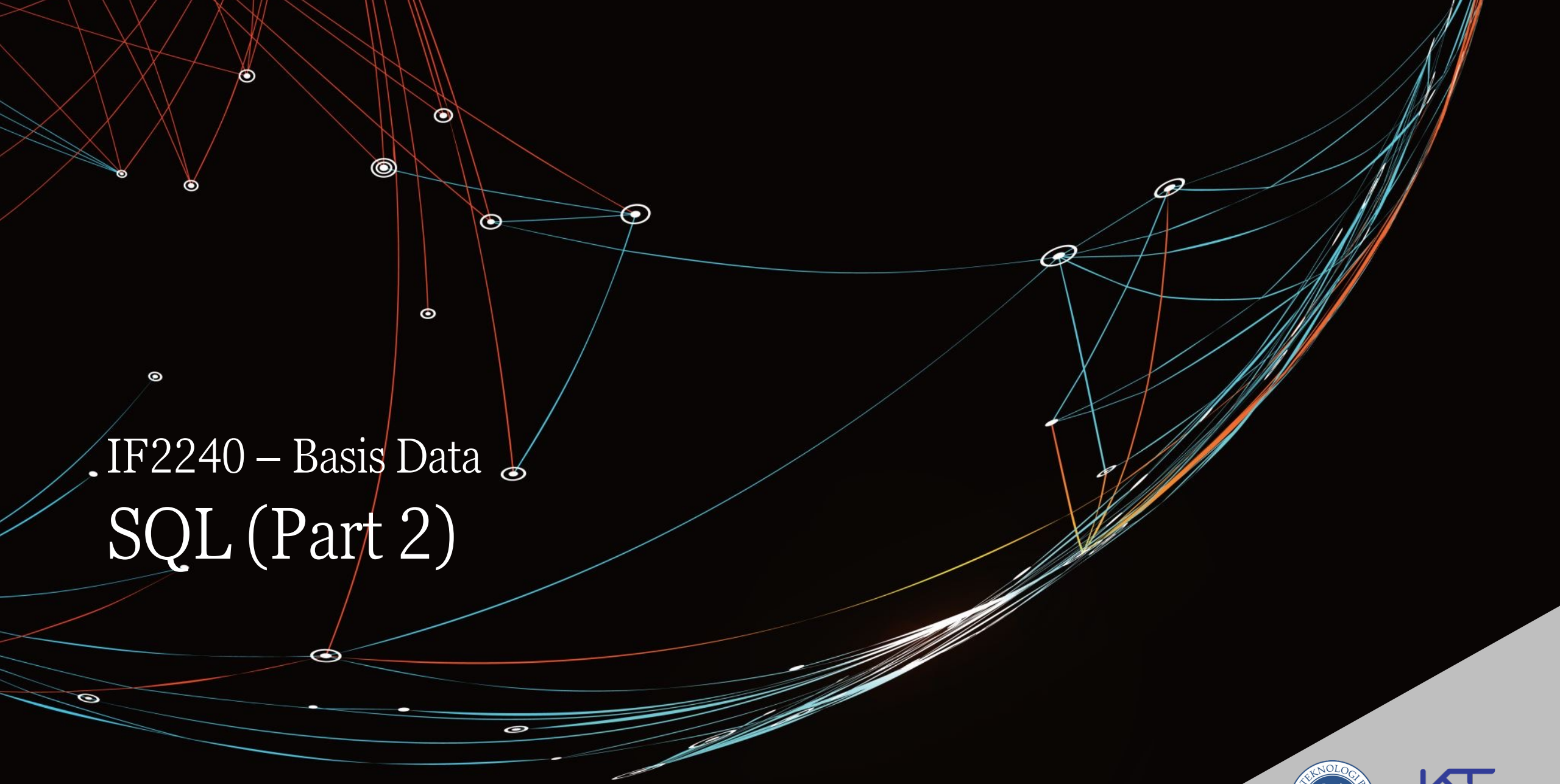
The predicate in a **where** clause can involve Boolean operations (**and**, **or**, **not**); thus the definitions of the Boolean operations need to be extended to deal with the value **unknown**.

- **and** : (*true* **and** *unknown*) = *unknown*,
  (*false* **and** *unknown*) = *false*,
  (*unknown* **and** *unknown*) = *unknown*
- **or**: (*unknown* **or** *true*) = *true*,
  (*unknown* **or** *false*) = *unknown*
  (*unknown* **or** *unknown*) = *unknown*

Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Latihan-1

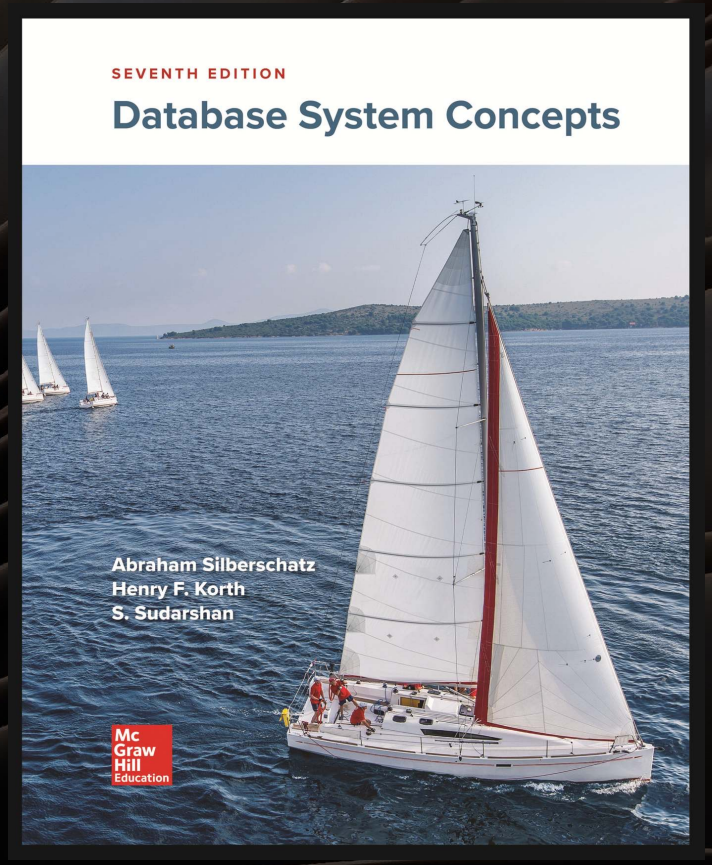
1. Tampilkan data ID dan nama student yang berasal dari departemen “Comp. Sci.”.
2. Tampilkan data instruktur dari departemen “Comp. Sci” yang memiliki gaji antara 7.000.000 hingga 12.000.000 terurut mengecil berdasarkan gaji.
3. Tampilkan data student dengan nama berawalan “Budi” dan terdiri dari sekurang-kurangnya 2 kata.
4. Tampilkan daftar mata kuliah (kode, nama, sks) yang diambil oleh student dengan ID “13521400” pada semester 2 tahun 2022.
5. Tampilkan daftar seluruh kelas yang ditawarkan oleh departemen “Comp. Sci.” pada semester 2 tahun 2022. Informasi yang ingin ditampilkan meliputi kode kuliah, nama, jumlah sks, nomor kelas (sec_id), serta nama dari instruktur kelas tersebut. Daftar terurut berdasarkan kode kuliah dan nomor kelas.



IF2240 – Basis Data

SQL (Part 2)

KNOWLEDGE & SOFTWARE ENGINEERING

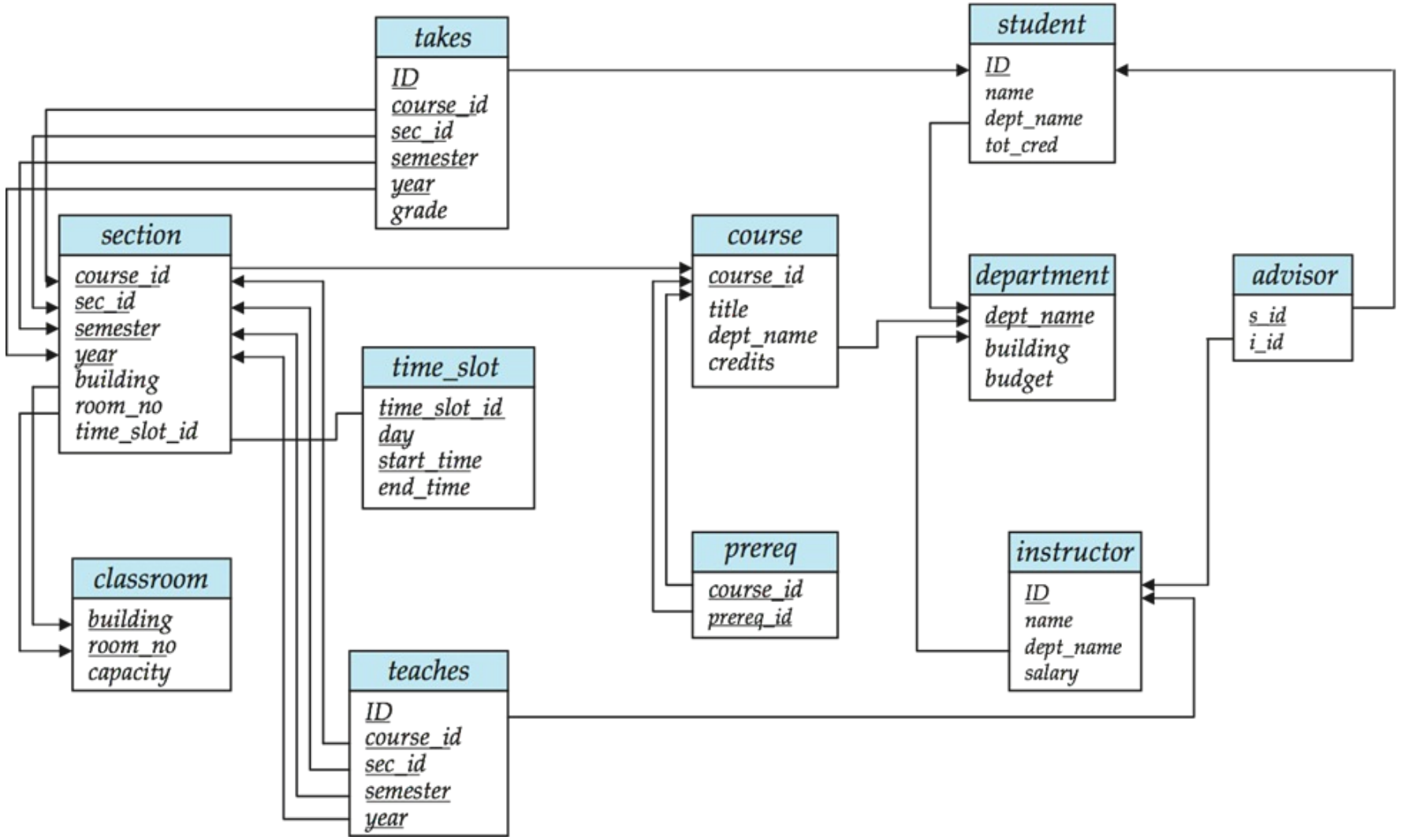


Sumber

Silberschatz, Korth, Sudarshan: "Database System Concepts", 7th Edition

- Chapter 3 : Introduction to SQL

KNOWLEDGE & SOFTWARE ENGINEERING


takes
ID
course_id
sec_id
semester
year
grade
student
ID
name
dept_name
tot_cred
section
course_id
sec_id
semester
year
building
room_no
time_slot_id
course
course_id
title
dept_name
credits
department
dept_name
building
budget
advisor
s_id
i_id
time_slot
time_slot_id
day
start_time
end_time
prereq
course_id
prereq_id
instructor
ID
name
dept_name
salary
classroom
building
room_no
capacity
teaches
ID
course_id
sec_id
semester
year

Aggregate Functions

These functions operate on the multiset of values of a column of a relation, and return a value

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

Aggregate Functions Examples

Find the average salary of instructors in the Computer Science department

- **select avg** (*salary*)
  from *instructor*
  where *dept_name*= 'Comp. Sci.';

Find the total number of instructors who teach a course in the Spring 2018 semester

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

Find the number of tuples in the *course* relation

- **select count** (*)
  from *course*;

Aggregate Functions – Group By

Group By: Group tuple that have the same values into summary rows

- Often used with aggregate functions

Find the average salary of instructors in each department

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

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

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

Aggregation (Cont.)

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

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

Aggregate Functions – Having Clause

HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions

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

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

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

How About Some More Queries?

6. Find the highest salary earned by instructors from Comp. Sci. Department.
7. Find the enrollment of each section that was offered in Autumn 2009.
8. Find all departments with more than 1000 student enrollments, together with the number of students for each of them. The list should be ordered by dept_name
9. Find all instructors earning the highest salary (there may be more than one with the same salary)

Joined Relations

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

A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join

The join operations are typically used as subquery expressions in the **from** clause

Three types of joins:

- Natural join
- Inner join
- Outer join

Natural Join in SQL

Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column.

List the names of instructors along with the course ID of the courses that they taught

- **select** *name, course_id*
  from *students, takes*
  where *student.ID = takes.ID*;

Same query in SQL with "natural join" construct

- **select** *name, course_id*
  from *student* **natural join** *takes*;

Natural Join in SQL (Cont.)

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

select $A_1, A_2, \ldots A_n$
from r_1 **natural join** r_2 **natural join** .. **natural join** r_n
where P ;

Student Relation

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

Takes Relation

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

student natural join *takes*

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

Dangerous in Natural Join

Beware of unrelated attributes with same name which get equated incorrectly

Example -- List the names of students instructors along with the titles of courses that they have taken

- Correct version

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

- Incorrect version

```
select name, title
from student natural join takes natural join course;
```

  - This query omits all (student name, course title) pairs where the student takes a course in a department other than the student's own department.
  - The correct version (above), correctly outputs such pairs.

Outer Join

An extension of the join operation that avoids loss of information.

Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.

Uses *null* values.

Three forms of outer join:

- left outer join
- right outer join
- full outer join

Outer Join Examples

Relation *course*

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

Relation *prereq*

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

Observe that

course information is missing CS-347

prereq information is missing CS-315

Left Outer Join

course natural left outer join *prereq*

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

In relational algebra: *course* ⟕ *prereq*

Right Outer Join

course natural right outer join *prereq*

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

In relational algebra: *course* ⟖ *prereq*

Full Outer Join

course natural full outer join *prereq*

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

In relational algebra: *course* ⟗ *prereq*

Joined Types and Conditions

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

These additional operations are typically used as subquery expressions in the **from** clause

Join condition – defines which tuples in the two relations match.

- **natural**
- **on** <predicate>
- **using** (A_1, A_2, …, A_n)

Join type – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

- **inner join**
- **left outer join**
- **right outer join**
- **full outer join**

Joined Relations – Examples

course natural right outer join *prereq*

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

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

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

Joined Relations – Examples

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

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

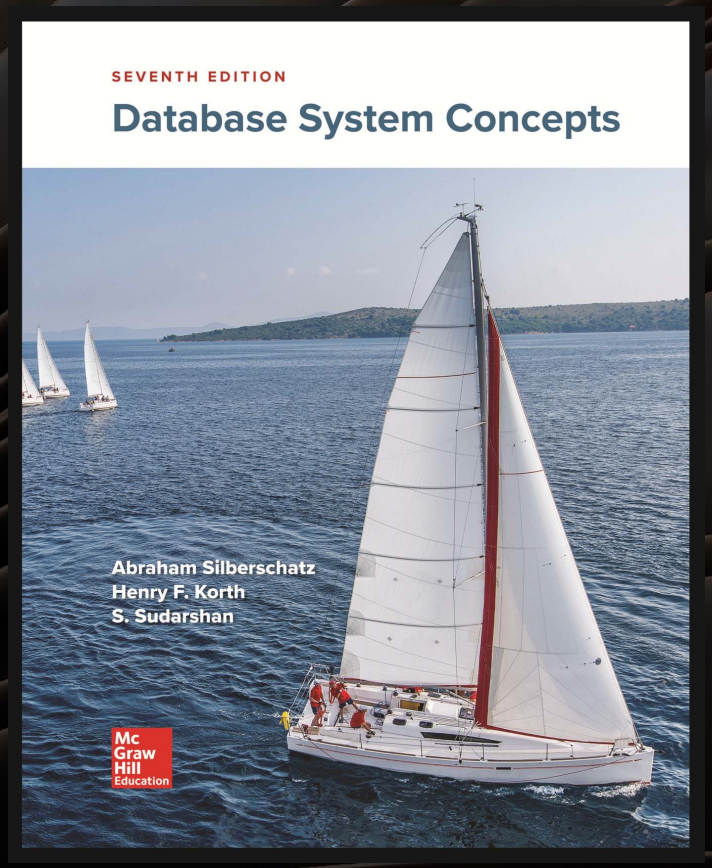
What is the difference between the above, and a natural join?

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

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

IF2240 – Basis Data
SQL (Part 3)

KNOWLEDGE & SOFTWARE ENGINEERING

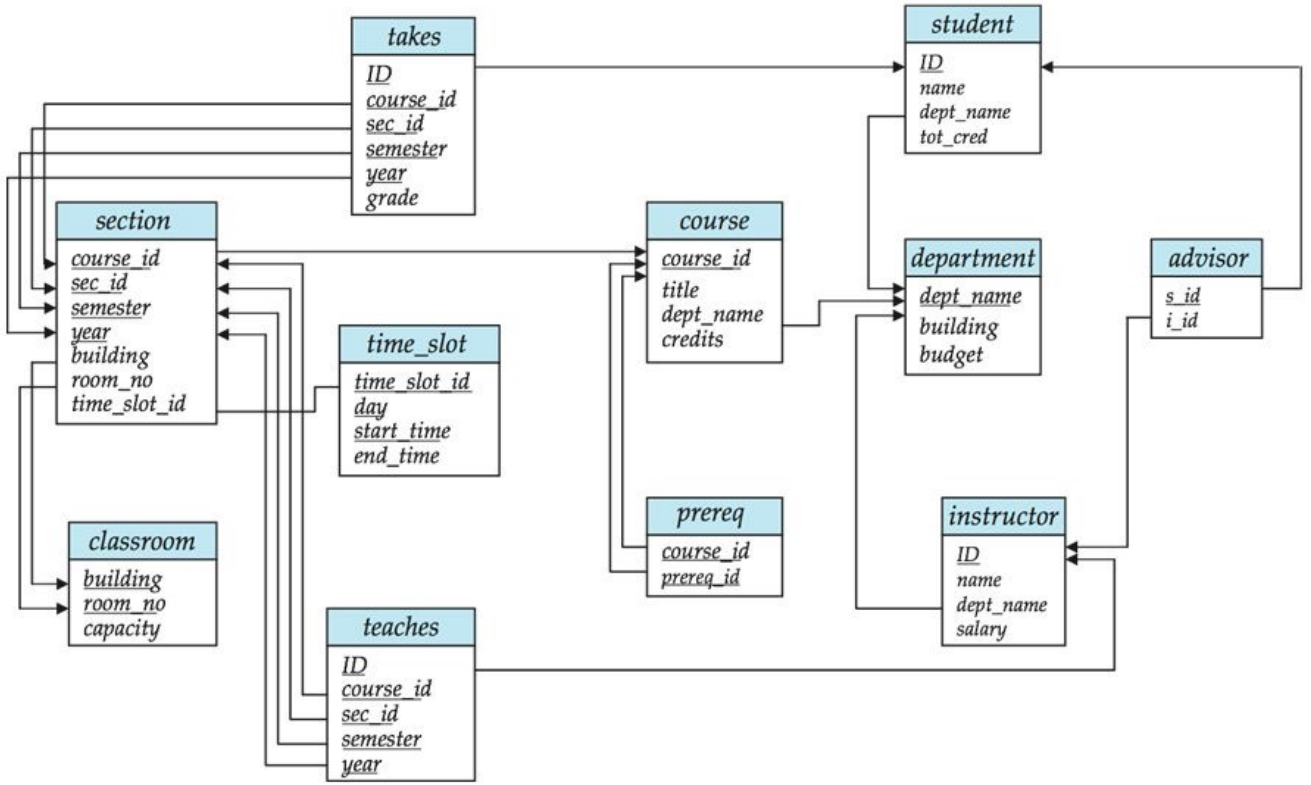


Sumber

Silberschatz, Korth, Sudarshan: "Database System Concepts", 7th Edition

- Chapter 3 : Introduction to SQL
- Chapter 4 : Intermediate SQL

Schema Diagram



Nested Subqueries

SQL provides a mechanism for the nesting of subqueries. A **subquery** is a **select-from-where** expression that is nested within another query.

The nesting can be done in the following SQL query

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

as follows:

- **Where clause:** P can be replaced with an expression of the form:

  B <operation> (subquery)

  B is an attribute and <operation> to be defined later.
- **From clause:** r_i can be replaced by any valid subquery
- **Select clause:**

  A_i can be replaced be a subquery that generates a single value.

Nested Query in Where Clause

- Set Membership (IN): to check whether an attribute value is a member of the result set produced by a subquery.
- Set Comparison (<relational operator> SOME/ALL): to evaluate whether an attribute value satisfies the <relational operator> condition against some or all tuples in the subquery result.
- Test of Empty Relation (EXISTS): to determine whether a subquery returns any tuples (i.e., whether the subquery produces a result).
- Test for Absence of Duplicates (UNIQUE): to check whether the tuples in the subquery result are unique (i.e., no duplicate values exist).

Set Membership

Find courses offered in Fall 2017 and in Spring 2018

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

Find courses offered in Fall 2017 but not in Spring 2018

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

Set Membership (Cont.)

Name all instructors whose name is neither "Mozart" nor Einstein"

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

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

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

Note: Above query can be written in a much simpler manner.
The formulation above is simply to illustrate SQL features

Set Comparison – "some" Clause

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

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

Same query using > **some** clause

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

Definition of "some" Clause

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

(5 < **some** | 0 / 5 / 6 |) = true (read: 5 < some tuple in the relation)

(5 < **some** | 0 / 5 |) = false

(5 = **some** | 0 / 5 |) = true

(5 ≠ **some** | 0 / 5 |) = true (since 0 ≠ 5)

(= **some**) ≡ **in**
However, (≠ **some**) ≡ **not in**

Set Comparison – "all" Clause

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

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

Definition of "all" Clause

F <comp> **all** $r \Leftrightarrow \forall\ t \in r$ (F <comp> t)

(5 < **all** {0, 5, 6}) = false

(5 < **all** {6, 10}) = true

(5 = **all** {4, 5}) = false

(5 ≠ **all** {4, 6}) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**
However, (= **all**) ≠ **in**

Test for Empty Relations

The **exists** construct returns the value **true** if the argument subquery is nonempty.

exists $r \Leftrightarrow r \neq \emptyset$

not exists $r \Leftrightarrow r = \emptyset$

Use of "exists" Clause

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

```
select course_id
from section as S
where semester = 'Fall' and year = 2017 and
        exists  (select *
                    from section as T
                    where semester = 'Spring' and year= 2018
                            and S.course_id = T.course_id);
```

Correlation name – variable S in the outer query

Correlated subquery – the inner query

Use of "not exists" Clause

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

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

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

Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$

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

Test for Absence of Duplicate Tuples

The **unique** construct tests whether a subquery has any duplicate tuples in its result.

The **unique** construct evaluates to "true" if a given subquery contains no duplicates .

Find all courses that were offered at most once in 2017

```
 select T.course_id
from course as T
where unique ( select R.course_id
                        from section as R
                        where T.course_id= R.course_id
                                and R.year = 2017);
```

Subqueries in the From Clause

SQL allows a subquery expression to be used in the **from** clause

Find the average instructors' salaries of those departments where the average salary is greater than $42,000."

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

Note that we do not need to use the **having** clause

Another way to write above query

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

With Clause

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

Find all departments with the maximum budget

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

Complex Queries using With Clause

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

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

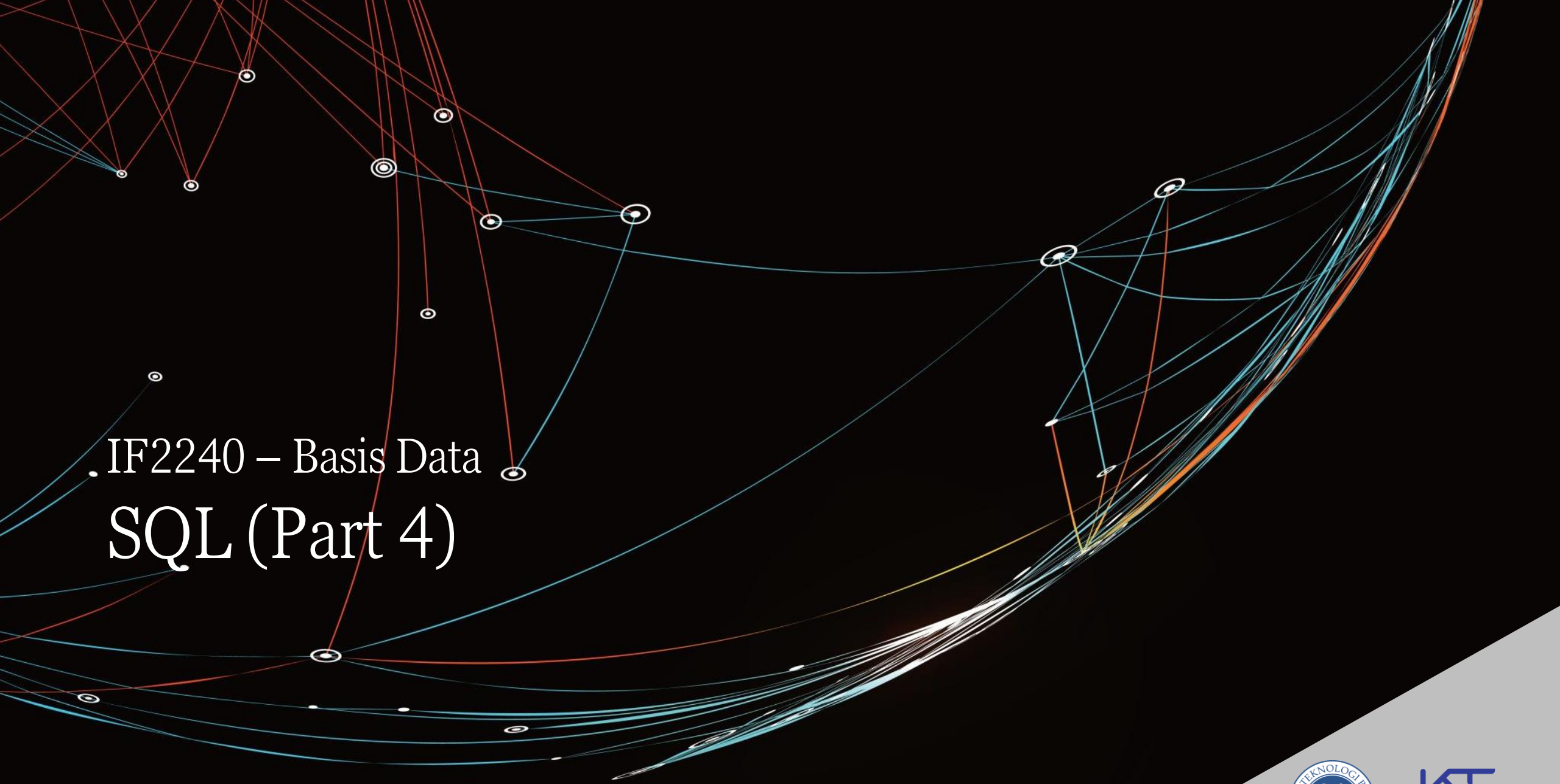
Scalar Subquery

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

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

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

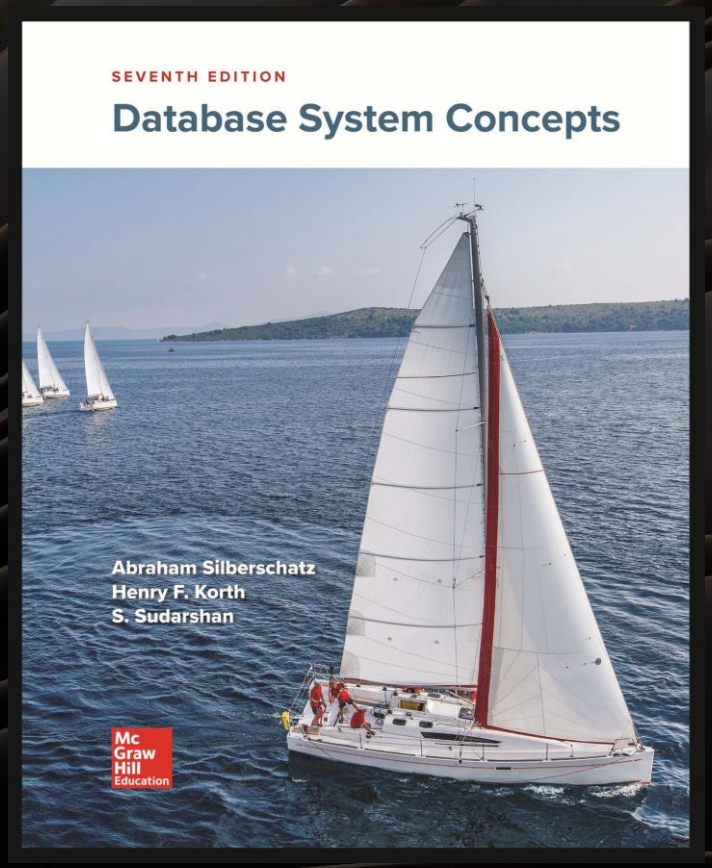
Runtime error if subquery returns more than one result tuple



IF2240 – Basis Data

SQL (Part 4)

KNOWLEDGE & SOFTWARE ENGINEERING

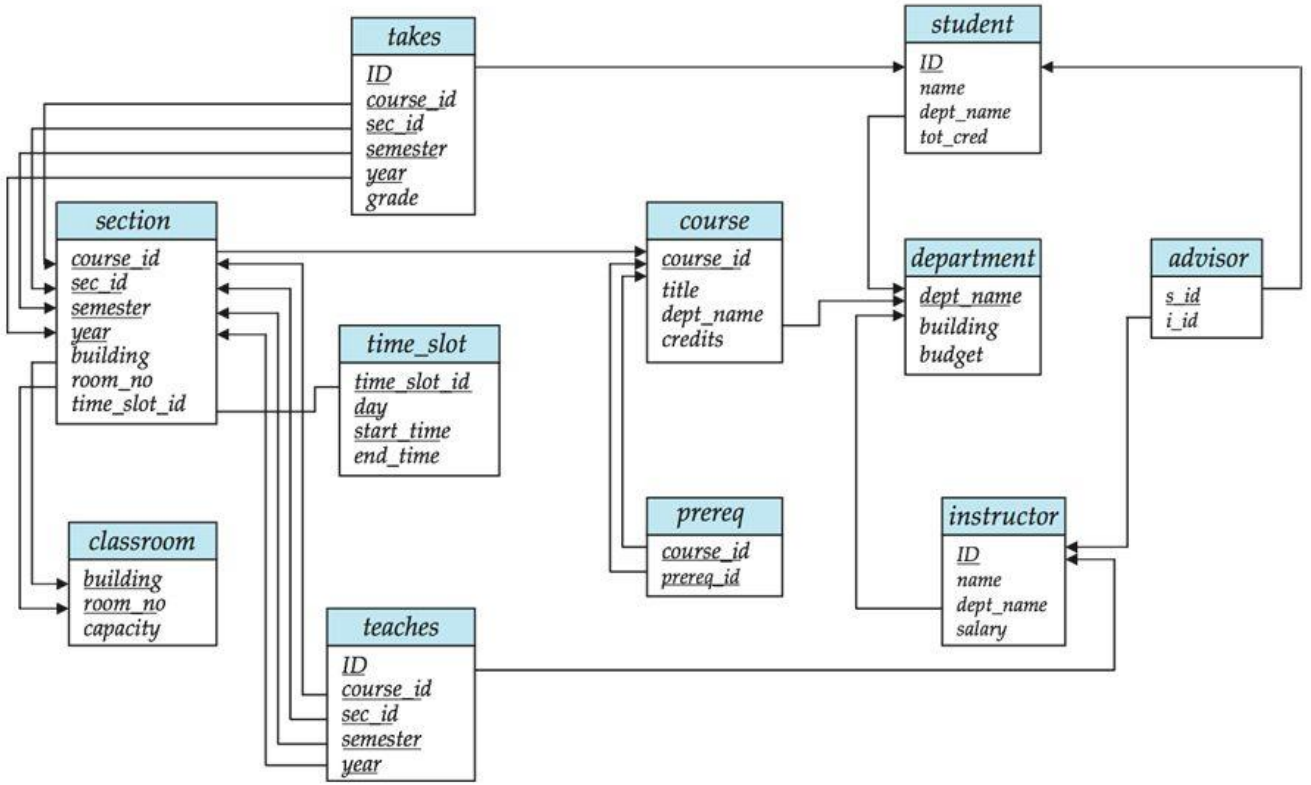


Sumber

Silberschatz, Korth, Sudarshan: "Database System Concepts", 7th Edition

- Chapter 3 : Introduction to SQL
- Chapter 4 : Intermediate SQL

Schema Diagram



Database System Concepts - 6th Edition 2 ©Silberschatz, Korth and Sudarshan

View

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 instructor's name and department, but not the salary. This person should see a relation described, in SQL, by

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

A **view** provides a mechanism to hide certain data from the view of certain users.

Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

View Definition

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

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

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

Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.

View definition is not the same as creating a new relation by evaluating the query expression

- Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

View Definition and Use

A view of instructors without their salary

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

Find all instructors in the Biology department

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

Create a view of department salary totals

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

Views Defined Using Other Views (1/2)

One view may be used in the expression defining another view

A view relation v_1 is said to ***depend directly*** on a view relation v_2 if v_2 is used in the expression defining v_1

A view relation v_1 is said to ***depend on*** view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2

A view relation v is said to be ***recursive*** if it depends on itself.

Views Defined Using Other Views (2/2)

```
create view physics_fall_2017 as
   select course.course_id, sec_id, building, room_number
   from course, section
   where course.course_id = section.course_id
            and course.dept_name = 'Physics'
            and section.semester = 'Fall'
            and section.year = '2017';


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

Materialized Views

Certain database systems allow view relations to be physically stored.

- Physical copy created when the view is defined.
- Such views are called **Materialized view**:

If relations used in the query are updated, the materialized view result becomes out of date

- Need to **maintain** the view, by updating the view whenever the underlying relations are updated.

Modification of the Database

Modification of the Database

Deletion of tuples from a given relation.

Insertion of new tuples into a given relation

Updating of values in some tuples of a given relation

Deletion

delete from *<table_name>*
where *<condition>*;

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.

conditions involving tuples from other relations are expressed using subquery.

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

Deletion (Cont.)

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

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

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

Insertion

insert into *<table_name>* [(*<attributes_list>*)]
values (*<attributes_values>*);

Add a new tuple to *course*

insert into *course*
values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

or equivalently

insert into *course* (*course_id*, *title*, *dept_name*, *credits*)
values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

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

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

Insertion (Cont.)

insert into <*table_name*> [(<*attributes_list*>)]
<*query*>;

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

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

The select from where statement is evaluated fully before any of its results are inserted into the relation.
Otherwise queries like

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

would cause problem

Updates

```
update <table_name>
set <attribute_name> = <expression>
where <condition>;
```

Give a 5% salary raise to all instructors

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

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

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

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

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

Updates (Cont.)

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

- Write two update statements:

```
update instructor
    set salary = salary * 1.03
    where salary > 100000;
update instructor
    set salary = salary * 1.05
    where salary <= 100000;
```

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

Case Statement for Conditional Updates

Same query as before but with case statement

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

Updates with Scalar Subqueries

Recompute and update tot_creds value for all students

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

Sets tot_creds to null for students who have not taken any course

Instead of sum(credits), use:

```
case
 when sum(credits) is not null then sum(credits)
 else 0
end
```

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 into the instructor relation

- Must have a value for salary.

Two approaches

- Reject the insert
- Insert the tuple

('30765', 'Green', 'Music', null)

into the instructor relation

Some Updates Cannot be Translated Uniquely

```
create view instructor_info as
        select ID, name, building
        from instructor, department
        where instructor.dept_name = department.dept_name;

insert into instructor_info
            values ('69987', 'White', 'Taylor');
```

Issues

- Which department, if multiple departments in Taylor?
- What if no department is in Taylor?

And Some Not at All

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

What happens if we insert

('25566', 'Brown', 'Biology', 100000)

into history_instructors?

View Updates in SQL

Most SQL implementations allow updates only on simple views

- The from clause has only one database relation.
- The select clause contains only attribute names of the relation, and does not have any expressions, aggregates, or distinct specification.
- Any attribute not listed in the select clause can be set to null
- The query does not have a group by or having clause.

Data Definition Language

Data Definition Language

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

1. The schema for each relation.
2. The type of values associated with each attribute.
3. The Integrity constraints
4. The set of indices to be maintained for each relation.
5. Security and authorization information for each relation.
6. The physical storage structure of each relation on disk.

Domain Types in SQL (1/2)

char(n). Fixed length character string, with user-specified length *n*.

varchar(n). Variable length character strings, with user-specified maximum length *n*.

int. Integer (a finite subset of the integers that is machine-dependent).

smallint. Small integer (a machine-dependent subset of the integer domain type).

numeric(p,d). Fixed point number, with user-specified precision of *p* digits, with *d* digits to the right of decimal point. (ex., **numeric**(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)

real, double precision. Floating point and double-precision floating point numbers, with machine-dependent precision.

float(n). Floating point number, with user-specified precision of at least *n* digits.

Domain Types in SQL (2/2)

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

Create Table Construct

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

create table *r*

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

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

Example:

create table *instructor* (

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

Integrity Constraints in Create Table

Types of integrity constraints

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

SQL prevents any update to the database that violates an integrity constraint.

Example:

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

And a Few More Relation Definitions

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

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

And more still

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

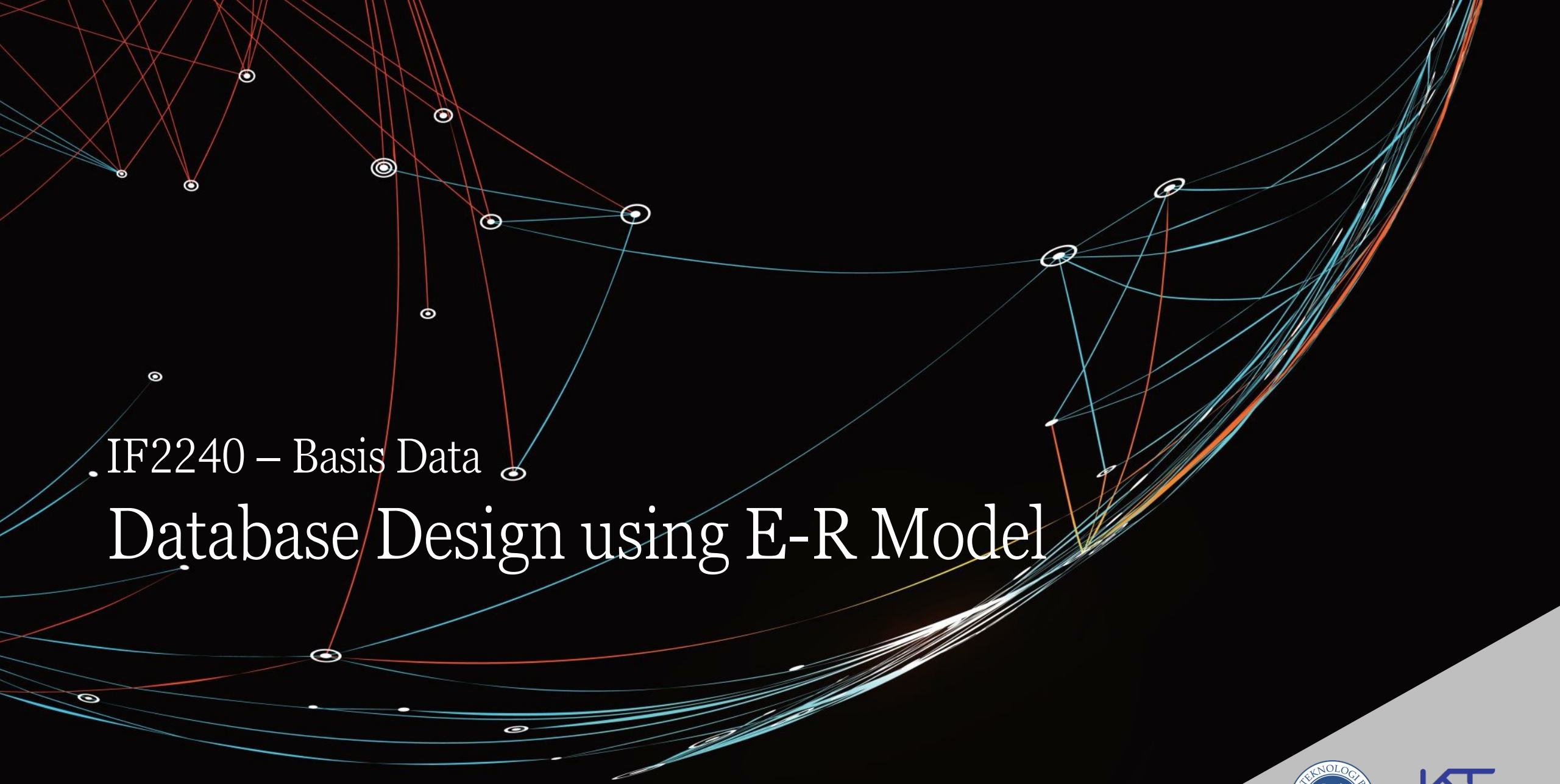
Updates to tables

Drop Table

- drop table *r*

Alter

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

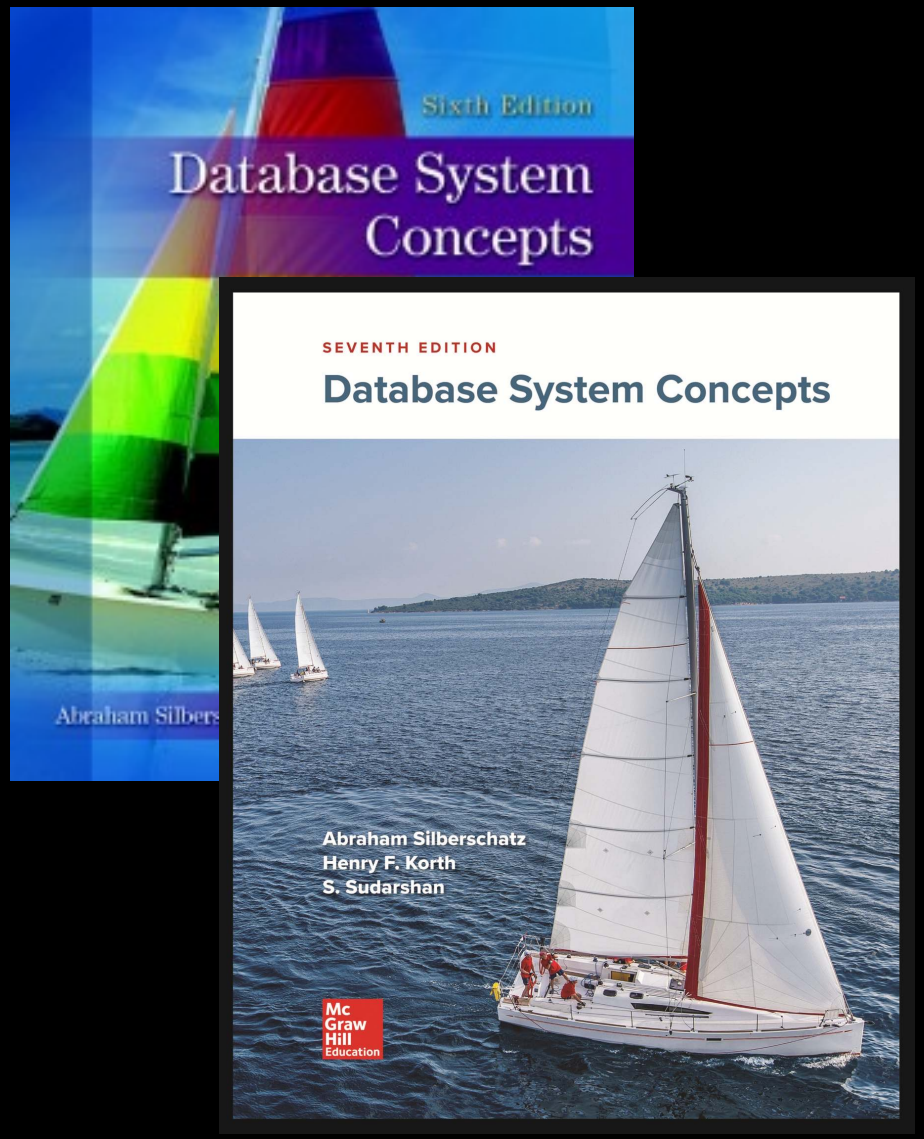


IF2240 – Basis Data

Database Design using E-R Model

Modified form Silberschatz's slides, Database System Concept, 7th edition

KNOWLEDGE & SOFTWARE ENGINEERING



References

Abraham Silberschatz, Henry F. Korth, S. Sudarshan : "**Database System Concepts**", 6th Edition

- Chapter 7: Database Design and the E-R Model

Abraham Silberschatz, Henry F. Korth, S. Sudarshan : "**Database System Concepts**", 7th Edition

- Chapter 6: Database Design using E-R Model

Design Phases

Initial phase

characterize fully the data needs of the prospective database users.

Second phase

choosing a data model

Final Phase

Moving from an abstract data model to the implementation of the database

- Logical Design – Deciding on the database schema.
- Physical Design – Deciding on the physical layout of the database

Design Alternatives

In designing a database schema, we must ensure that we avoid two major pitfalls:

Redundancy: a bad design may result in repeat information Redundant representation of information may lead to data inconsistency among the various copies of information

Incompleteness: a bad design may make certain aspects of the enterprise difficult or impossible to model.

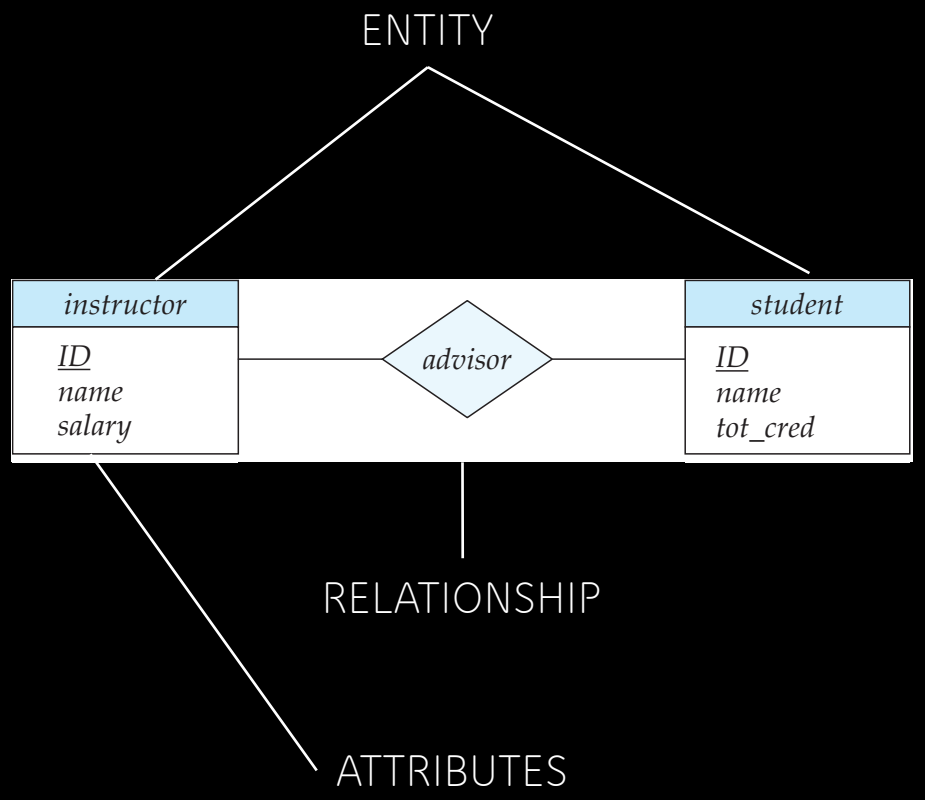
Design Approaches

Entity Relationship Model (covered in this material)

- Models an enterprise as a collection of *entities* and *relationships*
  - Entity: a "thing" or "object" in the enterprise that is distinguishable from other objects (Described by a set of *attributes)*
- Relationship: an association among several entities
- Represented diagrammatically by an *entity-relationship diagram:*

Normalization Theory (next topic)

- Formalize what designs are bad, and test for them



ER model -- Database Modeling

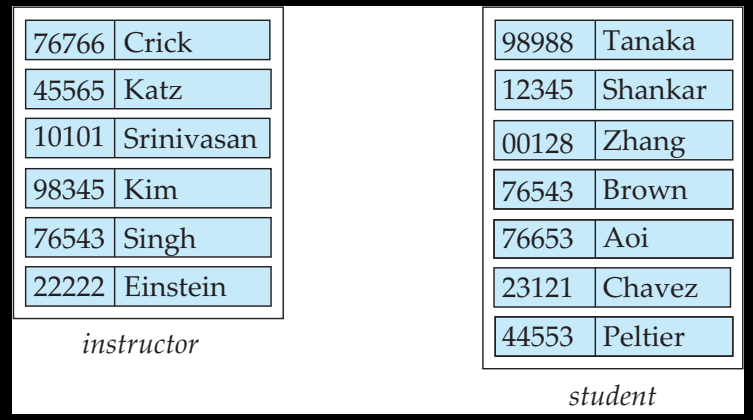
The ER data model was developed to facilitate database design by allowing specification of an **enterprise schema** that represents the overall logical structure of a database.

The ER data model employs three basic concepts:

- entity sets,
- relationship sets,
- attributes.

The ER model also has an associated diagrammatic representation, the **ER diagram**, which can express the overall logical structure of a database graphically.

Entity Sets

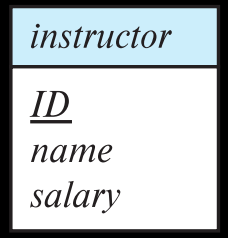


76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

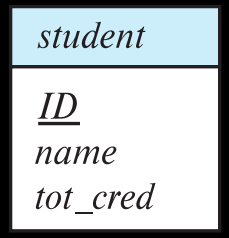
instructor

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

student



instructor
ID
name
salary



student
ID
name
tot_cred

Entity sets can be represented graphically as follows:

- Rectangles represent entity sets.
- Attributes listed inside entity rectangle
- Underline indicates primary key attributes

An **entity** is an object that exists and is distinguishable from other objects.

- Example: specific person, company, event, plant

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

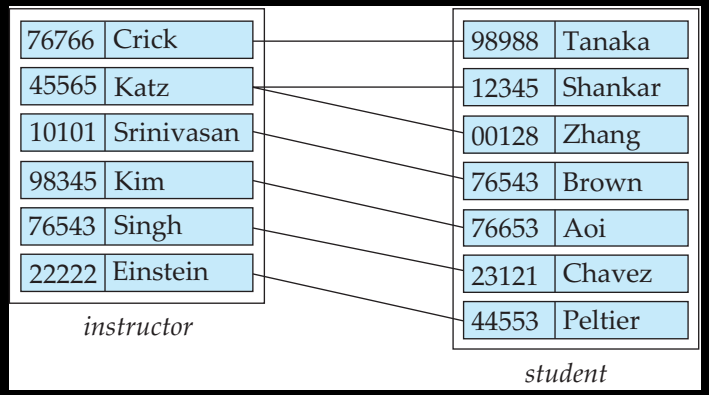
An entity is represented by a set of attributes; i.e., descriptive properties possessed by all members of an entity set.

- Example:

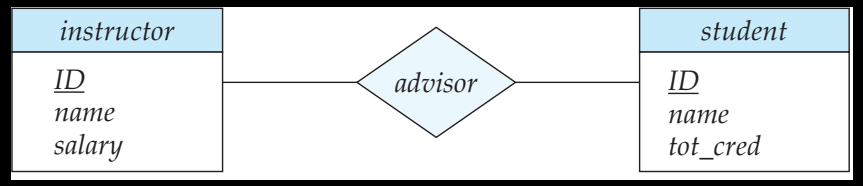
  instructor = (ID, name, salary)
  course= (course_id, title, credits)

- *Domain* – the set of permitted values for each attribute

A subset of the attributes form a **primary key** of the entity set; i.e. uniquely identifying each member of the set.



Example: we define the relationship set *advisor* to denote the associations between students and the instructors who act as their advisors.



Diamonds represent relationship sets.

Relationship Sets

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

Example:
44553 (Peltier) *student* entity — ***advisor*** relationship set — 22222 (Einstein) *instructor* entity

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

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

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

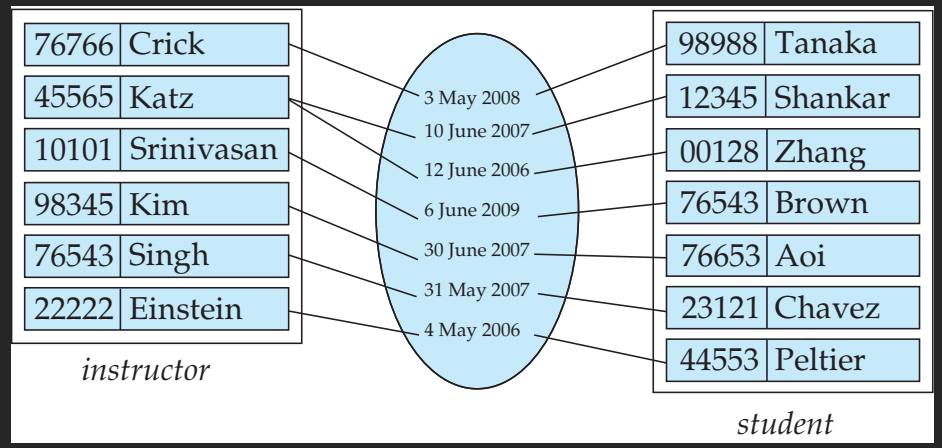
- Example:

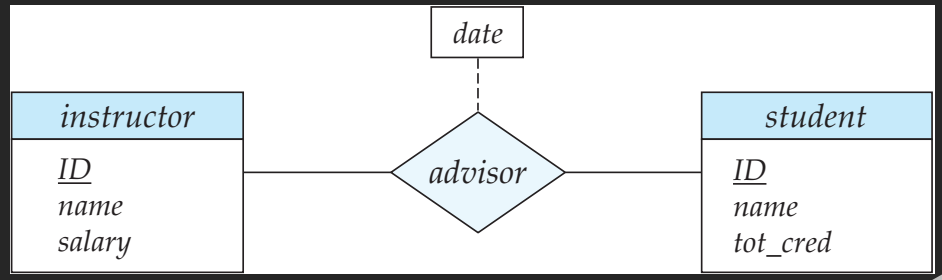
$$(44553,22222) \in advisor$$

Relationship Sets with Attributes

An attribute can also be associated with a relationship set.

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



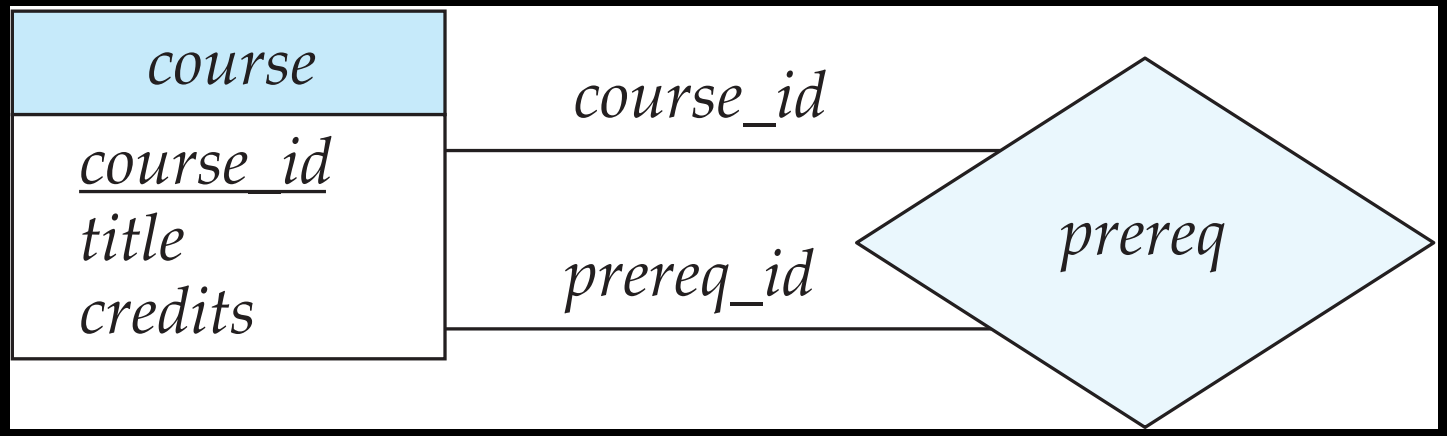


Roles

Entity sets of a relationship need not be distinct

- Each occurrence of an entity set plays a "role" in the relationship

The labels "*course_id*" and "*prereq_id*" are called **roles**.



Degree of a Relationship Set

Binary Relationship

- Involve two entity sets (or degree two). Most relationship sets in a database system are binary.

Non-Binary Relationship

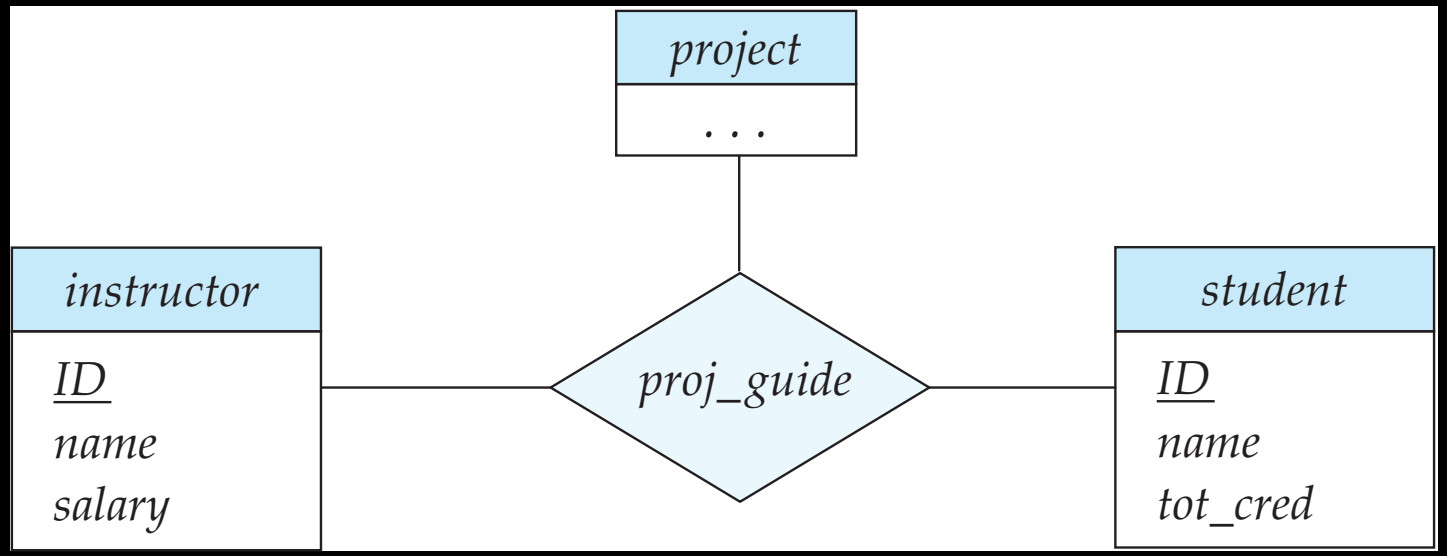
- Relationships between more than two entity sets are rare.
  - Example: *students* work on research *projects* under the guidance of an *instructor*. Relationship *proj_guide* is a ternary relationship between *instructor*, *student*, and *project*

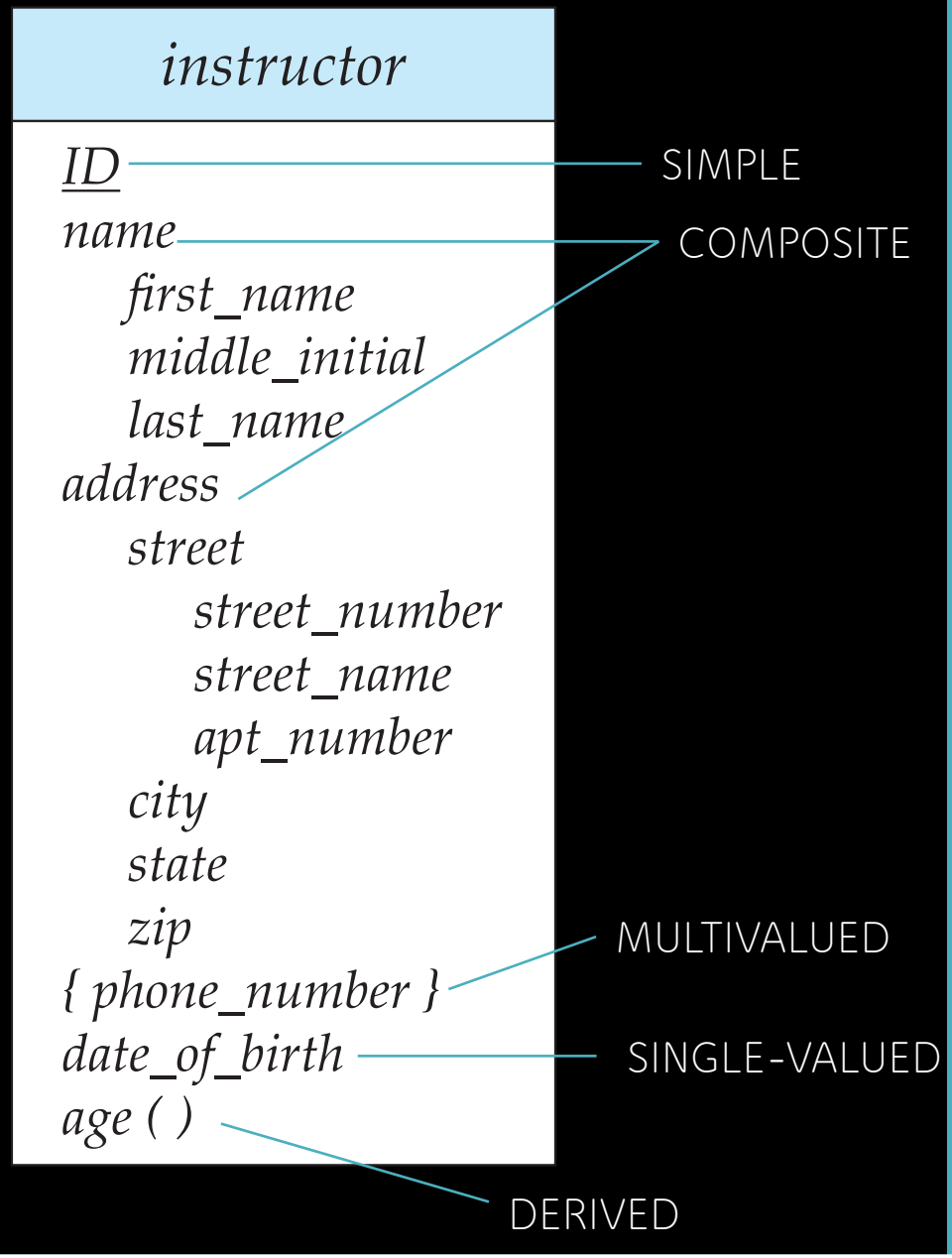
Non-binary Relationship Sets

Most relationship sets are binary

There are occasions when it is more convenient to represent relationships as non-binary.

E-R Diagram
with a Ternary Relationship





Complex Attributes

Attribute types:

- **Simple** and **composite** attributes.
- **Single-valued** and **multivalued** attributes
  - Example: multivalued attribute: *phone_numbers*
- **Derived** attributes
  - Can be computed from other attributes
  - Example: age, given date_of_birth

KNOWLEDGE & SOFTWARE ENGINEERING

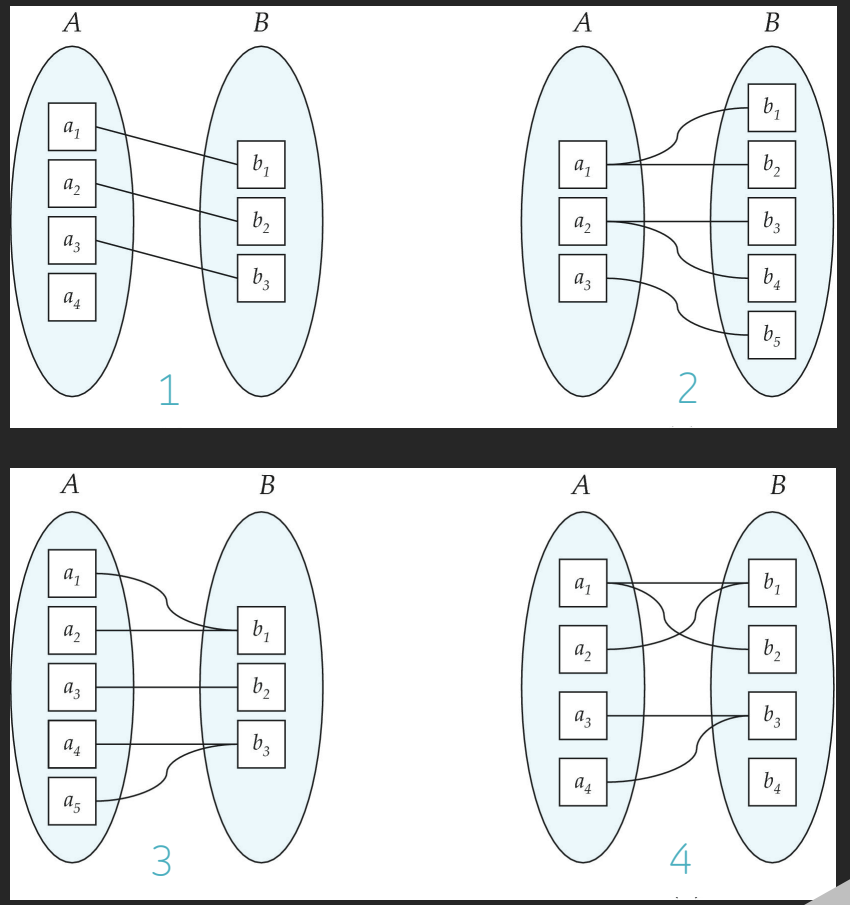
Mapping Cardinality Constraints

Express the number of entities to which another entity can be associated via a relationship set.

Most useful in describing binary relationship sets.

For a binary relationship set the mapping cardinality must be one of the following types:

1. One to one
2. One to many
3. Many to one
4. Many to many



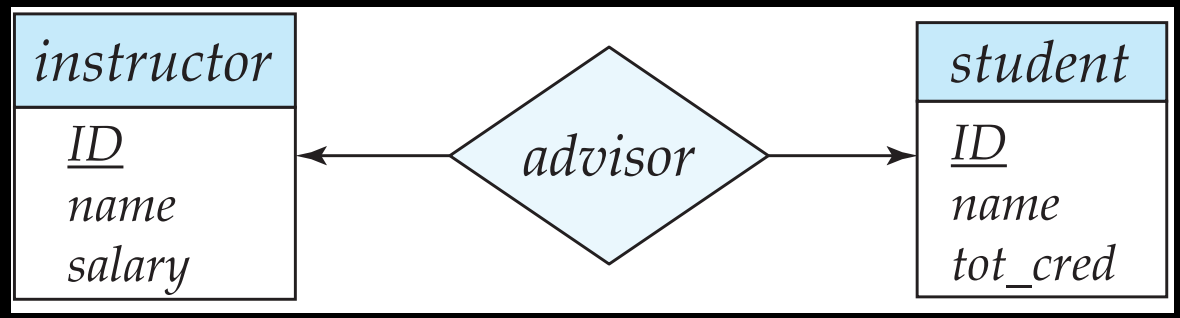
Note: Some elements in *A* and *B* may not be mapped to any elements in the other set

Representing Cardinality Constraints in ER Diagram (1/2)

We express cardinality constraints by drawing either a directed line (→), signifying "one," or an undirected line (–), signifying "many," between the relationship set and the entity set.

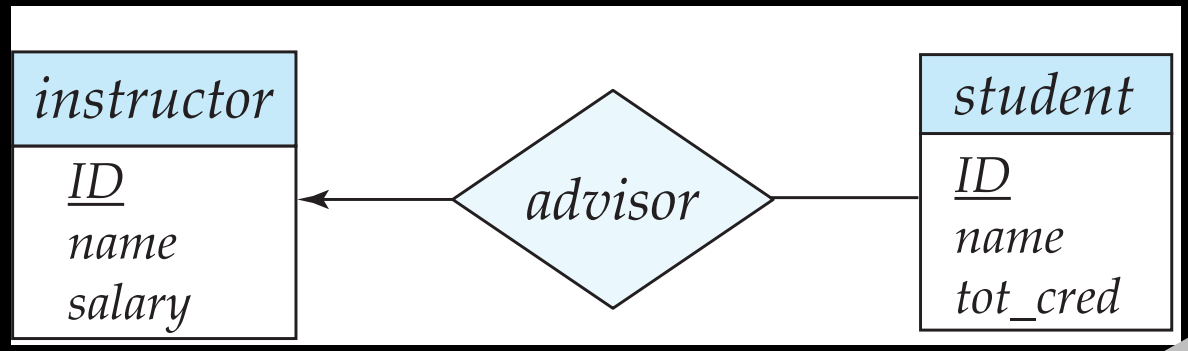
One-to-one relationship between an *instructor* and a *student* :

- 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-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,

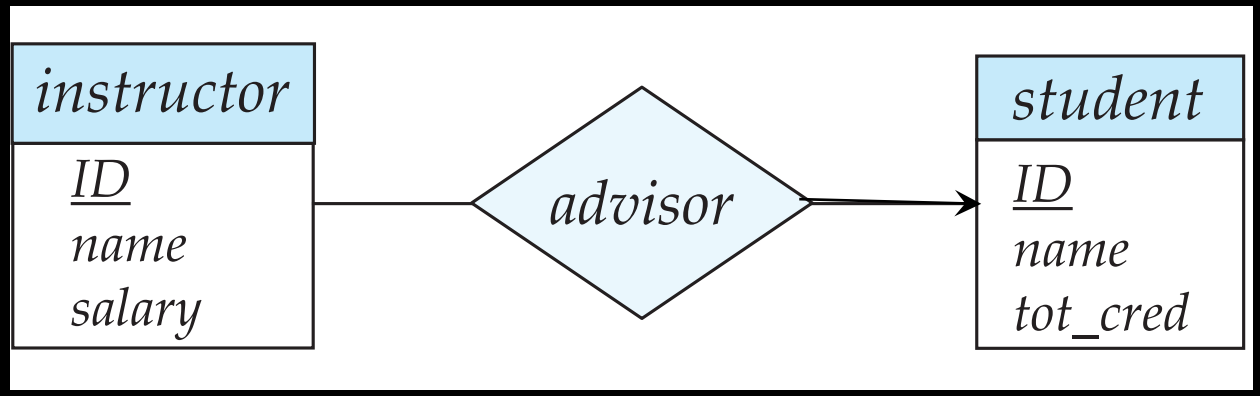


Representing Cardinality Constraints in ER Diagram (2/2)

We express cardinality constraints by drawing either a directed line (→), signifying "one," or an undirected line (–), signifying "many," between the relationship set and the entity set.

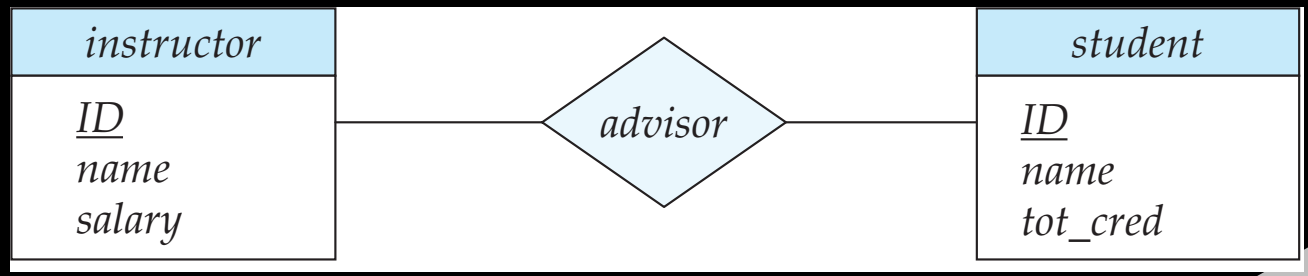
Many-to-one relationship between an instructor and a student,

- an instructor is associated with at most one student via advisor,
- and a student is associated with several (including 0) instructors via advisor



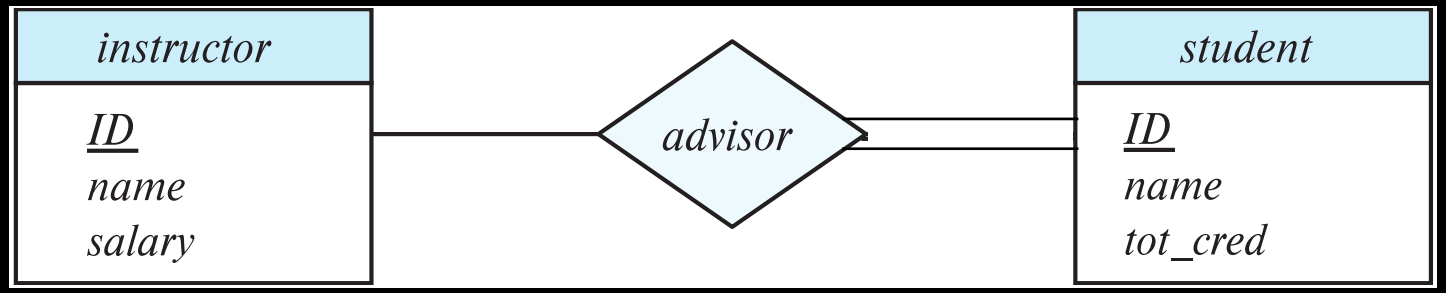
Many-to-many relationship between an instructor and a student

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



Total and Partial Participation

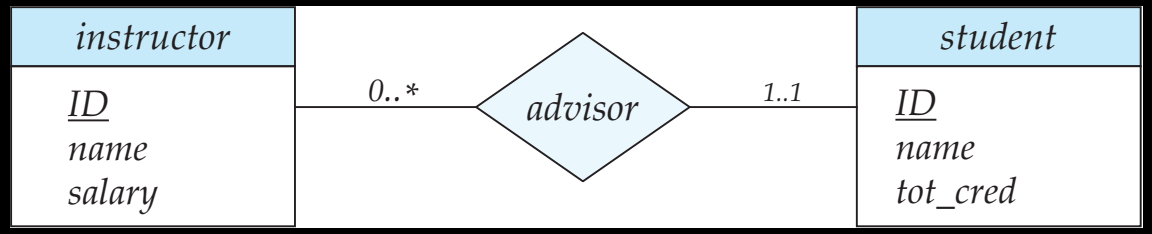
- **Total participation** (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set
- **Partial participation**: some entities may not participate in any relationship in the relationship set



- Example : participation of *student* in *advisor* relation is total
  - every *student* must have an associated instructor
- Example : participation of *instructor* in *advisor* relation is partial
  - some *instructor* may not have an associated student

Notation for Expressing More Complex Constraints

- A line may have an associated minimum and maximum cardinality, shown in the form *l..h*, where *l* is the minimum and *h* the maximum cardinality
  - A minimum value of 1 indicates total participation.
  - A maximum value of 1 indicates that the entity participates in at most one relationship
  - A maximum value of * indicates no limit.
- Example



  - Instructor can advise 0 or more students. A student must have 1 advisor; cannot have multiple advisors

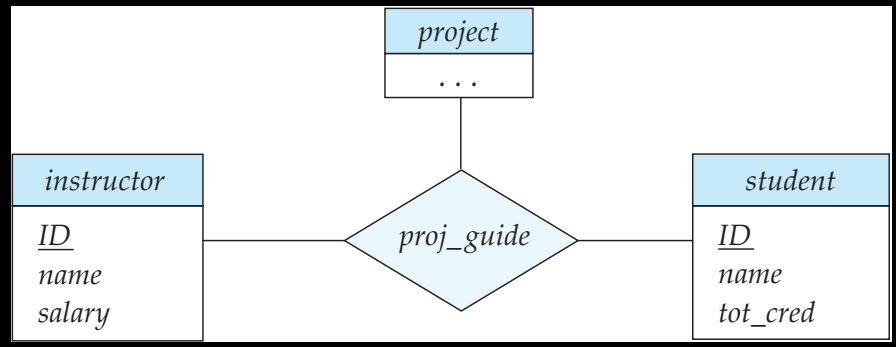
Cardinality Constraints on Ternary Relationship

We allow at most one arrow out of a ternary (or greater degree) relationship to indicate a cardinality constraint

For example, an arrow from *proj_guide* to *instructor* indicates each student has at most one guide for a project

If there is more than one arrow, there are two ways of defining the meaning.

- For example, a ternary relationship *R* between *A*, *B* and *C* with arrows to *B* and *C* could mean
    1. Each *A* entity is associated with a unique entity from B and *C* OR
    2. Each pair of entities from (*A*, *B*) is associated with a unique *C* entity,
       and each pair (*A*, *C*) is associated with a unique *B*
- Each alternative has been used in different formalisms
- To avoid confusion, we outlaw more than one arrow



Primary Key

Primary keys provide a way to specify how entities and relations are distinguished. We will consider:

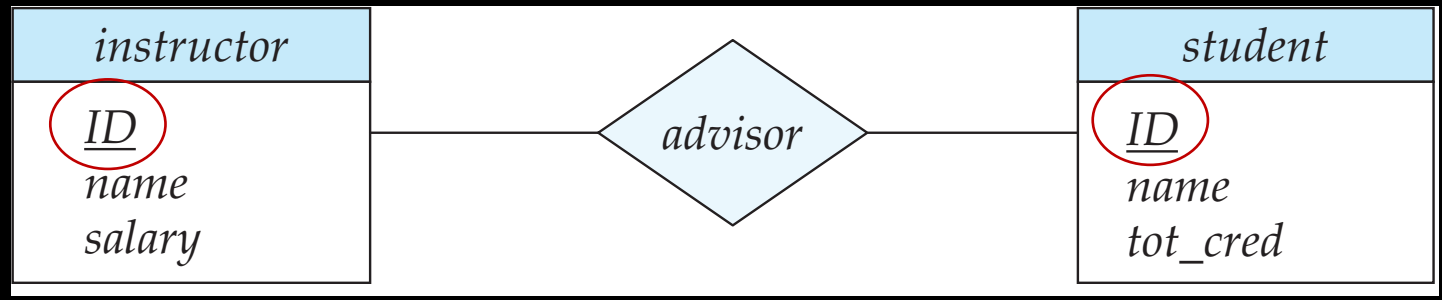
- Entity sets
- Relationship sets.
- *later on.. Weak entity sets*

By definition, individual entities are distinct.

The values of the attribute values of an entity must be such that they can uniquely identify the entity.

- No two entities in an entity set are allowed to have exactly the same value for all attributes.

A key for an entity is a set of attributes that suffice to distinguish entities from each other



Primary Key

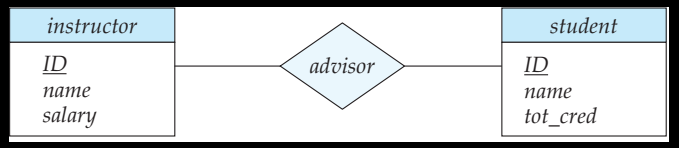
Primary keys provide a way to specify how entities and relations are distinguished. We will consider:

- Entity sets
- Relationship sets
- *later on.. Weak entity sets*

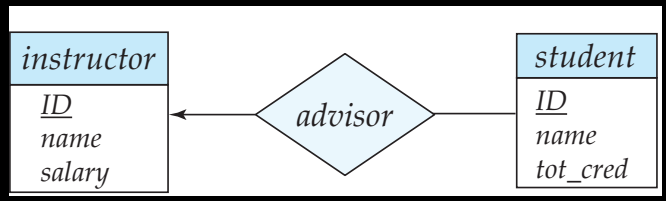
To distinguish among the various relationships of a relationship set we use the individual primary keys of the entities in the relationship set.

The choice of the primary key for a relationship set depends on the mapping cardinality of the relationship set.

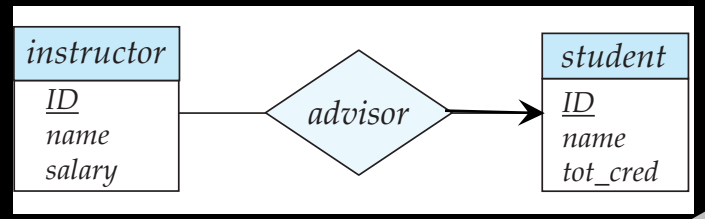
Many-to-Many relationships. The preceding union of the primary keys is a minimal superkey and is chosen as the primary key.



One-to-Many relationships. The primary key of the "Many" side is a minimal superkey and is used as the primary key.



Many-to-one relationships. The primary key of the "Many" side is a minimal superkey and is used as the primary key.



One-to-one relationships. The primary key of either one of the participating entity sets forms a minimal superkey, and either one can be chosen as the primary key.