



Technical University of Denmark

DTU Compute

Department of Applied Mathematics and Computer Science

02170 Database Systems Course

Course Introduction, Spring 2023

© Anne Haxthausen

These slides have been prepared by Anne Haxthausen. A few parts have been borrowed with permission from Flemming Schmidt.

Course Introduction

- Teachers and Students
- Database Systems: What and Why
- Course and Learning Objectives
- Course Prerequisites
- Course Material
- Course Activities and Assessment
- Plans
- Your Duty to Read Messages
- Study Advice
- If You Have Questions

Teachers

- Course responsible and main lecturer

- Associate Professor Anne Haxthausen, DTU Compute
- Background
 - MSc in Applied Mathematics, DTU 1985
 - PhD in Computer Science, DTU 1989
 - Software Engineer at Dansk Datamatik Center and Computer Resources International 1988 – 1994
 - Researcher & teacher at DTU since 1995



Teaching Assistants

- Aryan Mirzazadeh <s204489@student.dtu.dk> Windows 10
- Frederik Munk Svanholm Frost <s184182@student.dtu.dk> Windows 10 + ((Mac))
- Kasper Telkamp Nielsen <s170397@student.dtu.dk> Windows 10 + Linux: Ubuntu
- Kevin Miras Moore <s204462@student.dtu.dk> Mac + Windows
- Noah Bastian Christiansen <s184186@student.dtu.dk> Windows + Linux
- Nojan Rezvani <s204426@student.dtu.dk> Windows in February
- Óli Kárason Mikkelsen <s174269@student.dtu.dk> Windows 11
- Sofie-Amalie Petersen <s174255@student.dtu.dk> Mac
- Mathilde Elia <s215811@student.dtu.dk> Windows only first week



Aryan



Frederik



Kasper



Kevin



Noah



Nojan



Óli



Sofie-Amalie

Students

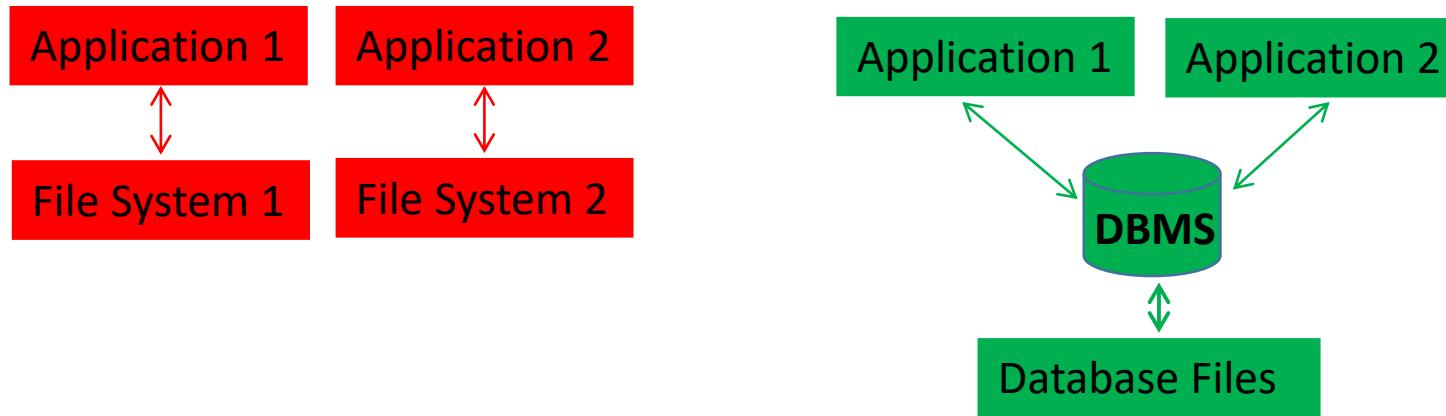
- approx. 270 students from 26 different DTU study lines + other universities + continuing education

Database Systems: What

- A Database System contains data and programs:
 - A Database is a collection of structured, interrelated data.
 - A Database Management System (DBMS) is a collection of programs used to access and manipulate data in a database.
 - DBMS examples: MariaDB, MySQL, Oracle, IBM DB2, Microsoft Access, ...
- Database Systems are widely used:
 - Banking: Transactions
 - Airlines: Reservations, schedules
 - Universities: Registration, grades
 - Sales: Customers, products, purchases
 - Online retailers: Order tracking, customized recommendations
 - Manufacturing: Production, inventory, orders, supply chain
 - Human resources: Employee records, salaries, tax deductions
 - Internet: Google, Facebook, ...



Why Using Databases instead of File Systems?



- **Conventional file system** approach:
 - Applications store and access data **directly** in files.
 - Typically decentralized.
- **Advantages of using a database instead:**
 - Applications and data are separated. Data is centralized.
 - Easier data access and manipulation using a high level DB language.
 - Less risk of data duplication and data inconsistency.
 - Integrated handling of concurrent access from multiple users.
 - Integrated security and performance management.

Course and Learning Objectives

- Course objectives and scope
 - Give an introduction to *theory and usage* of databases.
 - Focus is on so-called *relational* databases.
- Learning objectives
 - Be able to *design and implement relational databases* from *data models*.
 - Be able to *use a widely used database language (SQL)* and *database management system (MariaDB)* to perform create, query and update operations on databases.
 - Be able to *understand tasks* normally performed by database designers, programmers and administrators.



Course Prerequisites

- **Discrete mathematics (01017/01019/01904):**
 - set theory and predicate logic.
- **Imperative programming:**
 - Variable and data types.
 - Methods/procedures/functions and parameter passing.
 - Statements (assignment, loops, branching statements, etc)
- **Algorithms and data structures:**
 - as taught in 02105
 - balanced trees as taught in 02110.

Course Material

- Mandatory Text Book:



- **Database Systems** Custom Edition for Course 02170.
 - Is an extract of *Database System Concepts* by Silberschatz, Korth & Sudarshan, McGraw-Hill 6. Edition, 2011.
 - Can be bought in the DTU book store **Polyteknisk Boghandel**.
 - Price: 247,50 kr for print book and 179 kr for e-book
 - Possible to order online.
 - **Typo list** can be found on DTU Learn -> Content.
- **Tools guides** can be found on DTU Learn -> Content.
- **Slides (incl. exercises), videos and solutions to exercises:**
 - can be found on DTU Learn -> Content where they are organized according to the week in which they should be used;
 - slides will be made available not later than the evening before use;
 - solutions will be made available at the end (11:45) of their exercise session.

Course Activities and Assessment

■ Activities:

- **Lectures** most (but not all) Thursday 8:00-9:50.
 - Format (*physical* or *streaming* or *pre-recorded video* or *none*): see on DTU Learn Content under the course week number.
 - Reading stuff and slides (and sometimes videos) can be found on DTU Learn Content under the course week number.
- **Exercises** 10:00-11:45, where the TAs will advice you.
 - Exercises can be found in the slides on DTU Learn Content under the week number.
 - Work together in groups.
 - Solutions are provided at 11:45 on DTU Learn under the week number.
- **Mandatory group project** where you and your team mates will develop a database. Groups of 5 persons.
- **Written exam.**

■ Assessment:

- An *approved group project* is mandatory for participation in the written examination!
- The *final mark* is the mark from the *written examination*.

Tentative Lecture Plan

Date	Topic	Book Chapters	Lecture Format
02.02	0. Course Introduction.	1.1-1.6	Physical + Zoom
02.02	1. The Relational Model.	2.1-2.7	Physical + Zoom
02.02	1.5 Database Tools		Physical + Zoom
09.02	2. Introductory SQL.	3.1-3.10 (part 1)	See on Learn->Week2
16.02	3. Introductory SQL.	3.1-3.10 (part 2)	See on Learn->Week3
21.02	Deadline for Group Registration		See on Learn->Week4
23.02	4. Intermediate SQL.	4.1-4.7	See on Learn->Week5
02.03	5. Advanced SQL.	5.1-5.3, relevant parts of 5.7	Only pre-recorded video
09.03	6. Entity-Relationship Diagrams.	7.1-7.7, (7.9), 7.10-7.11	See on Learn->Week6
16.03	7. Mandatory Group Project.		No lecture
23.03	8. Mandatory Group Project.		No lecture
28.03	Deadline for Group Report Delivery		
30.03	9. Relational Algebra. Relational Calculi.	6.1-6.4	See on Learn->Week9
06.04	Easter holidays.		
13.04	10. Normalization.	8.1-8.8 (not 8.4.5), 8.10	See on Learn->Week10
20.04	11. Indexing & Hashing.	11.1-3 (except 11.3.3-11.3.4), 11.5-6, 11.8, 11.10-11	See on Learn->Week11
27.04	12. Exam Preparation.	Old exam questions.	See on Learn->Week12
04.05	13. Exam preparation.	Old exam questions.	No lecture
In May	Written Examination.		

- A *detailed plan* for each week can be found on DTU Learn Content.
- Will be updated during the course. So check it out each week.

Plans for 8:00-9:50 Today

- **Lecture 8:00 - ...:**

1. This Course Introduction (this slide set 0)
2. The Relational Model (slide set 1)
3. Database Tools (slide set 1.5)

Slides are on DTU Learn Content under Week 1. **Videos** will be provided (later).

Exercises 10:00 – 11:45 Today

- Exercise description can be found on DTU Learn Content under Week 1:
 1. Install and try the database tools to be used in this course.
 2. Demo Exercises and Exercises in the Relational Model. Solutions at 11:00.
- Take place in building 303A: Databars IT 46+47+48 and the hall outside these.
- Today (not needed next weeks), **organize your-self at**
 - "Linux tables" in IT-46
 - "Mac tables" in Hall East (the "tramway side" of the hall)
 - "Windows tables" in IT-47 + IT-48 + Hall West (+ IT-46 if more space is needed)so you can also help each other with installations.

TA	IT-46 (38)	IT-47 (58)	IT-48 (38)	Hall East (96)	Hall West (50)
Aryan			Windows		
Frederik				Windows (((+ Mac)))	Windows
Kasper	Linux + Windows				
Kevin				Mac (+ Windows)	
Noah		Windows (+Linux)			
Nojan		Windows			
Oli					Windows
Sofie-Amalie				Mac	
Mathilde					Windows

About Group Registration

- We have 8 TAs for 270+ students, so you must work together and help each other in groups which can get then help from the TAs.
- You should form groups of **5 persons** for the mandatory group project.
- Look for group partners:
 - contact people you know, or
 - talk with people you meet at DTU in the exercise sessions, or
 - use the Discussion Forum "Look for Group Partners" on DTU Learn
- When you have found a group, you should **register** it on DTU Learn. Must be done by end of **Tuesday 21th of February**.

Your Duty to Read Messages

- It is your **mandatory** duty
 - to follow and read announcements on DTU Learn and mails. I may send important messages, e.g.
 - about changes in the time schedule and teaching format
 - info about the mandatory group project and the exam
- Please remember to enable notifications in DTU Learn.

Please remember to enable notifications in learn!
(from <https://learnsupport.dtu.dk/faq.php>)

▼ How do I make sure to get notified when my instructor posts new announcements in the course?

You can sign up to be notified via email about any new announcements in your course by turning on notifications in your profile on DTU Learn. To do this, log in to DTU Learn, click on your name in the top right corner. Click "Notifications." Check the boxes "Announcements - new announcement available" and "Announcements - announcement updated".

Study Advice

■ How to optimize your work

- 1) Attending *Lectures* will save you a great deal of time!
- 2) Solving *Demo Exercises* and *Exercises* will enhance learning!
- 3) Optimizing your *Group Project* will prepare you for your examination.
Project **must be passed** in order to attend the written exam!
- 4) Reading *Textbook Chapters* is needed and highly recommended!



If You Have Questions

- Questions to lectures are most welcome during the lecture question time.
- Otherwise, as this is a BIG class, **please first contact one of the teaching assistants**. If they can't answer, they will contact me.



Technical University of Denmark

DTU Compute

Department of Applied Mathematics and Computer Science

The Relational Model

A Data Model for Relational Databases

©Anne Haxthausen and Flemming Schmidt

These slides have been prepared by Anne Haxthausen, partly reusing some slides by Flemming Schmidt, which again partly reused some slides by Silberschatz, Korth. Sudarshan, 2010.

Contents

- **Basic Concepts**
 - database, DBMS
 - data model
 - the Relational Model: relations/tables, relation schema, relation instance, domains, NULL values, keys, database schema diagram.
- **Database Languages (based on relational model)**
 - Basic concepts and SQL
- **Demo Exercises**
- **Exercises**

Databases and Data Models

- A **database** is a collection of structured, interrelated data.
 - Example: A university database could contain info about (1) **entities** like students and courses and (2) **relations** between the entities like which courses students take.
- A **database management system (DBMS)** is a collection of programs used to access and manipulate data in a database.
- Each database system has an underlying logical data model.
- A **data model** is a way to organise the data in a database.

Examples of data models:

- **relational model:** the most commonly used, the one we use in 02170
- object-based data models
- network model (old)
- hierarchical model (old)
- ...

The Relational Model

- Founders of the Relational Model turned the IT world upside down
 - They started making a great effort *to use mathematics as a foundation* for the Relational Model. Then they used best practice for implementation.
- The Relational Model is simple and based on set theory
 - All the data is logically structured in *relations (tables)* representing *entities* and their *relationships*.
 - A database is considered as *a set of relations*.
 - Relational Algebra offers simple operations on data.
- Databases based on the relational model are called *relational databases*.

Relations - Mathematical Definition

Mathematical Definition

- Let D_1, D_2, \dots, D_n be sets of values.
- The **Cartesian Product** $D_1 \times D_2 \times \dots \times D_n$ is the set of Tuples (a_1, a_2, \dots, a_n) where each value $a_i \in D_i$.
- A **relation** r over $D_1 \times D_2 \times \dots \times D_n$ is a subset of $D_1 \times D_2 \times \dots \times D_n$ i.e. a relation is a set of tuples (a_1, a_2, \dots, a_n) , where each $a_i \in D_i$.
Each tuple *represents* a **relationship** between the elements a_1, a_2, \dots, a_n
- A relation can be represented by a **table** with a row for each tuple.

Example:

The relation $\{ (apple, 5), (banana, 4) \}$ can be represented by

apple	5
banana	4

Relational Database Terminology

- A Relation is seen as a Table
 - with a name
 - with named **Attributes (columns)** of data
 - the rows are called **Tuples**
 - each Tuple (row) represents an entity or a relationship

- **Example:**

Relation **Instructor** has the Attributes: **InstID**, **InstName**, **DeptName** and **Salary**, and the first Tuple represents instructor Srinivasan, who works in the Computing Science Department, has an Instructor ID 10101, and a Salary of 65000.00.

Instructor

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

4	Attributes/columns
12	Tuples/rows

Terminology: Tables and Relations

- **Three ways of saying almost the same thing**
 - A Table of Cells stored in Rows and Columns
 - A Relation of Atoms stored in Tuples and Attributes
 - A Table of Data Elements stored in Rows and Attributes

Data in a ...	with ...	and ...
Table	Rows	Columns
Relation	Tuples	Attributes

Relation Schema and Relation Instance

- A Relation Schema $R(A_1, \dots, A_n)$
 - Defines the overall design of a relation.
 - Consists of
 - the name R of the relation, e.g. Instructor
 - the attribute names, A_1, \dots, A_n , e.g. InstID
 - a domain D_i for each attribute A_i
 - attributes constituting primary key are underlined
 - **Example:**
 $\text{Instructor}(\underline{\text{InstID}}, \text{InstName}, \text{DeptName}, \text{Salary})$
 - Is changed infrequently, if at all!
- A Relation Instance r of a relation R
 - $r \subseteq D_1 \times D_2 \times \dots \times D_n$
 - Is the data stored in the relation at a particular moment in time.
 - Over time the relation instance may change frequently!

Instructor

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

Domains

- A Domain (similar to type)
 - Is a Set of allowed Attribute Values
- Domain Examples
 - InstID: Integers from [10000, 99999]
 - DeptName: Elements from the set {Biology, Comp. Sci., Elec. Eng., Finance, History, Music, Physics}
 - Attribute Values must be atomic
- All Domains have a NULL value
 - A NULL value signifies a not existing value or a unknown value
 - To insert instructor Hansen with ID 79797 into the relation Instructor without knowing his department and salary, in MySQL simply insert the tuple:
INSERT Instructor VALUES ('79797', 'Hansen', NULL, NULL);

Instructor

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

Keys

Relation Schemas

Instructor(InstID, InstName, DeptName, Salary)
Department(DeptName, Building, Budget)

Instructor

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

What makes Tuples unique?

- In Instructor each tuple has a unique InstID
- In Department the DeptName value makes each tuple unique.

A Key

- Is the Attribute, or set of Attributes, that makes relation tuples unique.
- No two tuples have the same Key
- Sometimes it takes more than one attribute to make a Key. For example:
Classroom(Building, Room, Capacity)

Department

DeptName	Building	Budget
Biology	Watson	90000.00
Comp. Sci.	Taylor	100000.00
Elec. Eng.	Taylor	85000.00
Finance	Painter	120000.00
History	Painter	50000.00
Music	Packard	80000.00
Physics	Watson	70000.00

Keys

Let $R(A_1, A_2, \dots, A_n)$ be a relation schema and $K \subseteq \{A_1, A_2, \dots, A_n\}$, e.g. $K = \{A_2, A_5\}$.

- **(Super)key:**

- K is a **superkey** of R , if values for K are sufficient to identify a unique tuple of each permissible relation instance r of R . In other terms: no two rows have the same superkey.
- Example: $\{\text{InstID}\}$ and $\{\text{InstID}, \text{InstName}\}$ are both superkeys of Instructor.

- **Candidate Key:**

- A Superkey K is a **candidate key** if K is minimal.
- Example: $\{\text{InstID}\}$ is a candidate key for Instructor.

- **Primary Key (Constraints):**

- One of the candidate keys is selected by the DB designer to be the **primary key**.
- In the relational schema, attributes constituting the primary key are underlined, like in Instructor(InstID, InstName, DeptName, Salary).
- This implies a **constraint on the allowed relation instances**:
 - (1) no two rows may have the same value for the primary key,
 - (2) the primary key value must not be NULL.

- **Foreign Key (Constraints):** see next page.

Foreign Keys, Example

Let $R(A_1, A_2, \dots, A_n)$ be a relation schema, $K \subseteq \{A_1, A_2, \dots, A_n\}$

- K can be specified to be a **foreign key** referencing another relation R' , if K is a primary key of R' .
- This implies **a referential integrity constraint on the allowed relation instances r of R and r' of R' :** for any tuple in r , there must exist a tuple in r' having the same values for K .

Example:

- Given **Relation Schemas**
Instructor(InstID, InstName, DeptName, Salary)
Department(DeptName, Building, Budget)
- Attribute **DeptName** in relation **Instructor** is a **foreign key** referencing the **Department** relation.
- This implies **a referential integrity constraint on the allowed relation instances of Instructor and Department:** for any tuple in **Instructor**, there must exist a tuple in **Department** having the same values for **DeptName**.
- **Insert and delete violations:** Deleting the last tuple of **Department** or inserting a tuple having **DeptName Mathematics** in **Instructor** would break the constraint.

Instructor

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

Department

DeptName	Building	Budget
Biology	Watson	90000.00
Comp. Sci.	Taylor	100000.00
Elec. Eng.	Taylor	85000.00
Finance	Painter	120000.00
History	Painter	50000.00
Music	Packard	80000.00
Physics	Watson	70000.00

Relational Database Schemas and Diagrams

- A **Database Schema** consists of the relation schemas of all database tables.
- A **Database Schema Diagram** depicts the relation schemas + primary keys + foreign keys of all database tables.

Database Schema Diagram for a University

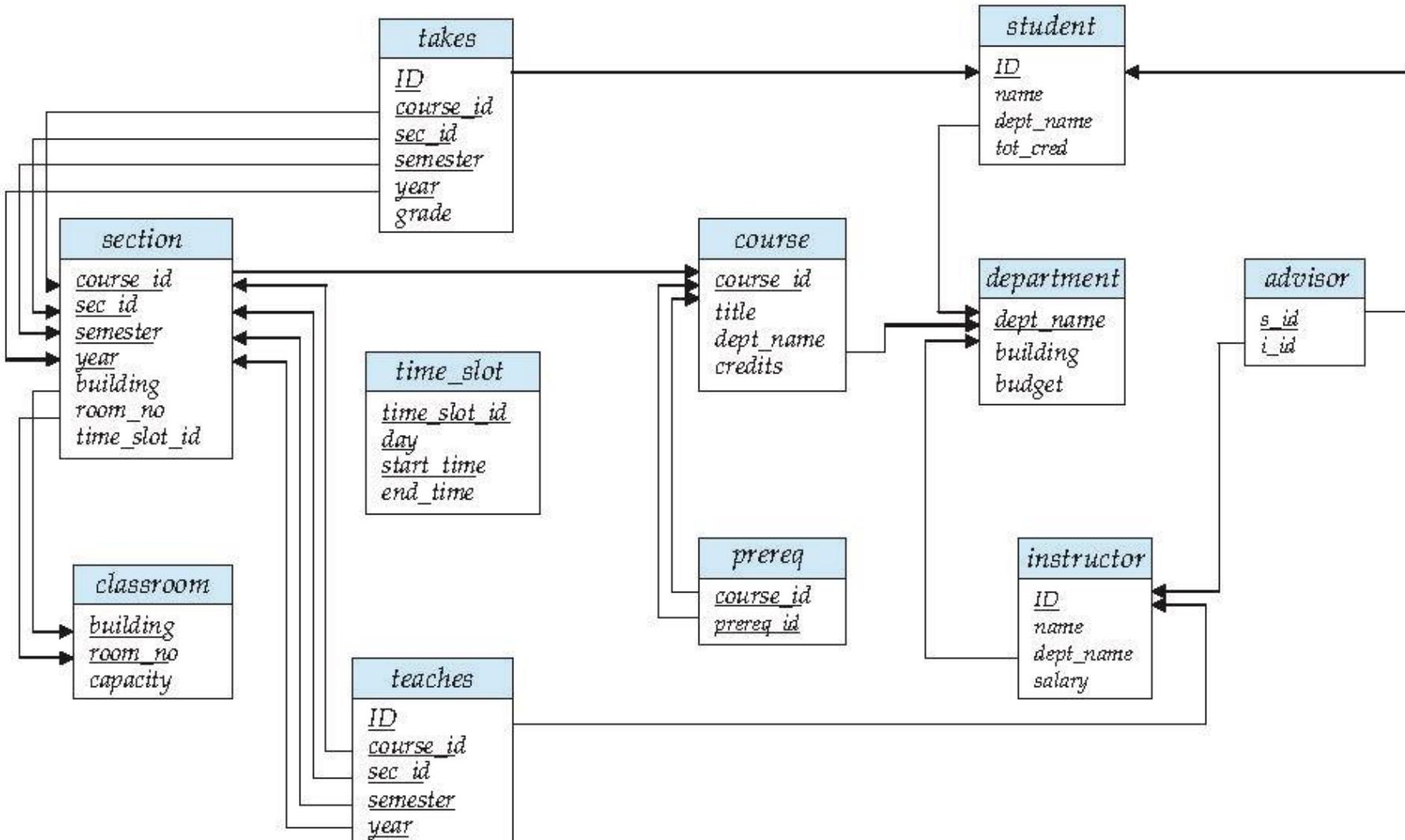


Table Names are shown in blue boxes. Primary Key Attributes are underlined.

Foreign Keys are shown with arrows.

Database Languages

- A **query language** is a language used to extract data from a database.
For **relational query languages**, the output is a table/relation.
- **SQL – Structured Query Language:** most widely used query language
 - Example of a query to find physics instructors with a salary < 80000:
SELECT InstID, InstName FROM Instructor
WHERE DeptName = 'Physics' AND Salary < 80000;
- **Formal Query Languages based on Mathematics:**
 - **Relational Algebra** with Relations as variables, for example:
$$\prod_{\text{InstID, InstName}} (\sigma_{\text{DeptName} = \text{'Physics'} \wedge \text{Salary} < 80000} (\text{Instructor}))$$
 - **Tuple Calculus** with Tuple Variables (t and s), for example:
$$\{t \mid \exists s \in \text{Instructor} (t[\text{InstID}] = s[\text{InstID}] \wedge t[\text{InstName}] = s[\text{InstName}] \wedge s[\text{DeptName}] = \text{'Physics'} \wedge s[\text{Salary}] < 80000)\}$$
 - **Domain Calculus with Domain Variables** (id, in, dn and sa):
$$\{< \text{id}, \text{in} > \mid \exists \text{dn}, \text{sa} (< \text{id}, \text{in}, \text{dn}, \text{sa} > \in \text{Instructor} \wedge \text{dn} = \text{'Physics'} \wedge \text{sa} < 80000)\}$$

SQL – Structured Query Language

- SQL is a special-purpose programming language designed for managing data stored in a relational database.
- Originally SQL was based upon *relational algebra and tuple calculus*.
- SQL was one of the first commercial languages for Edgar F. Codd's relational model, as described in his influential 1970 paper, "A Relational Model of Data for Large Shared Data Banks".
- SQL quickly became the most widely used database language.
- SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987.
- Since then, the standard has been revised to include a larger set of features. Despite the existence of such standards, though, most SQL code is not completely portable among different database systems without adjustments.

SQL Command Categories

- **Data Definition Language (DDL)**

CREATE DATABASE, DROP DATABASE,
CREATE TABLE, ALTER TABLE, DROP TABLE,
CREATE INDEX, ALTER INDEX, DROP INDEX,
CREATE VIEW and DROP VIEW

- **Data Manipulation Language (DML)**

INSERT, UPDATE and DELETE

- **Data Query Language**

SELECT (i.e. SELECT attributes FROM tables WHERE condition GROUP BY attributes)

- **Data Control Language**

CREATE USER, RENAME USER, DROP USER
ALTER PASSWORD, GRANT, REVOKE and CREATE SYNONYM

- **Data Administration Commands**

START AUDIT and STOP AUDIT

- **Transactional Control Commands**

SET TRANSACTION, COMMIT, ROLLBACK and SAVEPOINT



Schema & Instance in SQL Context

- Relation Schema

Instructor(InstID, InstName, DeptName, Salary)

- Used for creating an SQL Table:

```
CREATE TABLE Instructor (
    InstID      VARCHAR(5) PRIMARY KEY,
    InstName    VARCHAR(20),
    DeptName   VARCHAR(20),
    Salary      DECIMAL(8,2));
```

- Relation Instance

- Content of a relation Instructor shown in SQL:

```
SELECT InstID, InstName, DeptName, Salary FROM Instructor;
```

Instructor

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

SQL Data Definition Language

- **Data Definition Language DDL**

- Defines the Database Schema to the Data Dictionary. Example:

```
CREATE TABLE Instructor (
    InstID      VARCHAR(5),
    InstName    VARCHAR(20) NOT NULL,
    DeptName   VARCHAR(20),
    Salary      DECIMAL(8,2),
    PRIMARY KEY(InstID),
    FOREIGN KEY(DeptName) REFERENCES Department(DeptName)
    ON DELETE SET NULL);
```

- **Data Dictionary contains metadata (i.e. data about data)**

- Database relation schemas, authorization permissions, and more ...
 - Integrity constraints

Primary Key: InstID uniquely identifies instructors, each row has unique InstID!

Foreign Key: DeptName value in any Instructor row must appear in Department.

SQL Data Manipulation Language

- **Data Manipulation Language DML**
 - Language for manipulating data in the database
- **Like** for Instructor(InstID, InstName, DeptName, Salary):
 - Insert a new instructor
`INSERT Instructor VALUES ('79797', 'Hansen', 'Finance', NULL);`
 - Move 'Wu' to the Physics department and set salary to 95000
`UPDATE Instructor SET DeptName = 'Physics', Salary = 95000.00 WHERE InstID = 12121;`
 - Wu has left, delete him from the table
`DELETE FROM Instructor WHERE InstID = 12121;`

SQL Data Query Language

■ Data Query Language DQL

- Given Instructor(InstID, InstName, DeptName, Salary)
- Find the name of the instructor with InstID equal to 22222:
SELECT InstName FROM Instructor WHERE InstID = '22222';

InstName
Einstein

- Find the InstID and Building of instructors in the Physics department:
SELECT InstID, Building FROM Instructor, Department
WHERE Department.DeptName = 'Physics' AND
Instructor.DeptName = Department.DeptName;

InstID	Building
22222	Watson
33456	Watson

Long names when ambiguity
Short: AttributeName
Long: TableName.AttributeName



Summary

- **Basic Concepts:** database, DBMS, data model, relational model, relations/tables, relation schema, relation instance, domains, NULL values, keys, database schema diagram.
- **Language Concepts in SQL.**

Readings

- In Database Systems Concepts please read Chapters 1.1-1.6 & 2
- Pay special attention to the Summaries.

Demo Exercises



Demo Exercises clarify ideas and concepts from the Lecture to provide you with good Database Skills.

Discuss and do the Demo Examples.

Relational Model Terminology

1.1 Terminology

Make assumptions and describe the data below using Relational Terminology.

Student

StudID	StudName	Birth	DeptName	TotCredits
00128	Zhang	1992-04-18	Comp. Sci.	102
12345	Shankar	1995-12-06	Comp. Sci.	32
19991	Brandt	1993-05-24	History	80

Instructor

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
32343	El Said	History	60000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
83821	Brandt	Comp. Sci.	92000.00

Advisor

StudID	InstID
00128	45565
12345	10101

Department

DeptName	Building	Budget
Comp. Sci.	Taylor	100000.00
History	Painter	50000.00
Physics	Watson	70000.00

Database Scheme Diagram

1.2 Database Schema Diagram

Draw a Database Schema Diagram showing
Relations, Attributes, Primary Keys and
Foreign Keys using the Relation Schemas:

Student (StudID, StudName, Birth,
DeptName, TotCredits)

Instructor (InstID, InstName,
DeptName, Salary)

Advisor (StudID, InstID)

Department (DeptName, Building,
Budget)

Solutions to Demo Exercises



Relational Model Terminology

1.1 Terminology

Make assumptions and describe the data below using Relational Terminology.

Student

StudID	StudName	Birth	DeptName	TotCredits
00128	Zhang	1992-04-18	Comp. Sci.	102
12345	Shankar	1995-12-06	Comp. Sci.	32
19991	Brandt	1993-05-24	History	80

Instructor

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
32343	El Said	History	60000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
83821	Brandt	Comp. Sci.	92000.00

Advisor

StudID	InstID
00128	45565
12345	10101

Department

DeptName	Building	Budget
Comp. Sci.	Taylor	1000000.00
History	Painter	500000.00
Physics	Watson	700000.00

1. A Relational Database Instance!
2. With 4 Relations (tables) named Student, Instructor, Advisor and Department.
3. Relation Student has the Schema:
Student (StudID, StudName, Birth, DeptName, TotCredits)
defining the 5 Attributes (columns).
4. Relation Student has 3 Tuples (rows).
5. The Student Primary Key Attribute named StudID makes all Tuples in Student unique.
6. The Student Foreign Key called DeptName references the Department Relation.
7. Domain for TotCredits is Integers.
8. The Advisor Primary Key makes all Tuples unique. It is either <StudID> alone, meaning a student has just one instructor, or <StudID, InstID>, meaning that a student can have many advisors.
9. The Relation Advisor has two Foreign Keys: StudID referencing Student and InstID referencing Instructor.

Database Schema Diagram

1.2 Database Schema Diagram

Draw a Database Schema Diagram showing Relations, Attributes, Primary Keys and Foreign Keys using the Relation Schemas:

Student (StudID, StudName, Birth, DeptName, TotCredits)

Instructor (InstID, InstName, DeptName, Salary)

Advisor (StudID, InstID)

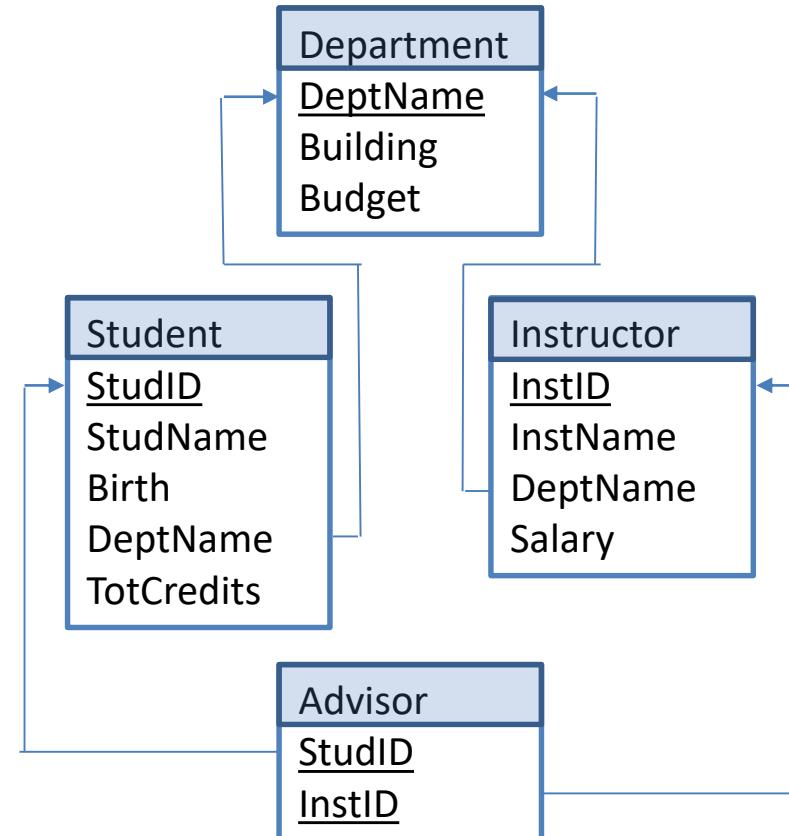
Department (DeptName, Building, Budget)

Database Schema Diagram

Write Relation Names in blue boxes.

Write Attribute Names below the relation name with Primary Key Attributes underlined.

Depict Foreign Keys by arrows.



Exercises



Please answer all exercises
to demonstrate your
Database Skills.

Pencil and Paper Exercises
Solutions are available at 11:00

Primary Keys and Foreign Keys

1.4 Primary Keys

Underline potential Primary Key attributes in the Relation Schemas:

Employee (FullName, Street, City)

Works (FullName, CompanyName, Salary)

Company (CompanyName, City)

1.5 Insert and Delete Violations

Give examples of insert and delete violations of the Foreign Key constraint.

Instructor

	InstID	InstName	DeptName	Salary
	12121	Wu	Finance	90000
	15151	Mozart	Music	40000
	22222	Einstein	Physics	95000

Department

	DeptName	Building	Budget
	Finance	Painter	120000
	Music	Packard	80000
	Physics	Watson	70000

Foreign Keys and Schema Diagrams

1.6 Foreign Keys

Banking Database Relation Schemas:

Branch(BName, BCity, Assets)

Customer(CName, CStreet, CCity)

Loan(LNumber, BName, Amount)

Borrower(CName, LNumber)

Account(ANumber, BName, Balance)

Depositor(CName, ANumber)

1.7 Database Schema Diagram

Draw a Database Schema Diagram for the Banking Database.

Check the table below and write the Foreign Keys and Referenced Relations.

Relation	Primary Keys	Foreign Keys	Referencing
Branch	BName		
Customer	CName		
Loan	LNumber		
Borrower	CName LNumber		
Account	ANumber		
Depositor	CName ANumber		



Technical University of Denmark

DTU Compute

Department of Applied Mathematics and Computer Science

Database Tools

Installation and Use

©Anne Haxthausen

Overview

- About MariaDB and MySQL Workbench
- Exercise:
 1. Install MariaDB and MySQL Workbench
 2. Try the Command Line Client
 3. Try the Workbench



About MariaDB



- Is a widely used *Relational Database Management System* supporting SQL.
- Is an open-source, community-developed fork of MySQL.
- Some *prominent users* : Google, Mozilla, Wikipedia, archlinux, RedHat, and Fedora.
- We will use it in this course.



- Online documentation:
<https://mariadb.com/kb/en/library/documentation/>

About MariaDB and its User Interfaces

■ MariaDB Server

- Receives SQL input and returns SQL output to other programs via Port 3306.



■ MySQL Workbench

- Enables editing of SQL queries in a **Graphical User Interface** to be sent to execution in the MariaDB Server.
- Entity-Relationship Diagrams for data modeling



■ MariaDB Command Line Client

- Enables editing of SQL queries in a **Command Line Interface**

■ Program Stack

MySQL Workbench

Command Line Client

DB Applications

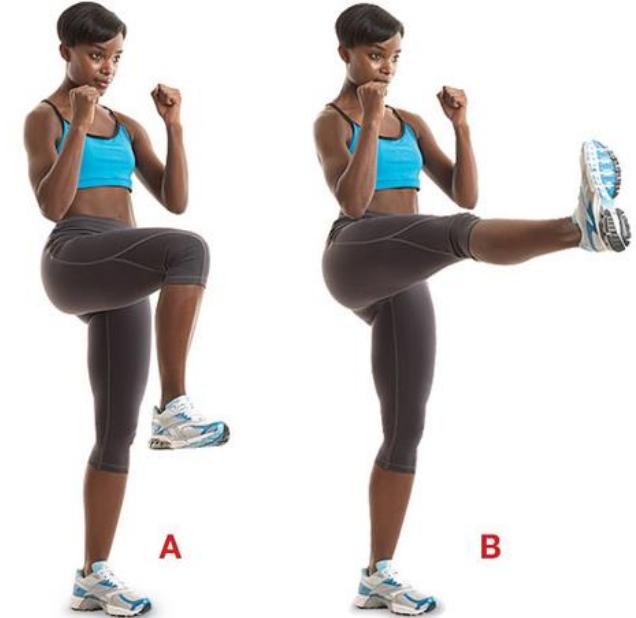
MariaDB Server (Port 3306)

Operating System (Windows 10 Pro)

Personal Computer Hardware (Lenovo T450)

Exercises

- Purpose:
to get the MariaDB server together with its Command Line Client and MySQL Workbench up and running on your PC.
- Exercises on the next pages:
 1. Install the Tools
 2. Try the Command Line Client
 3. Try the Workbench



1 Install the Tools

For Windows:

In the Course Content of DTU Learn, in the *Tools Guides* folder, there are installation guides for

1. Installing the MariaDB Server with the MySQL Command Line Client on Windows.
2. Installing the MySQL Workbench on Windows.

For Mac:

In the Course Content of DTU Learn, in the *Tools Guides* folder, there is a document with installation tips for Mac.

For Linux:

The installation procedure depends on the Linux distribution and package manager you are using.

Therefore no general recipe is given, but some tips can be found in the document

InstallationsOnLinux.pdf, which can be found in the Course Content of DTU Learn, in the *Tools Guides* folder. You might also use Google to find a solution.

2. Try the Command Line Client

Exercise: try the Command Line Client.

1. Download “FamilyDB.sql” file from the *Databases* folder in the Course Content of DTU Learn to a file, e.g. it could be `C:\Users\aeha\Documents\02170\Datasets\FamilyDB.sql`
This file contains SQL commands to create and populate a table with data.
2. Start MySQL Command Line Client and enter your server password:
 - Under **Windows** you start the tool from the start menu by selecting `MariaDB->MySQL Client (MariaDB ...)`
 - Under **Linux** and **Mac** you start the tool by the command: `mysql -u root -p` or `mariadb -u root -p`
3. Now you can enter commands on the command line after the `MariaDB [...]>` prompt.
Note: An SQL Statement is terminated with a “;”
4. Try the following to see what the system replies:

```
MariaDB [(none)]> \h
```

```
MariaDB [(none)]> create database FamilyDB;
```

```
MariaDB [(none)]> use FamilyDB;
```

```
MariaDB [FamilyDB]> source C:/Users/aeha/Documents/02170/Databases/FamilyDB.sql;
```

In some cases the slashes (/) in the path of the source command should be changed to backslashes (\).

```
MariaDB [FamilyDB]> show databases;
```

```
MariaDB [FamilyDB]> show tables;
```

```
MariaDB [FamilyDB]> select * from family;
```

```
MariaDB [FamilyDB]> exit;
```

2. Try the Command Line Client

Here you see the output from two of the commands:



The image shows a terminal window titled "Select MySQL Client (MariaDB 10.3 (...". The window displays the following MySQL command-line session:

```
MariaDB [FamilyDB]> show tables;
+-----+
| Tables_in_familydb |
+-----+
| family           |
+-----+
1 row in set (0.001 sec)

MariaDB [FamilyDB]> select * from family;
+-----+-----+
| PersonName | Birthday |
+-----+-----+
| Henry      | 19591006 |
| Lilly      | 19940224 |
+-----+-----+
2 rows in set (0.001 sec)

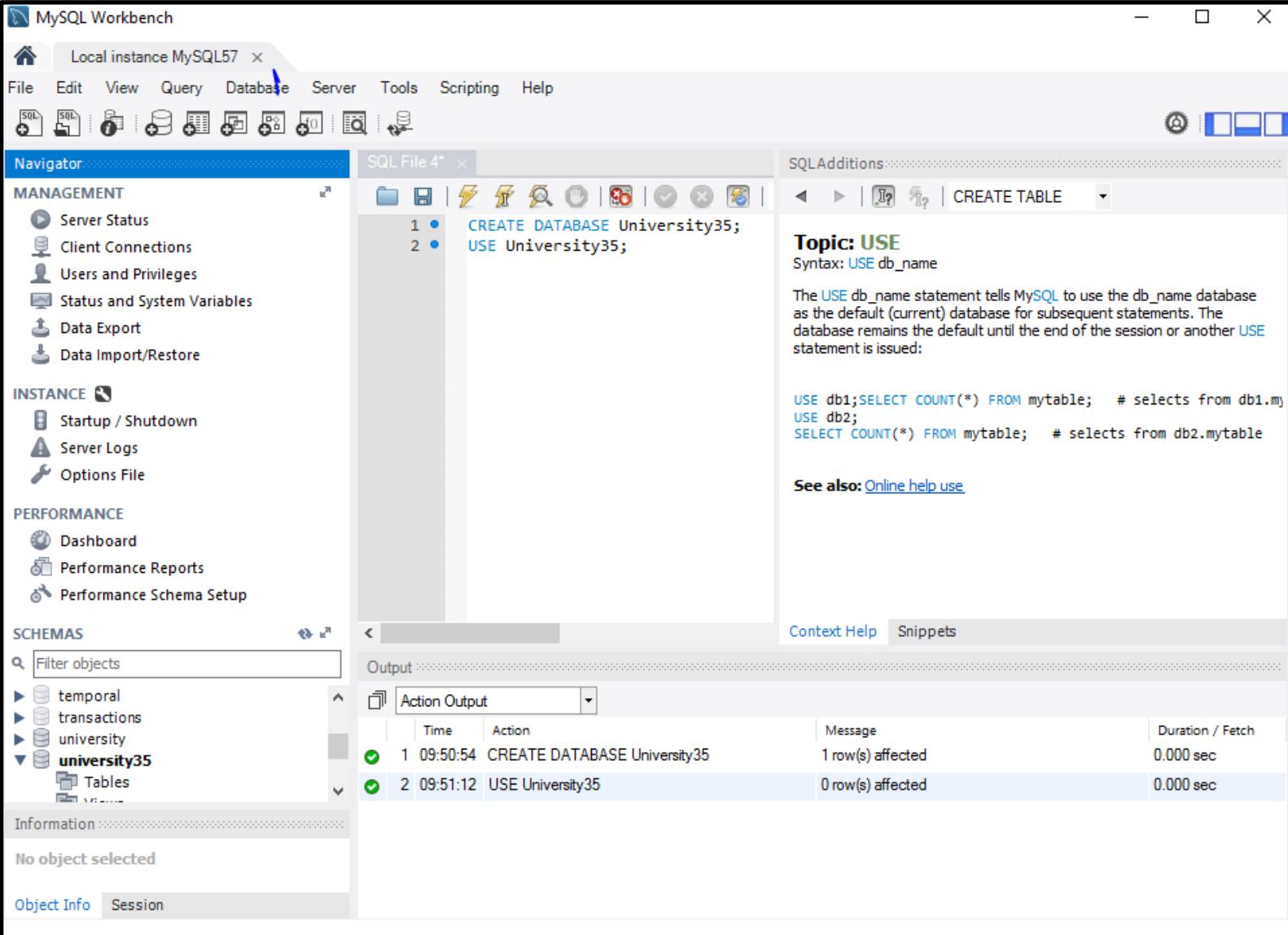
MariaDB [FamilyDB]>
```

3. Try MySQL Workbench

Exercise:

Try [MySQL Workbench](#) by following the guide “MySQL Workbench – Ultra-short Introduction” (in the file *WorkbenchUserGuide.pdf*) which can be found in the *Tools Guides* folder in the Course Content of DTU Learn.

MySQL Workbench



The screenshot shows the MySQL Workbench interface with the following details:

- Toolbar:** Includes icons for Home, SQL, Data, Schemas, Tables, Scripts, and Help.
- Navigator:** Lists categories like MANAGEMENT, INSTANCE, PERFORMANCE, and SCHEMAS. Under SCHEMAS, 'university35' is expanded, showing 'Tables' and 'Views'.
- SQL Editor:** Titled "SQL File 4*", contains the following SQL code:

```
1 CREATE DATABASE University35;
2 USE University35;
```
- SQLAdditions:** A tooltip for the "Topic: USE" command. It defines the USE statement as telling MySQL to use a specific database as the default (current) database for subsequent statements. It notes that the database remains the default until the end of the session or another USE statement is issued. It also shows examples of using USE with SELECT statements.
- Output:** Titled "Action Output", shows the results of the executed SQL statements:

Time	Action	Message	Duration / Fetch
1 09:50:54	CREATE DATABASE University35	1 row(s) affected	0.000 sec
2 09:51:12	USE University35	0 row(s) affected	0.000 sec

MySQL Workbench

MySQL Panels

- Navigator Panel: to start/stop the Server, ...
- SQL Panel: to write and execute SQL statements
- Output Panel: to show status (GYR) for each executed SQL statement
- SQL Additions Panel: to display SQL syntax and options
- Information Panel: to show information about an object or connection



Technical University of Denmark

DTU Compute

Department of Applied Mathematics and Computer Science

Introductory SQL, Part 1

Create Databases and Select Data

Part 1 of Chapter 3 in the Text Book

©Anne Haxthausen and Flemming Schmidt

These slides have been prepared by Anne Haxthausen, using a figure from Thorkjørn Konstantinovitz and partly reusing/modifying slides by Flemming Schmidt, which again partly reused some slides by Silberschatz, Korth. Sudarshan, 2010.

Contents

- University Database (running example)
- SQL Command Categories (book sec. 3.1)
- SQL Data Definition (book sec. 3.2)
- SQL Data Manipulation (book sec. 3.9, except parts using subqueries)
- SQL Data Queries (book sec. 3.3-3.4, except 3.4.2)
- Making New Tables From Old Tables
- Demo Exercises & Exercises



University Database

- Database example used throughout the textbook and in this course!

1. Statement of Requirements

- Contains details about entities and their relationship and related tasks needed to run a university. Often established via client interviews.

2. Database Schema

- Contains information of the database in a listing of Relation Schemas with attribute names and primary keys.

3. Database Schema Diagram

- Contains information of relations, attributes, primary keys and foreign keys in a diagram format.

4. SQL Data Definition

5. Database Instance

- Shows the relation contents at a specific moment in time

1 - Statement of Requirements

(see also Section 1.6.2 in the text book)

- **Based on client interviews**

The university is organized in named **departments**.

Departments have assigned yearly **budgets** and are located in **buildings**.

Each department employs named **instructors**.

Instructors have assigned **salaries**.

Each department has named **students**.

Each student has a **birthday**, total **credits** for passed examinations, and can have one department **advisor** assigned.

Each department offers named **courses**.

Each course is offered each **year** in defined **semesters**. Each offering of a course is called a **section** and is **taught** by instructors and **taken** by students, who gets a credit for passing the examination. Courses may have other courses as **prereq**. Sections are conducted in a weekly **timeslot**, in a specific campus **class room** (**building room**), which has a maximum student **capacity**.

And more about who uses the database and for what purpose...

2 - Database Schema

■ A set of Relation Schemas

Each with a Relation Name, Attribute Names and Primary Key Attributes underlined

Classroom(Building, Room, Capacity)

Department(DeptName, Building, Budget)

Course(CourseID, Title, DeptName, Credits)

Instructor(InstID, InstName, DeptName, Salary)

Section(CourseID, SectionID, Semester, StudyYear, Building, Room, TimeSlotID)

Teaches(InstID, CourseID, SectionID, Semester, StudyYear)

Student(StudID, StudName, Birth, DeptName, TotCredits)

Takes(StudID, CourseID, SectionID, Semester, StudyYear, Grade)

Advisor(StudID, InstID)

TimeSlot(TimeSlotID, DayCode, StartTime, EndTime)

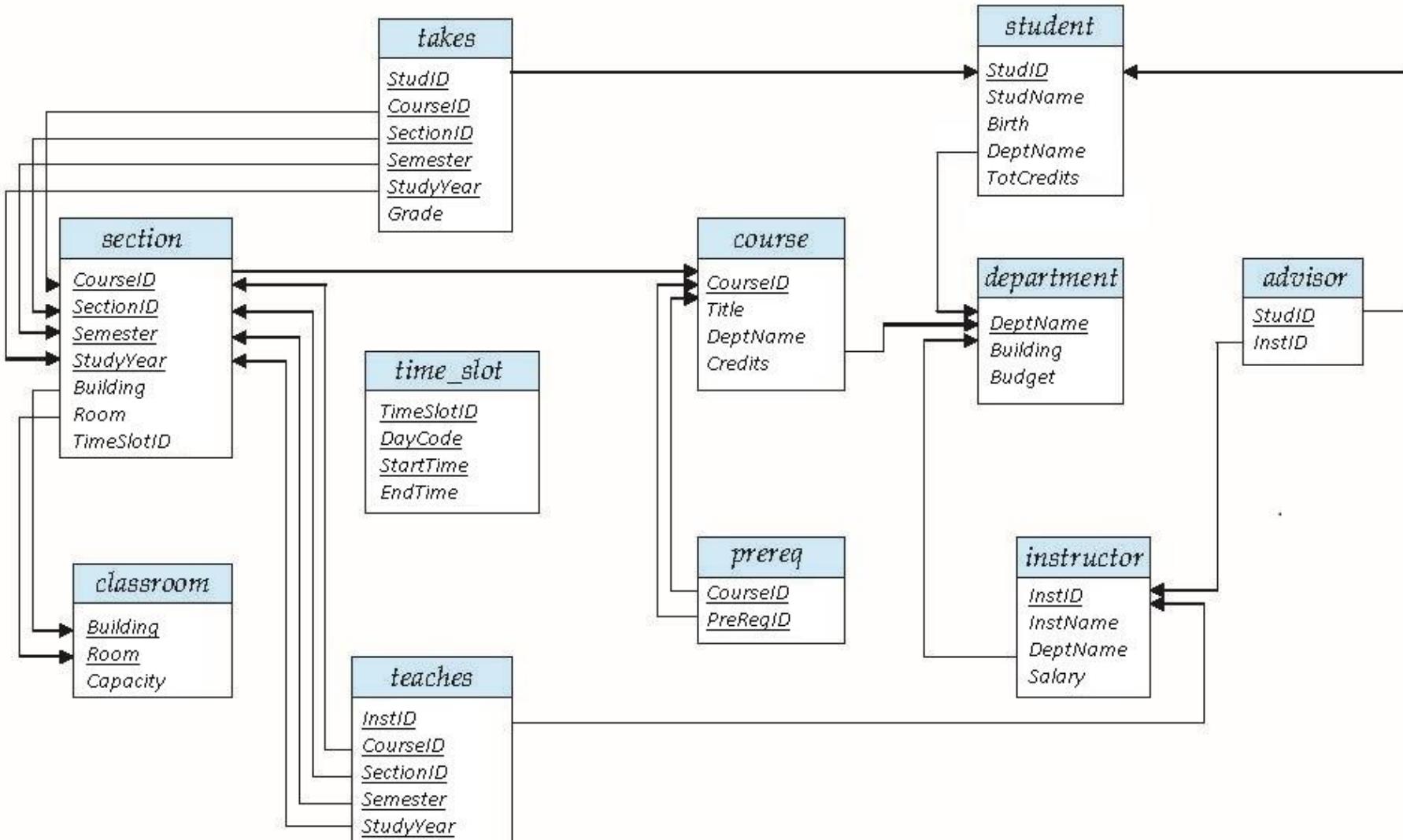
PreReq(CourseID, PreReqID)

■ Challenge

- How to give tables and attributes short but meaningful names
- E.g. How to name the ID attribute of an instructor: InstructorID, IID, Inst_ID or InstID?
- Here some of the names differ slightly from those in the textbook.

3 - Database Schema Diagram

from the book, but with new names



4 - SQL Data Definition

- SQL CREATE TABLE statements for each of the tables can be found in the **UniversityDB.sql** script on DTU Learn under *Content->Databases* .

5 - Database Instance (1 of 4)

Instructor

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

Student

StudID	StudName	Birth	DeptName	TotCredits
00128	Zhang	1992-04-18	Comp. Sci.	102
12345	Shankar	1995-12-06	Comp. Sci.	32
19991	Brandt	1993-05-24	History	80
23121	Chavez	1992-04-18	Finance	110
44553	Peltier	1995-10-18	Physics	56
45678	Levy	1995-08-01	Physics	46
54321	Williams	1995-02-28	Comp. Sci.	54
55739	Sanchez	1995-06-04	Music	38
70557	Snow	1995-11-22	Physics	0
76543	Brown	1994-03-05	Comp. Sci.	58
76653	Aoi	1993-09-18	Elec. Eng.	60
98765	Bourikas	1992-09-23	Elec. Eng.	98
98988	Tanaka	1992-06-02	Biology	120

Advisor

StudID	InstID
12345	10101
44553	22222
45678	22222
00128	45565
76543	45565
23121	76543
98988	76766
76653	98345
98765	98345

Department

DeptName	Building	Budget
Biology	Watson	90000.00
Comp. Sci.	Taylor	100000.00
Elec. Eng.	Taylor	85000.00
Finance	Painter	120000.00
History	Painter	50000.00
Music	Packard	80000.00
Physics	Watson	70000.00

Database Instance (2 of 4)

Teaches

InstID	CourseID	SectionID	Semester	StudyYear
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
10101	CS-101	1	Fall	2009
45565	CS-101	1	Spring	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
10101	CS-315	1	Spring	2010
45565	CS-319	1	Spring	2010
83821	CS-319	2	Spring	2010
10101	CS-347	1	Fall	2009
98345	EE-181	1	Spring	2009
12121	FIN-201	1	Spring	2010
32343	HIS-351	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

Takes

StudID	CourseID	SectionID	Semester	StudyYear	Grade
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	NULL

Database Instance (3 of 4)

Course

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

PreReq

CourseID	PreReqID
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Database Instance (4 of 4)

Section

CourseID	SectionID	Semester	StudyYear	Building	Room	TimeSlotID
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Classroom

Building	Room	Capacity
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

TimeSlot

TimeSlotID	DayCode	StartTime	EndTime
A	M	08:00:00	08:50:00
A	W	08:00:00	08:50:00
A	F	08:00:00	08:50:00
B	M	09:00:00	09:50:00
B	W	09:00:00	09:50:00
B	F	09:00:00	09:50:00
C	M	11:00:00	11:50:00
C	W	11:00:00	11:50:00
C	F	11:00:00	11:50:00
D	M	13:00:00	13:50:00
D	W	13:00:00	13:50:00
D	F	13:00:00	13:50:00
E	T	10:30:00	11:45:00
E	R	10:30:00	11:45:00
F	T	14:30:00	15:45:00
F	R	14:30:00	15:45:00
G	M	16:00:00	16:50:00
G	W	16:00:00	16:50:00
G	F	16:00:00	16:50:00
H	W	10:00:00	12:30:00

Note: In writing and testing SQL queries a print of the Diagram and the Database Instance is most useful.

SQL Command Categories (book sec 3.1)

■ SQL Command Categories with Examples

- **Data Definition Language (DDL)**

CREATE DATABASE, DROP DATABASE,
CREATE TABLE, ALTER TABLE, DROP TABLE,
CREATE INDEX, ALTER INDEX, DROP INDEX,
CREATE VIEW and DROP VIEW.

- **Data Manipulation Language (DML)**

INSERT, UPDATE and DELETE

- **Data Query Language**

SELECT

- **Data Control Language**

CREATE USER, ALTER PASSWORD, GRANT, REVOKE and CREATE SYNONYM

- **Data Administration Commands**

START AUDIT and STOP AUDIT

- **Transactional Control Commands**

SET TRANSACTION, COMMIT, ROLLBACK and SAVEPOINT



SQL Data Definition Language (DDL) (book sec 3.2)

- The DDL provides commands for changing database schemas.
- In this section you will learn about
 - The commands CREATE DATABASE, USE and DROP DATABASE
 - The commands CREATE TABLE, DROP TABLE and ALTER TABLE
 - Types/Domains in SQL
 - Constraints in SQL

CREATE DATABASE, USE and DROP DATABASE

■ Create a Database

- Create a database with the name 'University':

```
CREATE DATABASE University;
```

■ Use a Database

- Instruct the DBMS to use the University as the default (current) database for subsequent statements. The database remains the default until the end of the session or another USE statement is issued.

```
USE University;
```

■ Drop an existing Database

```
DROP DATABASE University;
```

Domain Types in SQL

- **String types**

CHAR(n)	Fixed length character strings, with user-specified length n
VARCHAR(n)	Variable length character strings, with user-specified maximum length n.

- **Date and Time**

DATE	A Date in the format 'YYYY-MM-DD'
...	

- **Numeric types**

INT	Integers, a finite subset of the integers that is machine-dependent.
SMALLINT	Small integers (a machine-dependent subset of INT).
DECIMAL(p,d)	Fixed point numbers, with user-specified precision of p digits, with d digits to the right of decimal point.
FLOAT(n)	Floating point numbers (like 1.23 and 1.23E5) with user-specified precision of at least n digits.

Many more are available – the selection depends on the DBMS implementation.

Choosing Domains wisely will enhance consistency, save storage and improve response time!

CREATE TABLE

- An SQL relation is defined using the **CREATE TABLE** command

- Typical form: **CREATE TABLE R ($A_1 D_1, A_2 D_2, \dots, A_n D_n,$
(integrity-constraint₁),
...,
(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 (domain) of values of attribute A_i .
- Possible integrity constraints include: primary key, foreign key, not null.

- Example: Creation of a Table with a Primary Key

```
CREATE TABLE Classroom
  (Building  VARCHAR(15),
   Room      VARCHAR(7),
   Capacity   DECIMAL(4,0),
   PRIMARY KEY(Building, Room));
```

- A Primary Key represents an Integrity Constraint

In the example, it will ensure that all rows have

1. a unique (Building, Room) combination
2. Building and Room are not NULL.

CREATE TABLE

■ Example: creation of a table with a Foreign Key Constraint

CREATE TABLE Instructor

```
(InstID      VARCHAR(5),  
InstName    VARCHAR(20),  
DeptName    VARCHAR(20),  
Salary      DECIMAL(8,2),  
PRIMARY KEY (InstID),  
FOREIGN KEY(DeptName) REFERENCES Department(DeptName);
```

The foreign key will ensure that the DeptName in any **Instructor** row will also appear in **Department**.

In particular this means that the DBMS should as a default (for other possibilities see next page) disallow

- insert violations: the insertion of a row into **Instructor**, if its DeptName is not in the **Department** table
- delete violation: the deletion of a row from **Department** table, if its DeptName is used in a row in **Instructor**
- update violations: update of a DeptName in **Department**, if its DeptName is used in a row in **Instructor**

Instructor:

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

Department:

DeptName	Building	Budget
Biology	Watson	90000.00
Comp. Sci.	Taylor	100000.00
Elec. Eng.	Taylor	85000.00
Finance	Painter	120000.00
History	Painter	50000.00
Music	Packard	80000.00
Physics	Watson	70000.00

CREATE TABLE

- It is possible to define referential actions for foreign key constraints, e.g.:
 - ON DELETE SET NULL
 - ON DELETE CASCADE

- Example:**

CREATE TABLE Instructor

```
(InstID      VARCHAR(5),  
 InstName    VARCHAR(20),  
 DeptName    VARCHAR(20),  
 Salary      DECIMAL(8,2),  
 PRIMARY KEY (InstID),  
 FOREIGN KEY(DeptName) REFERENCES  
 Department(DeptName) ON DELETE SET NULL);
```

Now it is allowed to delete a **Department** row having a **DeptName** used in some rows in the **Instructor** table. In that case the **DeptName** in those **Instructor** rows are set to **NULL**.

Instructor

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	NULL	95000.00
32343	El Said	History	60000.00
33456	Gold	NULL	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

Department

DeptName	Building	Budget
Biology	Watson	90000.00
Comp. Sci.	Taylor	100000.00
Elec. Eng.	Taylor	85000.00
Finance	Painter	120000.00
History	Painter	50000.00
Music	Packard	80000.00
Physics	Watson	70000.00

CREATE TABLE

- It is possible to define referential actions for foreign key constraints, e.g.:
 - ON DELETE SET NULL
 - ON DELETE CASCADE

- Example:**

CREATE TABLE Instructor

```
(InstID      VARCHAR(5),  
 InstName    VARCHAR(20),  
 DeptName    VARCHAR(20),  
 Salary      DECIMAL(8,2),  
 PRIMARY KEY (InstID),  
 FOREIGN KEY(DeptName) REFERENCES  
 Department(DeptName) ON DELETE CASCADE);
```

Now it is allowed to delete a **Department** row having a DeptName used in some rows in the **Instructor** table. In that case those **Instructor** rows are deleted too.

Instructor

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

Department

DeptName	Building	Budget
Biology	Watson	90000.00
Comp. Sci.	Taylor	100000.00
Elec. Eng.	Taylor	85000.00
Finance	Painter	120000.00
History	Painter	50000.00
Music	Packard	80000.00
Physics	Watson	70000.00

CREATE TABLE

- **Creation of a table using NOT NULL**

CREATE TABLE Instructor

```
(InstID      VARCHAR(5),  
InstName    VARCHAR(20) NOT NULL,  
DeptName    VARCHAR(20),  
Salary      DECIMAL(8,2),  
PRIMARY KEY (InstID),  
FOREIGN KEY(DeptName) REFERENCES Department(DeptName) ON DELETE SET NULL);
```

- **NOT NULL is an Integrity constraint**

- NOT NULL will ensure that each row always has an instructor name

DROP TABLE and ALTER TABLE

- To delete all rows in a table and the table schema

DROP TABLE Student;

- To delete all rows in a table, but retain the table schema

DELETE FROM Student;

- To add an attribute with assigned NULL values

ALTER TABLE Student **ADD** Shoesize DECIMAL(2,0);

- To drop an attribute

ALTER TABLE Student **DROP** Shoesize;

- To add a primary key or a foreign key

See page 42.

SQL Data Manipulation (book sec 3.9)

■ Modification of the Database

- Insert new rows into a table
- Delete rows from a table
- Update rows in a table

■ Examples

- Add new instructor, that is adding a row
- Delete an instructor, that is deleting a row
- Update salary for an instructor, that is updating a value in a row

INSERT

- Simplest form

```
INSERT INTO R (... ,Ai, ...) VALUES (... ,vali, ...);
```

- R represents a relation
- A_i is an attribute of R, val_i is a value expression

- Add a new row to Course

```
INSERT Course (CourseID, Title, DeptName, Credits)  
VALUES ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

- Or equivalently

```
INSERT Course VALUES ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

- Add a new row in Student with TotCredits set to NULL

```
INSERT Student VALUES ('3003', 'Green', '1993-04-16', 'Finance', NULL);
```

- Adding multiple rows

```
INSERT Course VALUES  
('CS-437', 'Database Systems', 'Comp. Sci.', 4),  
('CS-528', 'Big Data Systems', 'Comp. Sci.', 5),  
('CS-530', 'Data Warehouse', 'Comp. Sci.', 4);
```

- Can also take the form **INSERT INTO R (... ,A_i, ...) SELECT ... FROM ... WHERE ...;**

(See page 42 and demo example 2.8.)

DELETE

■ Typical form

DELETE FROM R WHERE P;

- R represents a relation
- P is a (row) predicate over attribute names
- Removes those rows for which P is true

■ Delete a row

- **DELETE FROM Instructor WHERE InstID = 45565;**

■ Delete multiple rows

- **DELETE FROM Instructor WHERE DeptName='Finance';**

■ Delete all rows

- **DELETE FROM Instructor;**

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

UPDATE

■ Typical form

```
UPDATE R SET ... , Ai = vali , ... WHERE P;
```

- **R** represents a relation
- **P** is a (row) predicate over attribute names
- **A_i** is an attribute of **R**, **val_i** is a value expression
- changes the value of **A_i** to **val_i** for those rows for which **P** is true

■ Update multiple values in a single row

```
UPDATE Instructor SET Salary=85000, DeptName='Physics' WHERE InstID=45565;
```

■ Update multiple rows

- Increase salaries of instructors whose salary is over 80000 by 3%, and all others with a 5% raise.
- Write two update statements (the order is important):

```
UPDATE Instructor SET Salary=Salary*1.03 WHERE Salary>80000;
```

```
UPDATE Instructor SET Salary=Salary*1.05 WHERE Salary<=80000;
```

CASE Expressions

- Increase salaries of instructors whose salary is over 80,000 by 3%, and all others with a 5% raise.

```
UPDATE Instructor SET Salary =  
CASE  
WHEN Salary<=80000 THEN Salary*1.05  
ELSE Salary*1.03  
END;
```

- Consider giving more to those, who have plenty!

```
UPDATE Instructor SET Salary =  
CASE  
WHEN Salary BETWEEN 80000 AND 89999 THEN Salary+10000  
WHEN Salary BETWEEN 70000 AND 79999 THEN Salary+5000  
WHEN Salary BETWEEN 0 AND 69999 THEN Salary+2500  
ELSE Salary  
END;
```

SQL Data Queries (book sec 3.3.1-3.3.3, 3.4.1, 3.4.3, 3.4.4)

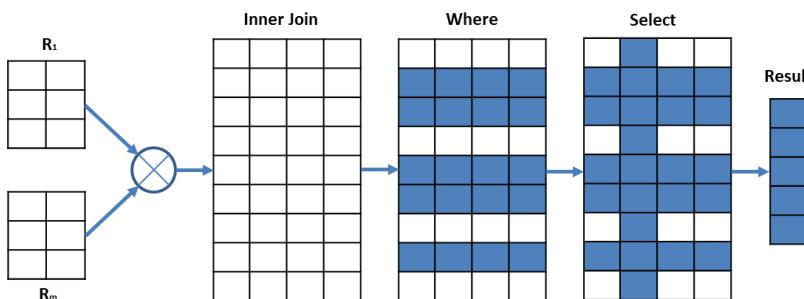
■ Basic Query Structure

- SQL Data Query Language provides the ability to query data
- The result of a SQL Query is a table

■ A typical basic SQL query has the form:

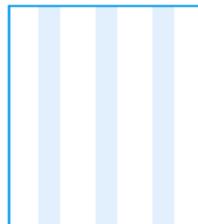
```
SELECT A1, A2, ..., An  
FROM r1, r2, ..., rm  
WHERE P;
```

- A_i represents an attribute
- r_i represents a relation
- P is a (row) predicate over attribute names. For each row it is either true or false.



SELECT

- The select clause lists the attributes desired in the result of a query
 - Corresponds to the projection operation of relational algebra



- Example: Like find the names of all instructors

```
SELECT InstName FROM Instructor;
```

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

- Note
 - SQL names are case insensitive. You may use upper- or lower-case letters, for example InstName \equiv INSTNAME \equiv instname.

SELECT: DISTINCT, ALL

- SQL generally allows duplicate rows in tables as well as in query results.
- To force the elimination of duplicates, insert the keyword DISTINCT after SELECT.
 - Like find the names of all departments from the Instructor table and remove duplicates:
`SELECT DISTINCT DeptName FROM Instructor;`

- Keyword ALL specifies that duplicates are not removed
`SELECT ALL DeptName FROM Instructor;`

SELECT: using *, expressions, AS

- A '*' in the SELECT clause denotes “all attributes”

SELECT * FROM Instructor;

- The select clause can contain **arithmetic expressions** involving the operation, +, −, *, and /, and operating on constants or attributes of rows.
- The query:

SELECT InstID, Salary/12 AS Monthly FROM Instructor;

InstID	Monthly
10101	5416.666667
12121	7500.000000
15151	3333.333333

- Returns a table where the value of the attribute Salary is divided by 12, giving the monthly salary instead of the yearly salary.

SELECT: using built-in functions

- The select clause can contain *built-in functions* associated with the built-in datatypes.

- Example (for the Date datatype):

- `SELECT CURDATE();`

```
curdate0
2015-06-22
```

- Example: Calculation of Age from Birth

- Basic rule:

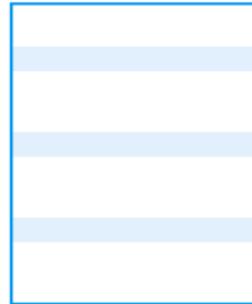
Never store dynamic values like Age, instead store static values like Birth, and calculate dynamic values like Age.

```
SELECT StudName, Birth,  
TIMESTAMPDIFF(YEAR, Birth, CURDATE()) AS Age  
FROM Student;
```

StudName	Birth	Age
Zhang	1992-04-18	23
Shankar	1995-12-06	19
Brandt	1993-05-24	22
Chavez	1992-04-18	23
Peltier	1995-10-18	19
Levy	1995-08-01	19
Williams	1995-02-28	20
Sanchez	1995-06-04	20
Snow	1995-11-22	19
Brown	1994-03-05	21
Aoi	1993-09-18	21
Bourikas	1992-09-23	22
Tanaka	1992-06-02	23

WHERE

- WHERE clause specifies conditions for the rows to be included in the result



- **Example:**

Find all instructors in Comp. Sci. department with Salary > 70000

```
SELECT InstName FROM Instructor  
WHERE DeptName='Comp. Sci.' AND Salary>70000;
```

- Comparisons with <, <=, >, >=, =, <> can be applied to arithmetic expressions and strings.
- Comparison results can be combined using the logical connectives **AND, OR, and NOT**.

FROM (section 3.3.2)

- The **FROM** clause lists the tables involved in the query
 - **FROM r**
 - **FROM r₁, ..., r_m**
corresponds to **Cartesian Product** operation $r_1 \times \dots \times r_m$ of Relational Algebra, also called a **join**.
- Example: Find the Cartesian Product: Instructor x Teaches

`Instructor(InstID, InstName, DeptName, Salary) Teaches(InstID, CourseID, SectionID, Semester, StudyYear)`

`SELECT * FROM Instructor, Teaches;`

- Generates every possible Instructor-Teaches pair
 - The result table has as attributes the union of Instructor&Teaches attrs
- | InstID | InstName | DeptName | Salary | InstID | CourseID | SectionID | Semester | StudyYear |
|--------|----------|----------|--------|--------|----------|-----------|----------|-----------|
|--------|----------|----------|--------|--------|----------|-----------|----------|-----------|
- The result table has $n * m$ rows, where n is the number of rows in Instructor and m is the number of rows in Teaches.
 - Beware of data explosion! A Cartesian Product of A x B x C, where each table has 100 rows, results in 1.000.000 rows (either the memory area will overflow with an error, or the response time will go towards infinity)!
 - Cartesian Product is not very useful directly, but useful combined with the WHERE clause condition.

Cartesian Product: Instructor x Teaches

- Combines each row in Instructor with all rows in Teaches
- Instructor(InstID, InstName, DeptName, Salary) Teaches(InstID, CourseID, SectionID, Semester, StudyYear)

SELECT * FROM Instructor, Teaches LIMIT 14;

InstID	InstName	DeptName	Salary	InstID	CourseID	SectionID	Semester	StudyYear
10101	Srinivasan	Comp. Sci.	65000.00	76766	BIO-101	1	Summer	2009
12121	Wu	Finance	90000.00	76766	BIO-101	1	Summer	2009
15151	Mozart	Music	40000.00	76766	BIO-101	1	Summer	2009
22222	Einstein	Physics	95000.00	76766	BIO-101	1	Summer	2009
32343	El Said	History	60000.00	76766	BIO-101	1	Summer	2009
33456	Gold	Physics	87000.00	76766	BIO-101	1	Summer	2009
45565	Katz	Comp. Sci.	75000.00	76766	BIO-101	1	Summer	2009
58583	Califieri	History	62000.00	76766	BIO-101	1	Summer	2009
76543	Singh	Finance	80000.00	76766	BIO-101	1	Summer	2009
76766	Crick	Biology	72000.00	76766	BIO-101	1	Summer	2009
83821	Brandt	Comp. Sci.	92000.00	76766	BIO-101	1	Summer	2009
98345	Kim	Elec. Eng.	80000.00	76766	BIO-101	1	Summer	2009
10101	Srinivasan	Comp. Sci.	65000.00	76766	BIO-301	1	Summer	2010
12121	Wu	Finance	90000.00	76766	BIO-301	1	Summer	2010

SELECT COUNT(*) FROM Instructor, Teaches;

COUNT(*)
180

More JOIN Examples

- Given
 - Instructor(InstID, InstName, DeptName, Salary)
 - Teaches(InstID, CourseID, SectionID, Semester, StudyYear)
- For all instructors who have taught some course, find their names and the course ID of the courses they taught.

```
SELECT InstName, CourseID
  FROM Instructor, Teaches
 WHERE Instructor.InstID=Teaches.InstID;
```

InstName	CourseID
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

More JOIN Examples

- Given
 - Section(CourseID, SectionID, Semester, StudyYear, Building, Room, TimeSlotID)
 - Course(CourseID, Title, DeptName, Credits)
- Find the CourseID, Semester, StudyYear and Title of each course offered by the Comp. Sci. department
 - SELECT** Section.CourseID, Semester, StudyYear, Title
 - FROM** Section, Course
 - WHERE** Section.CourseID=Course.CourseID **AND** DeptName='Comp. Sci.' ;

CourseID	Semester	StudyYear	Title
CS-101	Fall	2009	Intro. to Computer Science
CS-101	Spring	2010	Intro. to Computer Science
CS-190	Spring	2009	Game Design
CS-190	Spring	2009	Game Design
CS-315	Spring	2010	Robotics
CS-319	Spring	2010	Image Processing
CS-319	Spring	2010	Image Processing
CS-347	Fall	2009	Database System Concepts

NATURAL JOIN (section 3.3.3)

NATURAL JOIN

- Matches rows with the same values for all common attributes, and retains only one copy of each common attribute value.
- **SELECT * FROM Instructor NATURAL JOIN Teaches;**

InstID	InstName	DeptName	Salary	CourseID	SectionID	Semester	StudyYear
10101	Srinivasan	Comp. Sci.	65000.00	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000.00	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000.00	CS-347	1	Fall	2009
12121	Wu	Finance	90000.00	FIN-201	1	Spring	2010
15151	Mozart	Music	40000.00	MU-199	1	Spring	2010
22222	Einstein	Physics	95000.00	PHY-101	1	Fall	2009
32343	El Said	History	60000.00	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000.00	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000.00	CS-319	1	Spring	2010
76766	Crick	Biology	72000.00	BIO-101	1	Summer	2009
76766	Crick	Biology	72000.00	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000.00	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000.00	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000.00	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000.00	EE-181	1	Spring	2009

Natural Join in this example is over the common attribute name InstID.

NATURAL JOIN versus JOIN

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

```
SELECT InstName, CourseID  
FROM Instructor, Teaches  
WHERE Instructor.InstID=Teaches.InstID;
```

- The above SELECT expression is equivalent with:

```
SELECT InstName, CourseID  
FROM Instructor NATURAL JOIN Teaches;
```

Renaming: AS in **SELECT** and **FROM** (section 3.4.1)

- The SQL allows renaming tables and attributes using the **AS** clause: **OldName AS NewName**
 - Example:

```
SELECT InstID, InstName, Salary/12 AS MonthlySalary
FROM Instructor;
```
 - Example: Find the names of all instructors who have a higher salary than some instructor in the 'Comp. Sci.' department.

```
SELECT DISTINCT T.InstName
FROM Instructor AS T, Instructor AS S
WHERE T.Salary > S.Salary AND S.DeptName = 'Comp. Sci.');
```
- Keyword **AS** is optional and may be omitted
Instructor AS T ≡ Instructor T

SELECT revisited: use of * (section 3.4.3)

- For all instructors who have taught some course, select all attributes.

```
SELECT Instructor.* FROM Instructor, Teaches  
WHERE Instructor.InstID=Teaches.InstID;
```

ORDER BY (section 3.4.4)

- List in alphabetic order the names of all instructors

SELECT DISTINCT InstName FROM Instructor ORDER BY InstName;

- We may specify **DESC** for descending order or **ASC** for ascending order, for each attribute; Ascending order is the default.
- ... **ORDER BY InstName DESC;**

- Can sort on multiple attributes

**SELECT DeptName, InstName
FROM Instructor
ORDER BY DeptName, InstName;**

DeptName	InstName
Biology	Crick
Comp. Sci.	Brandt
Comp. Sci.	Katz
Comp. Sci.	Srinivasan
Elec. Eng.	Kim
Finance	Singh
Finance	Wu
History	Califieri
History	El Said
Music	Mozart
Physics	Einstein
Physics	Gold

Making New Tables from Old Tables

- Create a new table, like the old table, and insert values

```
CREATE TABLE Specialist1 LIKE Instructor; # Create an empty table
INSERT Specialist1 SELECT * FROM Instructor WHERE DeptName='Comp. Sci.';
```

- Create a table with contents from an old table

```
CREATE TABLE Specialist2 SELECT * FROM Instructor WHERE DeptName='Physics';
```

```
SELECT * FROM Specialist1;
```

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
45565	Katz	Comp. Sci.	75000.00
83821	Brandt	Comp. Sci.	92000.00

```
SELECT * FROM Specialist2;
```

InstID	InstName	DeptName	Salary
22222	Einstein	Physics	95000.00
33456	Gold	Physics	87000.00

- However, Specialist1 has a Primary Key(InstID), while Specialist2 has none!

```
ALTER TABLE Specialist2 ADD PRIMARY KEY(InstID); # To define Primary Key
```

- Neither Specialist1 nor Specialist2 has Foreign Keys defined!

```
ALTER TABLE Specialist1 ADD FOREIGN KEY(DeptName)
```

```
REFERENCES Department(DeptName); # To define a Foreign Key
```



Summary

- University Database Example
- Data Definition: CREATE TABLE, DROP TABLE and ALTER TABLE.
- Data Manipulation: INSERT, UPDATE and DELETE data
- Data Queries: SELECT ... FROM ... WHERE ...

Readings

- In Database Systems Concepts please read the Chapter 3.1, 3.2, 3.9 (except parts using subqueries), 3.3-3.4 (except 3.4.2).
- Pay special attention to the Summary

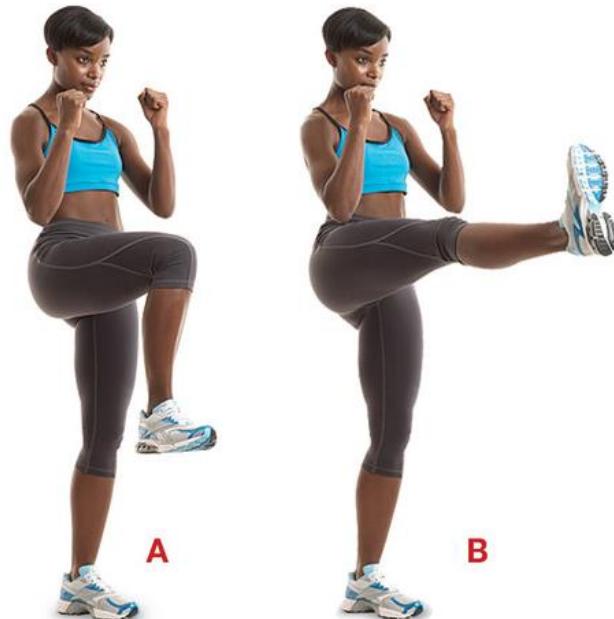
Next time

- The remaining parts of chapter 3: more advanced queries

Tools Installations

- *If you have not yet made exercises 1-3 of slide set no 1.5 from the first week then do them before starting on the exercises of this slide set.*

Demo Exercises



Demo Exercises clarify ideas and concepts from the Lecture to provide you with good Database Skills.

Discuss and redo the Demo Exercises.

Create and Download the University Database

Create the University Database

Start MySQL Workbench. Once it is running select connect to database and login with the Username and Password you created during installation.

Define a name for a university database (e.g. University) and start to use it by executing the following SQL statements:

```
CREATE DATABASE University;  
USE University;
```

Download the University Database

To create the university tables, download the “UniversityDB.sql” file to your PC from the *Databases* folder under DTU Learn Content.

Use 'Open an SQL script file in a new query tab' in MySQL Workbench, and select the file 'UniversityDB.sql' and execute it.

You can check that all the relation instances match the textbook 'Database Systems Concepts, appendix A, Detailed University Schema' by using queries like:

```
DESCRIBE Student;  
SELECT * FROM Student;
```

Basic SQL Queries

Demo Examples

The following examples will provide you with some basic SQL query knowledge, that you will need to solve the exercises. It is optional, but highly recommended, to copy the Demo Example queries into the workbench, and run them on the university database. Try to change some of the conditions in the queries, this will help you to get familiarized with basic SQL queries.

Each Demo Example contains a plain text, describing the query, the actual SQL query and the resulting dataset.

Please be aware that the examples given are not the only way to write the SQL queries, but just one way to do it. Often you will be able to write queries in more than one way.

2.1 SELECT all attributes

Get all the departments on the university.

```
SELECT * FROM Department;
```

DeptName	Building	Budget
Biology	Watson	90000.00
Comp. Sci.	Taylor	100000.00
Elec. Eng.	Taylor	85000.00
Finance	Painter	120000.00
History	Painter	50000.00
Music	Packard	80000.00
Physics	Watson	70000.00

2.2 SELECT some attributes

Get the name and building of all departments on the university.

```
SELECT DeptName, Building FROM Department;
```

DeptName	Building
Biology	Watson
Comp. Sci.	Taylor
Elec. Eng.	Taylor
Finance	Painter
History	Painter

Basic SQL Queries

2.3 SELECT rows based on single condition

Get all departments with a budget greater than or equal to 100000.

```
SELECT * FROM Department  
WHERE Budget >= 100000;
```

DeptName	Building	Budget
Comp. Sci.	Taylor	100000.00
Finance	Painter	120000.00

2.4 SELECT rows based on multiple conditions

Get all departments with a budget greater than or equal to 70000 and that is located in the Taylor building.

```
SELECT * FROM Department  
WHERE Budget >= 70000  
AND Building = "Taylor";
```

DeptName	Building	Budget
Comp. Sci.	Taylor	100000.00
Elec. Eng.	Taylor	85000.00

2.5 JOIN with condition and NATURAL JOIN

Get the instructors (id and name) that work in a department with a budget ≥ 100000 .

```
SELECT InstID, InstName  
FROM Instructor, Department  
WHERE  
Instructor.DeptName = Department.DeptName  
AND Budget >= 100000;
```

```
SELECT InstID, InstName  
FROM Instructor NATURAL JOIN Department  
WHERE Budget >= 100000;
```

InstID	InstName
10101	Srinivasan
45565	Katz
83821	Brandt
12121	Wu
76543	Singh

(The order of rows might differ for different systems.)

SQL Data Manipulation

2.6 UPDATE with calculation

Increase the salary of each instructor in the Comp. Sci. department by 10%.

```
UPDATE Instructor SET Salary = Salary * 1.10  
WHERE DeptName = 'Comp. Sci.';
```

2.7 DELETE

Delete all courses with credits less than or equal to 3.

```
DELETE FROM Course  
WHERE Credits <= 3;
```

Important note: In order to be able to execute the statement above, you will need to disable *safe mode* (if this is not already done) by executing:

```
SET SQL_SAFE_UPDATES = 0;
```

The reason is that if `SQL_SAFE_UPDATES = 1`, then the `WHERE` clause (of delete and update statements) must refer to the key attribute, but that is not the case in the delete statement above.

If you later wish to enable the safe mode again, this can be done by executing:

```
SET SQL_SAFE_UPDATES = 1;
```

2.8 INSERT with SELECT

Insert every student whose `TotCreds` attribute is greater than 100 as an instructor in the same department, with a salary of 10000.

```
INSERT INTO Instructor  
SELECT StudID, StudName, DeptName, 10000  
FROM Student  
WHERE TotCredits > 100;
```

PS. Run the `UniversityDB.sql` script again to restore tables to the original instance.

2.9 INSERT with VALUES: see 2.11

SQL Data Definition and Data Manipulation

2.10 Create table GradePoints(Grade, Points)

to provide a conversion from letter grades in the Takes table to numeric scores; For example an “A” grade could be specified to corresponds to 4 points, an “A-“ to 3.7 points, a “B+” to 3.3 points and so on.

```
CREATE TABLE GradePoints (
  Grade VARCHAR(2) PRIMARY KEY,
  Points DECIMAL(3,1));
```

2.11 Populate the table with data

```
INSERT GradePoints VALUES ('A', 4.0);
INSERT GradePoints VALUES ('A-', 3.7);
INSERT GradePoints VALUES ('B+', 3.3);
INSERT GradePoints VALUES ('B', 3.0);
INSERT GradePoints VALUES ('B-', 2.7);
INSERT GradePoints VALUES ('C+', 2.3);
INSERT GradePoints VALUES ('C', 2.0);
INSERT GradePoints VALUES ('F', 0.0);
```

2.12 Drop table

```
DROP TABLE GradePoints;
```

Exercises



Please answer all exercises
to demonstrate your
Database Skills.

Solutions are available at 11:45

Basic SQL Queries

Exercises

SQL queries should be run against the University database.

Have a print of the University Database Schema Diagram and the Database Instance readily available.

2.13 SELECT all attributes

Get all data in the Student table

2.14 SELECT all attributes

Get all data in the Course table

2.15 SELECT only one attribute

Get the name of all students

2.16 SELECT multiple attributes

Get the name and total credits of all students

2.17 SELECT multiple attributes

Get the name, salary and department of all instructors.

2.18 SELECT only some rows

Get the names of all students with a total credit of more than 100.

2.19 SELECT rows based on multiple conditions

Get the students in Computer Science with a total credit of more than 100.

2.20 SELECT rows based on multiple conditions

Get the rooms with a capacity between 25 and 50, or located in the Painter building.

2.21 SELECT rows based on single condition

Get all department names not located in the Taylor building.

2.22 SELECT with two tables

What are the Course ID, year and grade for all courses taken by student Shankar?

SQL Data Manipulation

2.23 INSERT with multiple rows

Create two new Comp. Sci. courses CS-102 and CS-103 in table Course titled Weekly Seminar and Monthly Seminar, both with 0 credits.

2.24 INSERT with multiple NULL values

Create a section for both CS-102 and CS-103 in Fall 2009, both with SectionID 1.

2.25 INSERT with SELECT and NULL

In table Takes enroll every student in the Comp. Sci. department in the section for CS-102.

2.26 DELETE

Delete both courses CS-102 and CS-103 in the Takes table.

2.27 UPDATE

Move the Finance department to the Taylor building.

PS. Run the UniversityDB Script to restore tables to initial instances.

Create a Database & Populate it with Data

2.28 Create a Database

Write SQL DDL statements corresponding to the Relation Schemas below for an Insurance Database.

Person (DriverID, DriverName, Address)

Car (License, Model, ProdYear)

Accident (ReportNumber, AccDate, Location)

Owns (DriverID, License)

Participants (ReportNumber, License, DriverID, DamageAmount)

Make any reasonable assumptions about data types, and declare primary and foreign keys.

2.29 Populate a Database

Write SQL DML statements to populate the database with data, to end up with:

SELECT * FROM Person;

DriverID	DriverName	Address
31262549	Hans Hansen	Jernbane Alle 74, 2720 Vanløse

SELECT * FROM Car;

License	Model	ProdYear
JW46898	Honda Accord Aut. 2.0	2001

SELECT * FROM Accident;

ReportNumber	AccDate	Location
3004000121	2015-06-18	2605 Brøndby

SELECT * FROM Owns;

DriverID	License
31262549	JW46898

SELECT * FROM Participants;

ReportNumber	License	DriverID	DamageAmount
3004000121	JW46898	31262549	6800



Technical University of Denmark

DTU Compute

Department of Applied Mathematics and Computer Science

Introductory SQL, Part 2

Create Databases and Select Data

Chapter 3 in the Text Book

©Anne Haxthausen and Flemming Schmidt

These slides have been prepared by Anne Haxthausen, partly reusing/modifying slides by Flemming Schmidt, which again partly reused some slides by Silberschatz, Korth. Sudarshan, 2010.

Contents

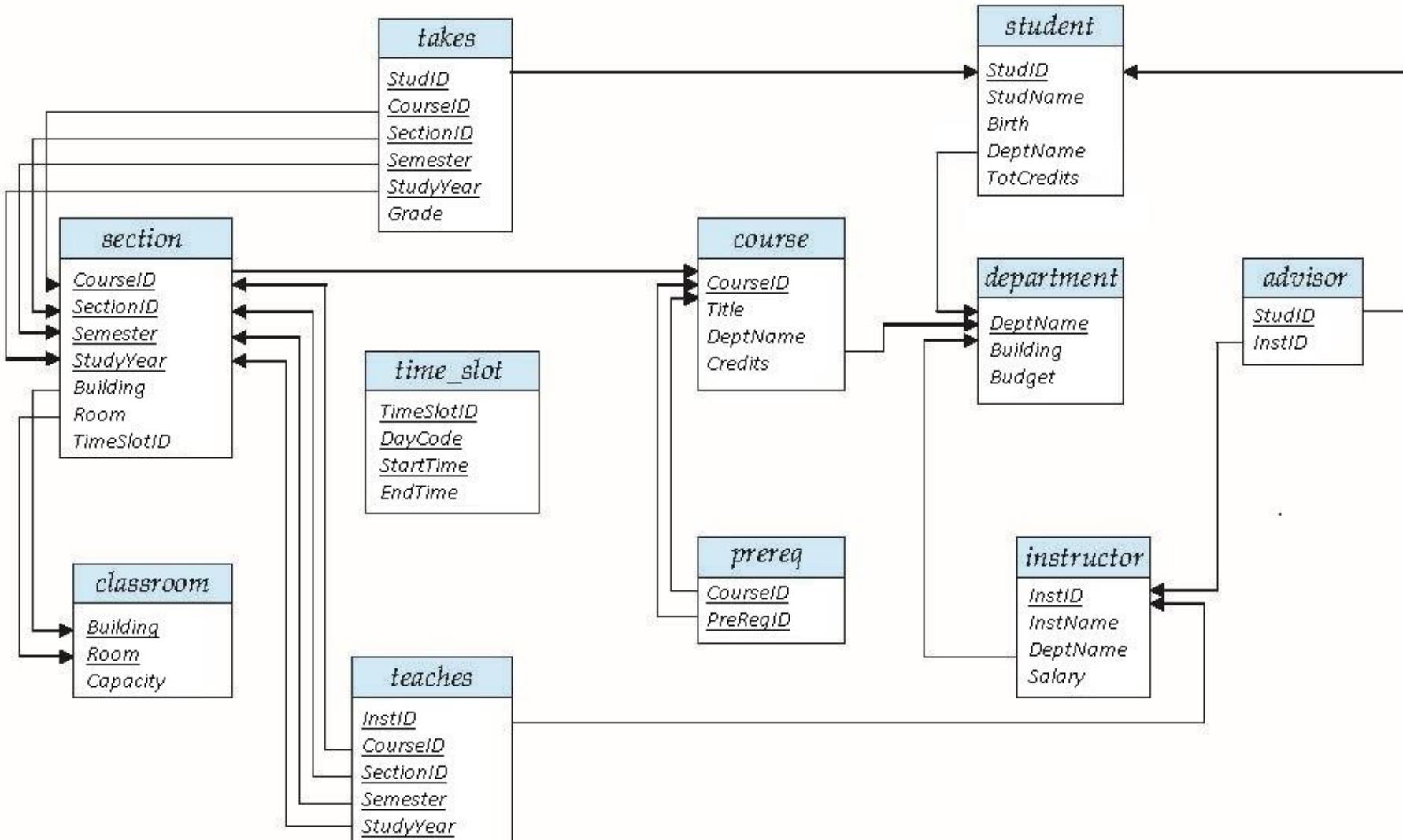
- University Database (running example)
- SQL Data, more details:
 - String Operations (book sec. 3.4.2)
 - NULL Values (book sec. 3.6)
- Advanced SQL Queries Using SELECT
 - Aggregate Functions + GROUP BY ... HAVING ... (book sec. 3.7)
 - Subqueries (book sec. 3.8, also used in 3.9)
 - Scalar subqueries (book sec. 3.8.7)
 - Conditions using IN, NOT IN (book sec. 3.8.1)
 - Conditions using op ALL, op SOME (book sec. 3.8.2)
 - Conditions using EXISTS, NOT EXISTS (book sec. 3.8.3)
- Composite SQL Queries Using UNION, INTERSECT and EXCEPT (book sec. 3.5)
- Demo Exercises & Exercises



University Database, revisited

- Database example used throughout the textbook and in this course!
 1. Database Schema Diagram
 2. Database Instance

Database Schema Diagram

 from the book, but with new names

Database Instance (1 of 4)

Instructor

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

Student

StudID	StudName	Birth	DeptName	TotCredits
00128	Zhang	1992-04-18	Comp. Sci.	102
12345	Shankar	1995-12-06	Comp. Sci.	32
19991	Brandt	1993-05-24	History	80
23121	Chavez	1992-04-18	Finance	110
44553	Peltier	1995-10-18	Physics	56
45678	Levy	1995-08-01	Physics	46
54321	Williams	1995-02-28	Comp. Sci.	54
55739	Sanchez	1995-06-04	Music	38
70557	Snow	1995-11-22	Physics	0
76543	Brown	1994-03-05	Comp. Sci.	58
76653	Aoi	1993-09-18	Elec. Eng.	60
98765	Bourikas	1992-09-23	Elec. Eng.	98
98988	Tanaka	1992-06-02	Biology	120

Advisor

StudID	InstID
12345	10101
44553	22222
45678	22222
00128	45565
76543	45565
23121	76543
98988	76766
76653	98345
98765	98345

Department

DeptName	Building	Budget
Biology	Watson	90000.00
Comp. Sci.	Taylor	100000.00
Elec. Eng.	Taylor	85000.00
Finance	Painter	120000.00
History	Painter	50000.00
Music	Packard	80000.00
Physics	Watson	70000.00

Database Instance (2 of 4)

Teaches

InstID	CourseID	SectionID	Semester	StudyYear
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
10101	CS-101	1	Fall	2009
45565	CS-101	1	Spring	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
10101	CS-315	1	Spring	2010
45565	CS-319	1	Spring	2010
83821	CS-319	2	Spring	2010
10101	CS-347	1	Fall	2009
98345	EE-181	1	Spring	2009
12121	FIN-201	1	Spring	2010
32343	HIS-351	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

Takes

StudID	CourseID	SectionID	Semester	StudyYear	Grade
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	NULL

Database Instance (3 of 4)

Course

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

PreReq

CourseID	PreReqID
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Database Instance (4 of 4)

Section

CourseID	SectionID	Semester	StudyYear	Building	Room	TimeSlotID
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Classroom

Building	Room	Capacity
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

TimeSlot

TimeSlotID	DayCode	StartTime	EndTime
A	M	08:00:00	08:50:00
A	W	08:00:00	08:50:00
A	F	08:00:00	08:50:00
B	M	09:00:00	09:50:00
B	W	09:00:00	09:50:00
B	F	09:00:00	09:50:00
C	M	11:00:00	11:50:00
C	W	11:00:00	11:50:00
C	F	11:00:00	11:50:00
D	M	13:00:00	13:50:00
D	W	13:00:00	13:50:00
D	F	13:00:00	13:50:00
E	T	10:30:00	11:45:00
E	R	10:30:00	11:45:00
F	T	14:30:00	15:45:00
F	R	14:30:00	15:45:00
G	M	16:00:00	16:50:00
G	W	16:00:00	16:50:00
G	F	16:00:00	16:50:00
H	W	10:00:00	12:30:00

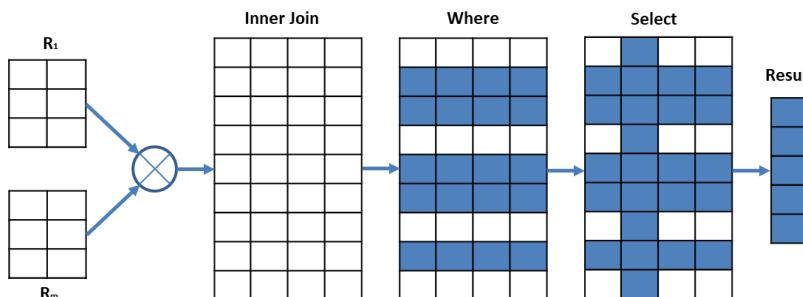
Note: When writing and testing SQL queries it is useful to have a print of the Diagram and the Database Instance.

SQL Data Queries

- **Basic Query Structure**
 - SQL Data Query Language provides the ability to query data
 - The result of a SQL Query is a table
- A typical *basic* SQL query has the form:

```
SELECT A1, A2, ..., An  
FROM r1, r2, ..., rm  
WHERE P;
```

- A_i represents an attribute
- r_i represents a relation (can e.g. be an id or a [natural] join expression)
- P is a (row) predicate over attribute names. For each row it is either true or false.



String Operations (section 3.4.2)

- Strings are sequences of characters enclosed with quotes
 - E.g. 'Anne'
 - How to include ' and \ in strings: 'Anne\'s' and 'A\\B'
- SQL supports a variety of string operations such as
 - =, <>, <, >: are case sensitive according to the SQL standard, **but not in MySQL and MariaDB**, where e.g. 'anne' = 'Anne' evaluates to 1 (true)
 - Concatenation, using **CONCAT**
 - Finding string length, extracting substrings, etc, see the DBMS manual.
 - Pattern matching, using **LIKE**
- Examples

```
SET @TeachingHistory = CONCAT('CS-101', ' ', 'CS-315', ' ', 'CS-347'); # @ defines a variable
SELECT @TeachingHistory;
```

@TeachingHistory
CS-101, CS-315, CS-347

String Operations: Pattern Matching

- **LIKE** is a string-matching operator for comparisons of character strings:
 - Syntax: `string-expr LIKE string-pattern`
 - Can be used where a Boolean expression is expected, e.g. in a `WHERE` clause.
 - Returns 1 (true) if `string-expr` matches `string-pattern`, otherwise 0 (false).
 - The `string-pattern` can use two special characters:
 - The `%` character matches any substring (of 0 - n characters).
 - The `_` character matches any (single) character.
- Pattern matching examples
 - `'Anne'` matches `'Anne'`, but not `'Hanne'`
 - `'Intro%'` matches any string beginning with “Intro”, e.g. `'Introduction'`
 - `'%duc%'` matches any string containing “duc” as a substring”, e.g. `'Introduction'`
 - `'___'` matches any string of exactly three characters, e.g. `'Ann'`
 - `'___ %'` matches any string of at least three characters, e.g. `'Hanne'`

String Operations: Pattern Matching

- Patterns are case sensitive according to the SQL standard, but not in MySQL and MariaDB where e.g.
 - 'anne' **LIKE** 'Anne'will evaluate to 1.
- **Query Example using pattern matching:** Find the names of all instructors whose name includes the substring “ri”.

```
SELECT InstName FROM Instructor  
WHERE InstName LIKE '%ri%';
```

InstName
Califieri
Crick
Srinivasan

NULL Values in Expressions (section 3.6)

- A row can have a **NULL** value for some attributes.
 - **NULL** signifies an unknown value or that a value does not exist.
- Arithmetic Expressions containing **NULL** return **NULL**.
 - Example: $5 + \text{NULL}$ returns **NULL**
- The predicate **IS NULL** can be used to check for Null.
 - Example: Find all instructors whose salary is unknown.
`SELECT InstName FROM Instructor WHERE Salary IS NULL;`
- The predicate **IS NOT NULL** can be used to check for not Null

NULL and Three Valued Logic

- Comparison Expressions containing **NULL** return **UNKNOWN**
 - Examples:
 $5 < \text{NULL}$, $\text{NULL} \leftrightarrow \text{NULL}$, and $\text{NULL} = \text{NULL}$ all evaluate to **UNKNOWN**
- Three-valued (conditional) logic using the truth value **UNKNOWN**

OR	FALSE	TRUE	UNKNOWN
FALSE	FALSE	TRUE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
UNKNOWN	UNKNOWN	TRUE	UNKNOWN

AND	FALSE	TRUE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
TRUE	FALSE	TRUE	UNKNOWN
UNKNOWN	FALSE	UNKNOWN	UNKNOWN

NOT	FALSE	TRUE	UNKNOWN
Result	TRUE	FALSE	UNKNOWN

- A **WHERE** or **HAVING** clause predicate is treated as **FALSE**, if it evaluates to **UNKNOWN**
- The predicate **IS [NOT] UNKNOWN**:
 - “**P IS UNKNOWN**” evaluates to **TRUE**, if predicate **P** evaluates to **UNKNOWN**

Treatment of **NULL** By **SELECT DISTINCT**

- **SELECT DISTINCT ... FROM ...** removes duplicate rows.
 - When comparing two tuples, it treats **NULL** as being equal to **NULL** (although **NULL = NULL** returns **UNKNOWN**).
 - **Example 1:** If there are two rows **(1, NULL)** and **(1, NULL)**, one of them are removed.
 - **Example 2:** If there are two rows **(1, 2)** and **(1, NULL)**, both are kept.

Aggregate Functions (section 3.7)

- These functions operate on the set of values in a column of a given attribute A and return a value:

AVG(A): Average of values in the A-column

MIN(A): Minimum of values in the A-column

MAX(A): Maximum of values in the A-column

SUM(A): Sum of values in the A-column

COUNT(A): Number of values in the A-column

- They can be used in **SELECT clauses** and **HAVING clauses**.
- Example:

```
SELECT AVG(Salary), MIN(Salary), MAX(Salary), SUM(Salary), COUNT(Salary)  
FROM Instructor;
```

AVG(Salary)	MIN(Salary)	MAX(Salary)	SUM(Salary)	COUNT(Salary)
75416.666667	40000.00	95000.00	905000.00	12

- COUNT(*)**: counts the number of rows in a relation/table.
- Example: **SELECT COUNT(*) FROM Course;**

COUNT(*)

13

NULL Values and Aggregates

- Find the total of all instructor salaries
 - `SELECT SUM(Salary) FROM Instructor;`
 - This statement ignores **NULL** amounts and sums the rest.
 - Result is **NULL**, only if ALL salaries are **NULL**.
- **AVG(A), MIN(A), MAX(A), SUM(A), and COUNT(A)** ignore rows where A is **NULL**.
- What if an A column only has **NULL** values or is empty?
 - **COUNT(A)** returns 0,
 - **MIN(A), MAX(A), SUM(A), and AVG(A)** return **NULL**
- **COUNT(*) does not ignore rows with NULL values.**

Duplicates and Aggregates

- Aggregate functions take **duplicate values** of the aggregate attribute **into account**.
- Example: Find the average salary of instructors in the Computer Science department

```
SELECT AVG(Salary) FROM Instructor  
WHERE DeptName='Comp. Sci.');
```

- To ignore **duplicate values**, use the **DISTINCT** keyword.
- Example: Find the total number of instructors who teach a course in the Spring 2010 semester

```
SELECT COUNT(DISTINCT InstID) FROM Teaches  
WHERE Semester='Spring' AND StudyYear=2010;
```

GROUP BY

- Find the name and average instructor salary for each department:

```
SELECT * FROM Instructor ORDER BY DeptName;
```

InstID	InstName	DeptName	Salary
76766	Crick	Biology	72000.00
10101	Srinivasan	Comp. Sci.	65000.00
83821	Brandt	Comp. Sci.	92000.00
45565	Katz	Comp. Sci.	75000.00
98345	Kim	Elec. Eng.	80000.00
76543	Singh	Finance	80000.00
12121	Wu	Finance	90000.00
32343	El Said	History	60000.00
58583	Califieri	History	62000.00
15151	Mozart	Music	40000.00
33456	Gold	Physics	87000.00
22222	Einstein	Physics	95000.00

```
SELECT DeptName, AVG(Salary)  
FROM Instructor  
GROUP BY DeptName;
```

DeptName	AVG(Salary)
Biology	72000.000000
Comp. Sci.	77333.333333
Elec. Eng.	80000.000000
Finance	85000.000000
History	61000.000000
Music	40000.000000
Physics	91000.000000

Aggregation with GROUP BY

- Study the table Testscores and possible Aggregates

```
SELECT * FROM Testscores;
```

Student	Test	Score
Brandt	A	47
Brandt	B	50
Brandt	C	NULL
Brandt	D	NULL
Chavez	A	52
Chavez	B	45
Chavez	C	53
Chavez	D	NULL

```
SELECT Student,  
       COUNT(Score) AS n,  
       SUM(Score) AS Total,  
       AVG(Score) AS Average,  
       MIN(Score) AS Lowest,  
       MAX(Score) AS Highest  
  FROM Testscores GROUP BY Student;
```

Student	n	Total	Average	Lowest	Highest
Brandt	2	97	48.5000	47	50
Chavez	3	150	50.0000	45	53

```
SELECT COUNT(*) FROM Testscores;
```

COUNT(*)
8

HAVING

- Find the name and average salary for those departments whose average salary is greater than 65000.

```
SELECT DeptName, AVG(Salary) FROM Instructor  
GROUP BY DeptName HAVING AVG(Salary) > 65000;
```

DeptName	AVG(Salary)
Biology	72000.000000
Comp. Sci.	77333.333333
Elec. Eng.	80000.000000
Finance	85000.000000
Physics	91000.000000

- Note: Predicates in the **HAVING** clause are applied after the formation of groups, whereas predicates in the **WHERE** clause are applied before forming groups.
So, **WHERE** is working on each Row and **HAVING** on each Group result.

GROUP BY

- Typical form

```
SELECT attribute-list1, aggregations  
FROM ... WHERE P1  
GROUP BY attribute-list2 HAVING P2
```

- Attributes in **attribute-list1** must be present in **attribute-list2**.
- Usually **attribute-list1** = **attribute-list2**.
- In **P2** there can be aggregations (like in the SELECT clause).
Attributes that appear in **P2** without being aggregated over, must be present in **attribute-list2**.

More General SQL Queries

- A more general form of SELECT queries:

```
SELECT attributes
  FROM r1, r2, ..., rm [WHERE P1]
  [GROUP BY group-spec [HAVING P2]]
  [ORDER BY order-spec];
```

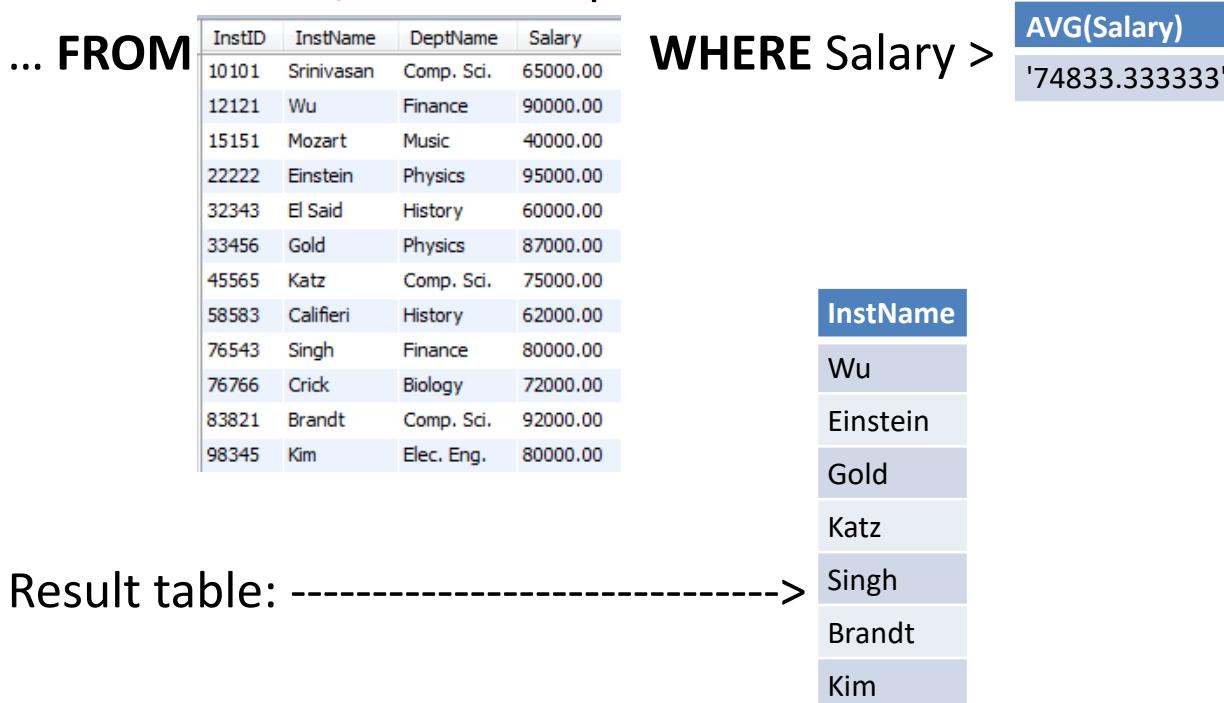
- Meaning:
 1. Calculate the relation r represented by r_1, r_2, \dots, r_m and remove rows from r not satisfying $P1$.
 2. Arrange the selected rows into groups having the same values for $group\text{-spec}$ and remove groups not satisfying $P2$.
 3. For each group calculate the $attributes$: this gives one tuple/row for each group.
 4. Order the rows according to the $order\text{-spec}$.

Subqueries (section 3.8)

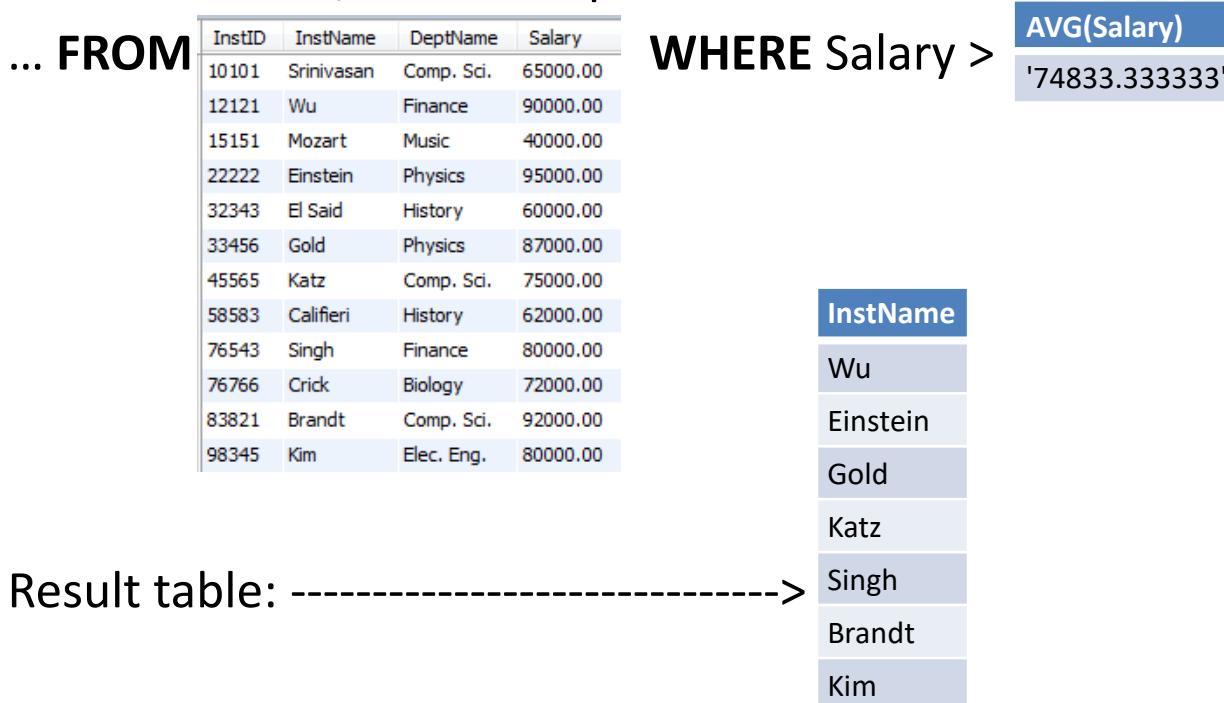
- **What:**
 - A SELECT statement is said to be a **subquery**, if it occurs nested inside another statement.
- **Where:**
 - A SELECT statement can be used to represent a relation in the WHERE, FROM and HAVING clauses of another (outer) SELECT statement.
- **Examples:**
 - Will be shown on the following slides.
- **Purpose:**
 - They provide alternative ways to perform operations that would otherwise require complex joins and unions.

Scalar Subqueries (section 3.8.7)

- If a subquery returns a **1x1** relation then it is said to be a **scalar subquery**.
- A scalar subquery can be used where a single value is expected (in the **SELECT, WHERE, FROM** and **HAVING** clauses). Example:
 - **SELECT InstName FROM Instructor WHERE Salary > (SELECT AVG(Salary) FROM Instructor);**
 - For an instance, this corresponds to:

... **FROM** 

InstID	InstName	DeptName	Salary	AVG(Salary)
10101	Srinivasan	Comp. Sci.	65000.00	
12121	Wu	Finance	90000.00	
15151	Mozart	Music	40000.00	
22222	Einstein	Physics	95000.00	
32343	El Said	History	60000.00	
33456	Gold	Physics	87000.00	
45565	Katz	Comp. Sci.	75000.00	
58583	Califieri	History	62000.00	
76543	Singh	Finance	80000.00	
76766	Crick	Biology	72000.00	
83821	Brandt	Comp. Sci.	92000.00	
98345	Kim	Elec. Eng.	80000.00	

WHERE Salary > 

InstName
Wu
Einstein
Gold
Katz
Singh
Brandt
Kim

Set Membership Conditions: IN and NOT IN (section 3.8.1)

- IN and NOT IN can be used in a WHERE/HAVING clause to form a condition.
- Typical forms:
 - **SELECT...**, A_i,... **FROM** ...
WHERE A_i [NOT] **IN** (value₁, value₂, ...);
 - **SELECT...**, A_i,... **FROM** ...
WHERE A_i [NOT] **IN** (SELECT B_j **FROM** ... **WHERE** ...); #often A_i = B_j
- Example: select the names of instructors whose name is neither Mozart, Einstein:
 - **SELECT DISTINCT** InstName **FROM** Instructor
WHERE InstName **NOT IN** ('Mozart', 'Einstein');
- Example: Find courses offered both in Fall 2009 and in Spring 2010
SELECT DISTINCT CourseID **FROM** Section
WHERE
 Semester='Fall' **AND** StudyYear=2009 **AND**
 CourseID **IN**
 (**SELECT** CourseID **FROM** Section **WHERE** Semester='Spring' **AND** StudyYear=2010);

Conditions Using ALL and SOME (section 3.8.2)

- Can be used in a WHERE/HAVING clause to form a condition.
- Typical forms:
 - $A \text{ op ALL (SELECT A FROM ... WHERE ...);}$
 - $A \text{ op SOME (SELECT A FROM ... WHERE ...);}$
- where op can be: $<$, \leq , $>$, $=$, \neq
- It checks whether $A \text{ op } v$ is true for all/some of the values v in the column specified by the SELECT.

SOME

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

SELECT InstName **FROM** Instructor

WHERE Salary > **SOME** (**SELECT** Salary **FROM** Instructor

WHERE DeptName = 'Finance');

- An instance of this corresponds to:

SELECT InstName **FROM**

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

WHERE Salary > **SOME**

Salary
90000
80000

- Which gives:

InstName
Wu
Einstein
Gold
Brandt

SOME conditions

- $A \text{ op } \text{SOME } r \Leftrightarrow \exists v \in r : (A \text{ op } v)$

Where r is a relation with one attribute and
 op can be: $<$, \leq , $>$, $=$, \neq

$(5 < \text{SOME} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{true}$

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

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

$(5 \neq \text{SOME} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true} \text{ (since } 0 \neq 5\text{)}$

$(= \text{SOME}) \equiv \text{IN}$

However, $(\neq \text{SOME}) \not\equiv \text{NOT IN}$

ALL conditions

- $A \text{ op } \text{ALL } r \Leftrightarrow \forall v \in r : (A \text{ op } v)$

$(5 < \text{ALL} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{false}$

$(5 < \text{ALL} \begin{array}{|c|} \hline 6 \\ \hline 10 \\ \hline \end{array}) = \text{true}$

$(5 = \text{ALL} \begin{array}{|c|} \hline 4 \\ \hline 5 \\ \hline \end{array}) = \text{false}$

$(5 \neq \text{ALL} \begin{array}{|c|} \hline 4 \\ \hline 6 \\ \hline \end{array}) = \text{true}$

$(\neq \text{ALL}) \equiv \text{NOT IN}$

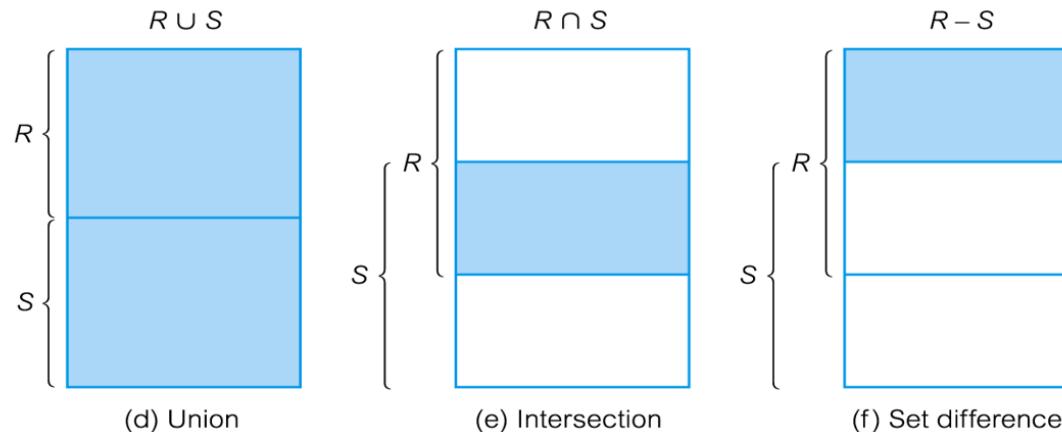
But: $(= \text{ALL}) \not\equiv \text{IN}$

Conditions Using EXISTS and NOT EXISTS (section 3.8.3)

- Can be used in a WHERE/HAVING clause to form a condition:
 - [NOT] EXISTS (SELECT ... FROM ... WHERE ...);
- It checks whether the relation is [not] non-empty
- Example: Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester.
 - **SELECT CourseID FROM Section AS S
WHERE Semester='Fall' AND StudyYear=2009 AND
EXISTS (SELECT *
 FROM Section AS T
 WHERE Semester = 'Spring' AND StudyYear= 2010
 AND S.CourseID = T.CourseID);**

Compound Queries Using UNION, INTERSECT & EXCEPT (section 3.5)

- Set Operations **UNION, INTERSECT & EXCEPT** have
 - Arguments:** two SELECT statements (denoting two relations R and S).
 - R and S **must be compatible:** Number and types of the attributes of R and S must be the same.
 - Result:** is the relation consisting of the union/intersection and set difference of the tuples in R and S .
 - UNION** and **INTERSECT** removes duplicates in the result.
 - EXCEPT** removes duplicates in its arguments before the operation is done.



- Applications of set operations constitute a query (like SELECT statements do).
- They can't appear inside a SELECT statement.
- Set INTERSECT and EXCEPT are implemented in MariaDB, but **not in MySQL**, where they instead be expressed by nested subqueries.

UNION, INTERSECT and EXCEPT

- Find courses offered in Fall 2009 or in Spring 2010

```
(SELECT CourseID FROM Section WHERE Semester='Fall' AND StudyYear=2009)  
UNION  
(SELECT CourseID FROM Section WHERE Semester='Spring' AND StudyYear = 2010);
```

R	S	R UNION S	R INTERSECT S	R EXCEPT S
CS-101	CS-101	CS-101		
CS-347	FIN-201	CS-347	CS-101	
PHY-101	MU-199	PHY-101		
	HIS-351			
	CS-319			
	CS-319			
	CS-315			

INTERSECT and EXCEPT

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

(SELECT CourseID FROM Section WHERE Semester='Fall' AND StudyYear=2009)

INTERSECT

(SELECT CourseID FROM Section WHERE Semester='Spring' AND StudyYear = 2010);

can be expressed by:

SELECT DISTINCT CourseID FROM Section

WHERE Semester='Fall' AND StudyYear=2009 AND CourseID

IN (SELECT CourseID FROM Section WHERE Semester='Spring' AND StudyYear=2010);

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

(SELECT CourseID FROM Section WHERE Semester='Fall' AND StudyYear=2009)

EXCEPT

(SELECT CourseID FROM Section WHERE Semester='Spring' AND StudyYear = 2010);

can be expressed by:

SELECT DISTINCT CourseID FROM Section

WHERE Semester='Fall' AND StudyYear=2009 AND CourseID

NOT IN (SELECT CourseID FROM Section WHERE Semester='Spring' AND StudyYear=2010);



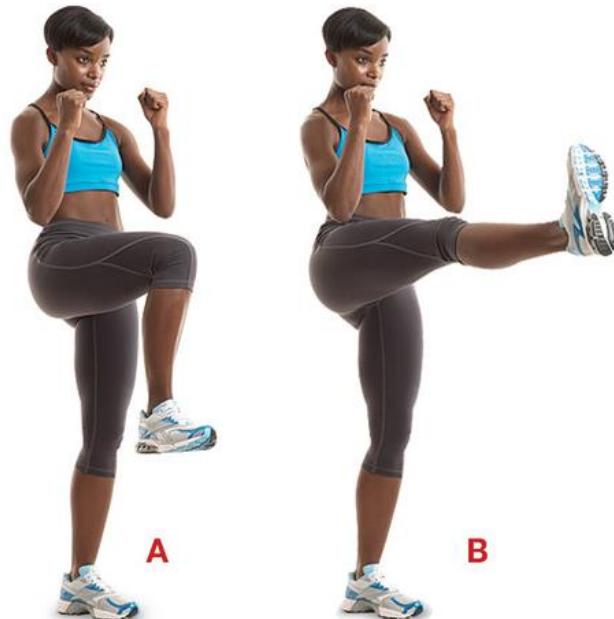
Summary

- **Data Queries:**
 - SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ... ORDER BY ...
 - ... UNION ...,
 - ... INTERSECT ...,
 - ... EXCEPT ...

Readings

- In Database Systems Concepts please complete reading Chapter 3
- Pay special attention to the Summary

Demo Exercises



Demo Exercises clarify ideas and concepts from the Lecture to provide you with good Database Skills.

Do the Demo Exercises. Solutions can be found on a later slide.

Advanced SQL Queries and Data Manipulation

The following SQL queries should be run against the University database.

3.1.1 SELECT with COUNT aggregation and GROUP BY

Find the number of enrolments of each course section that was offered in Fall 2009.

3.1.2 SELECT with SUM aggregation and multiple JOINs and GROUP BY

What is the course title and sum of course credits of the courses taught by instructor Brandt?

3.1.3 DELETE with NOT IN and nested SELECT

Delete all courses that have never been offered (that is, do not occur in the Section table).

Solutions to Demo Exercises



Advanced SQL Queries and Data Manipulation

The following SQL queries should be run against the University database.

3.1.1 SELECT with COUNT aggregation and GROUP BY

Find the number of enrolments of each course section that was offered in Fall 2009.

```
SELECT CourseID, SectionID, Count(StudID)
FROM Takes
WHERE Semester = 'Fall' AND StudyYear = 2009
GROUP BY CourseID, SectionID;
```

3.1.2 SELECT with SUM aggregation and multiple JOINs and GROUP BY

What is the course title and sum of course credits of the courses taught by instructor Brandt?

```
SELECT Title, SUM(Credits)
FROM Instructor
    NATURAL JOIN Teaches
    NATURAL JOIN Course
WHERE InstName = 'Brandt'
GROUP BY Title;
```

Title	SUM(Credits)
Game Design	8
Image Processing	3

3.1.3 DELETE with NOT IN and nested SELECT

Delete all courses that have never been offered (that is, do not occur in the Section table).

```
DELETE FROM Course
WHERE CourseID NOT IN
(Select CourseID FROM Section);
```

PS. Run the UniversityDB.sql script again to restore tables to the original instance.

Exercises

Please answer all exercises
to demonstrate your
Database Skills.

MySQL Exercises
Solutions are available at 11:45



Advanced SQL Queries and Data Manipulation

Exercises

SQL queries should be run against the University database.

Re-run the UniversityDB Script to restore tables to initial instances.

Have a print of the University Database Schema Diagram and the Database Instance readily available.

3.2.1 SELECT with MAX aggregation

Find the highest salary of any instructor

3.2.2 Scalar Subquery

Find all instructors earning the highest salary. There might be more than one with the same salary.

3.2.3 DELETE using IN

Delete courses BIO-101 and BIO-301 in the Takes table.

After this question run the UniversityDB Script to restore tables to initial instances.

3.2.4 Advanced WHERE condition using NOT IN

Find those students who have not taken a course.

3.2.5 Advanced WHERE condition using ALL

Find the name(s) of those department(s) which have the highest budget, i.e. a budget which is higher than or equal to those for all other departments.

3.2.6 Advanced WHERE condition using SOME

Find the names of those students who have the same name as some instructor. Use the SOME operator for this.

Make another statement querying the same, but without using SOME.

Create Table & Advanced SQL Queries

3.2.7 Create and populate a table

GradePoints(Grade, Points) to provide a conversion from letter grades to numeric scores such that

```
SELECT * FROM GradePoints;
```

gives:

Grade	Points
A	4.0
A-	3.7
B	3.0
B+	3.3
B-	2.7
C	2.0
C+	2.3
C-	1.7
D	1.0
D+	1.3
D-	0.7
F	0.0

This shows that an “A” corresponds to 4 points, an “A-” to 3.7 points, and so on.

The **Grade-Points** earned by a student **for a course offering** (section) is the number of credits for the course multiplied by the points for the grade that the student received.

The **Total Grade-Points** earned by a student is the sum of grade points for all courses taken by the students.

3.2.8 Find the Total Grade-Points

earned by each student who has taken a course. Hint: use **GROUP BY**.

3.2.9 Find the Total Grade-Points Average (GPA)

for each student who has taken a course, that is, the total grade-points divided by the total credits for the associated courses. Order the students by falling averages.

Hint: use **GROUP BY**.

Create Table & Advanced SQL Queries

3.2.10 Query using UNION

Now modify the queries from the two previous questions such that students who have not taken a course are also included in the result.

Hint: use the UNION operator.

3.2.11 Testscores Example

Create a relation schema Testscores (by a table declaration) and insert values such that

`SELECT * FROM Testscores;`
gives:

Student	Test	Score
Brandt	A	47
Brandt	B	50
Brandt	C	NULL
Brandt	D	NULL
Chavez	A	52
Chavez	B	45
Chavez	C	53
Chavez	D	NULL

Then find the maximal score for each student who has an average larger than 49.

Then find those students for whom some score is unknown.



Technical University of Denmark

DTU Compute

Department of Applied Mathematics and Computer Science

Intermediate SQL

Joins, Constraints, Views and Authorization

Chapter 4 in the Textbook

©Anne Haxthausen

Some of the slides have been based on some slides by Flemming Schmidt. Some examples are taken from the textbook by Silberschatz, Korth, and Sudarshan.

Contents

- JOINs (section 4.1)
- Constraints (section 4.4-4.5)
- Views (section 4.2)
- Authorization (section 4.6)
- Demo Exercises & Exercises
- Appendix: University Database

JOIN Operations

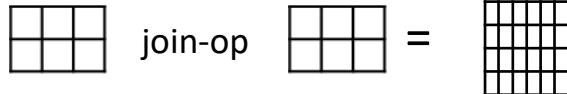
- There are *several kinds* of join operations.
- They take two tables as arguments and return a table as result:
The diagram illustrates the concept of a join operation. It shows two small tables on the left, each with 3 rows and 3 columns, separated by a 'join-op' symbol. An equals sign follows the join operator. To the right is a larger table with 6 rows and 6 columns, representing the result of the join operation.
- The result table is a subset of the Cartesian Product of the argument tables, which
 - combines rows of the two tables, if they match some *join condition*
 - may or may not retain rows that do not match rows in the other table (by padding NULLs for missing data in the other row)
 - include more or less of the attributes
- JOIN operations are typically used in the FROM clause of a SELECT command.

Table Examples

- In the following examples two tables Courses and PreReqs have been introduced, and the following instances (being subtables of the running example instances of Course and PreReq in order to be on one page) are considered.

Courses

CourseID	Title	DeptName	Credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

PreReqs

CourseID	PreReqID
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Cartesian Product = (INNER) JOIN

Courses

CourseID	Title	DeptName	Credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

Note: CS-315 is not listed in PreReqs!

PreReqs

CourseID	PreReqID
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Note: CS-347 is not listed in Courses!

SELECT * FROM Courses, PreReqs;

#Cartesian Product

SELECT * FROM Courses **JOIN PreReqs;**

#Equivalent expression

CourseID	Title	DeptName	Credits	CourseID	PreReqID
BIO-301	Genetics	Biology	4	BIO-301	BIO-101
CS-190	Game Design	Comp. Sci.	4	BIO-301	BIO-101
CS-315	Robotics	Comp. Sci.	3	BIO-301	BIO-101
BIO-301	Genetics	Biology	4	CS-190	CS-101
CS-190	Game Design	Comp. Sci.	4	CS-190	CS-101
CS-315	Robotics	Comp. Sci.	3	CS-190	CS-101
BIO-301	Genetics	Biology	4	CS-347	CS-101
CS-190	Game Design	Comp. Sci.	4	CS-347	CS-101
CS-315	Robotics	Comp. Sci.	3	CS-347	CS-101

In general:

All attributes are included.

All row combinations are included.

Attributes(R1, R2) =

Attributes(R1) union Attributes(R2)

#Rows(R1, R2) = #Rows(R1) * #Rows(R2)

NATURAL (INNER) JOIN

Courses

CourseID	Title	DeptName	Credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

PreReqs

CourseID	PreReqID
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

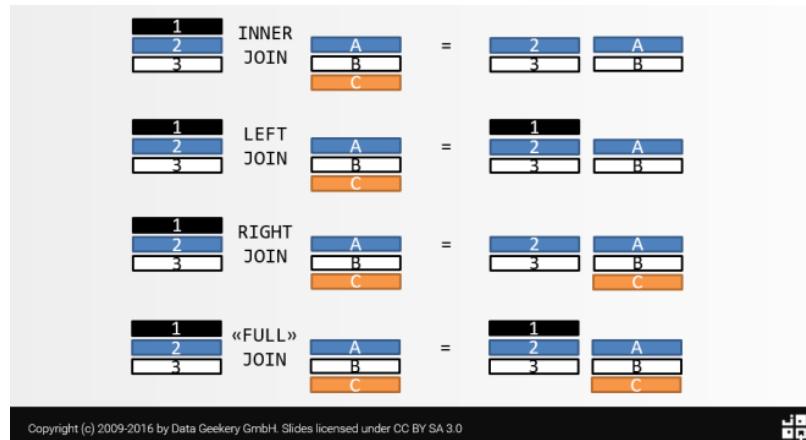
SELECT * FROM Courses NATURAL JOIN PreReqs;

CourseID	Title	DeptName	Credits	PreReqID
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101

- **Included attributes:** All, but only one occurrence of common attributes (`CourseID`).
- **Matching:** Only rows that have the same value of the common attributes (`CourseID`) are joined.
- The `CourseID CS-315` row in Courses is not in the result, because the JOIN cannot find a matching row in PreReqs. Same situation for `CourseID CS-347` in PreReqs. So **NATURAL JOIN** *might lose important information*.

OUTER JOINS

- Three kinds of **OUTER JOINs** to avoid loosing info:
 - LEFT OUTER JOIN** preserves the rows in the left table
 - RIGHT OUTER JOIN** preserves the rows in the right table
 - FULL OUTER JOIN** preserves the rows in both tables
- Keyword **OUTER** can be left out without changing the meaning.
- Each outer join first computes the corresponding inner join and then adds tuples where NULL is padded for missing data:



NATURAL LEFT OUTER JOIN

Courses

CourseID	Title	DeptName	Credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

PreReqs

CourseID	PreReqID
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

SELECT * FROM Courses NATURAL LEFT OUTER JOIN PreReqs;

CourseID	Title	DeptName	Credits	PreReqID
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	NULL

- Note: The result contains the left table with extra right table attributes.

NATURAL RIGHT OUTER JOIN

Courses

CourseID	Title	DeptName	Credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

PreReqs

CourseID	PreReqID
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

SELECT * FROM Courses NATURAL RIGHT OUTER JOIN PreReqs;

CourseID	PreReqID	Title	DeptName	Credits
BIO-301	BIO-101	Genetics	Biology	4
CS-190	CS-101	Game Design	Comp. Sci.	4
CS-347	CS-101	NULL	NULL	NULL

- Note: The result contains the right table with extra left table attributes
- Same as **SELECT * FROM Prereqs NATURAL LEFT OUTER JOIN Courses;**

NATURAL FULL OUTER JOIN

Courses

CourseID	Title	DeptName	Credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

PreReqs

CourseID	PreReqID
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

SELECT * FROM Courses NATURAL FULL OUTER JOIN PreReqs;

CourseID	Title	DeptName	Credits	PreReqID
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

- **NATURAL FULL OUTER JOIN** *is not supported by MariaDB and MySQL*, instead use:

**(SELECT * FROM Courses NATURAL LEFT OUTER JOIN PreReqs)
UNION
(SELECT CourseID, Title, DeptName, Credits, PreReqID
FROM Courses NATURAL RIGHT OUTER JOIN PreReqs);**

JOIN Conditions

- JOIN conditions define which rows in the two argument tables $T1$ and $T2$ match, and which attributes are present in the result of the JOIN:
 - $T1$ NATURAL join-op $T2$:
 - two rows match, if $T1.A = T2.A$ for each common attribute name A
 - common attribute names only occur ones in the result
 - $T1$ join-op $T2$ USING (A_1, \dots, A_n) , where A_1, \dots, A_n are (a subset of) common attributes:
 - two rows match, if $T1.A_1 = T2.A_1$ and ... and $T1.A_n = T2.A_n$
 - A_1, \dots, A_n only occur ones in the result
 - compared to NATURAL, it allows to *not join on all* common attributes
 - $T1$ join-op $T2$ ON *predicate*:
 - two rows match, if *predicate* is true
 - all attributes are included in the result
 - allows e.g. for ‘joining’ attributes named differently in the two tables

where join-op can be INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN,

where INNER and OUTER can be left out (without changing the meaning).

- Note: $T1$ join-op $T2$ USING (A_1, \dots, A_n) = $T1$ NATURAL join-op $T2$,
if A_1, \dots, A_n include *all* common attributes.
- Note: $T1$ JOIN $T2$ is short for $T1$ JOIN $T2$ ON TRUE

Examples of USING (A₁, A₂, ..., A_n)

SELECT * FROM Courses JOIN PreReqs **USING (CourseID);**

SELECT * FROM Courses **NATURAL JOIN PreReqs; # equivalent expression**

CourseID	Title	DeptName	Credits	PreReqID
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101

SELECT * FROM Courses **RIGHT OUTER JOIN PreReqs **USING** (CourseID);**

SELECT * FROM Courses **NATURAL RIGHT OUTER JOIN PreReqs; # equivalent expression**

CourseID	PreReqID	Title	DeptName	Credits
BIO-301	BIO-101	Genetics	Biology	4
CS-190	CS-101	Game Design	Comp. Sci.	4
CS-347	CS-101	NULL	NULL	NULL

Examples of JOIN ON Predicate

SELECT * FROM Courses JOIN PreReqs **ON Courses.CourseID = PreReqs.CourseID;**

SELECT * FROM Courses JOIN PreReqs **WHERE Courses.CourseID = PreReqs.CourseID;**

CourseID	Title	DeptName	Credits	CourseID	PreReqID
BIO-301	Genetics	Biology	4	BIO-301	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-190	CS-101

SELECT * FROM Courses LEFT OUTER JOIN PreReqs **ON Courses.CourseID = PreReqs.CourseID;**

CourseID	Title	DeptName	Credits	CourseID	PreReqID
BIO-301	Genetics	Biology	4	BIO-301	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-190	CS-101
CS-315	Robotics	Comp. Sci.	3	NULL	NULL

SELECT * FROM Courses LEFT OUTER JOIN PreReqs **ON TRUE**

WHERE Courses.CourseID = PreReqs.CourseID;

CourseID	Title	DeptName	Credits	CourseID	PreReqID
BIO-301	Genetics	Biology	4	BIO-301	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-190	CS-101

IMPORTANT: **ON** is a part of the join operation, while **WHERE** is calculated afterwards!

JOIN Summary

- **Syntax:**
 - $T1 \text{ JOIN } T2$ (equivalent to: $T1, T2$)
 - $T1 \text{ NATURAL join-op } T2$
 - $T1 \text{ join-op } T2 \text{ USING } (A_1, \dots, A_n)$, where A_1, \dots, A_n are (a subset of) common attributes
 - $T1 \text{ join-op } T2 \text{ ON } \text{predicate}$
- **JOIN conditions** define which rows in the two tables match, and which attributes are present in the result of the JOIN:
 - **NATURAL:**
 - two rows match, if $T1.A = T2.A$ for each common attribute name A
 - common attribute names only occur ones in the result
 - **USING** (A_1, \dots, A_n):
 - two rows match, if $T1.A_1 = T2.A_1$ and ... and $T1.A_n = T2.A_n$
 - A_1, \dots, A_n only occur ones in the result
 - **ON predicate:**
 - two rows match, if **predicate** is true
 - all attributes are included in the result
- **join-op** defines how rows in each table that do not match any row in the other table (based on the **JOIN condition**) are treated:
 - **INNER JOIN:** unmatched rows are not included
 - **LEFT OUTER JOIN:** unmatched rows in the left table are included, with NULLs for missing right table values.
 - **RIGHT OUTER JOIN:** unmatched rows in the right table are included, with NULLs for missing left table values.
 - **FULL OUTER JOIN:** as both for **LEFT** and for **RIGHT**

Constraints

■ Integrity constraints

- Express **rules about allowed data** in database tables.
- Can be **specified** in CREATE TABLE (or ALTER TABLE ADD) statements.
- If a data changing operation would lead to a **violation** of a specified constraint, the DBMS will **reject** it (if the feature is implemented by the DBMS).
- Purpose: to guard against wrong and inconsistent data in a database.

■ Integrity constraints on a single table

- **Data types**
- **NOT NULL**
- **PRIMARY KEY**
- ...

■ Integrity constraints between two tables

- **FOREIGN KEY** Referential Integrity

Further checks can be made by programming **Triggers**, that fire (i.e. executes), when the DBMS gets an INSERT, UPDATE or DELETE command.

Data Types as Attribute Value Constraints

- The data type given to an attribute represents a constraint:
 - the attribute must only be assigned values of the given type.
 - Data Types in MariaDB¹:
 - Numeric data types, e.g. **INT**, **DECIMAL(i,j)**, ...
 - String data types, e.g. **VARCHAR(n)**, **ENUM(s₁, ..., s_n)**
 - Date and Time datatypes, e.g. **DATE**, **TIME**, **YEAR**
 - ...
- For more info, see the MariaDB documentation here:
<https://mariadb.com/kb/en/library/documentation/>

1: The data types depend on the DBMS implementation.

Example of Attribute Value Constraint

- Ensure the Semester attribute values are checked

```
CREATE TABLE Section (
    CourseID      VARCHAR(8),
    SectionID     VARCHAR(8),
    Semester      ENUM('Fall', 'Winter', 'Spring', 'Summer'),
    StudyYear     DECIMAL(4,0),
    Building      VARCHAR(15),
    Room          VARCHAR(7),
    TimeSlotID    VARCHAR(4),
    PRIMARY KEY(CourseID, SectionID, Semester, StudyYear) );
```

```
INSERT Section VALUES('BIO-101','1','Summer','2009','Painter','514','B'); -- is ok
```

```
INSERT Section VALUES('BIO-101','1','Sommer','2009','Painter','514','B'); -- is not ok
```

NOT NULL and PRIMARY KEY Constraints

■ NOT NULL

- In CREATE TABLE declare LastName and Budget to be NOT NULL:

LastName VARCHAR(20) **NOT NULL**;

Budget DECIMAL(12,2) **NOT NULL**;

- Insertion of a new row demands that LastName and Budget are given valid attribute values from the data type specified. NULL values are not allowed.

■ PRIMARY KEY(A_1, A_2, \dots, A_n)

- For each row the PRIMARY KEY will be unique: no two rows have the same values for the primary key.
- Primary keys are automatically required to be NOT NULL.

Foreign Key Referential Integrity Constraints

Let R be a table, and K a subset of its attributes.

- K can be specified to be a **foreign key** referencing another table R' , if K is a primary key of R' .
- This implies a **referential integrity constraint on the allowed instances r of R and r' of R' :** for any row in r , there must exist a row in r' having the same values for K , unless an attribute of K is NULL.

Example:

- Attribute **DeptName** in **Instructor** is declared to be a **foreign key** referencing the **Department** table:

```
CREATE TABLE Instructor
(InstID      VARCHAR(5),
InstName     VARCHAR(20) NOT NULL,
DeptName     VARCHAR(20),
Salary       DECIMAL(8,2),
PRIMARY KEY (InstID),
FOREIGN KEY(DeptName) REFERENCES
Department(DeptName));
```

- This implies a **referential integrity constraint on the allowed instances of Instructor and Department:** for any row in **Instructor**, **DeptName** must be NULL, or there must exist a row in **Department** having the same values for **DeptName**.

Instructor			
InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

Department		
DeptName	Building	Budget
Biology	Watson	90000.00
Comp. Sci.	Taylor	100000.00
Elec. Eng.	Taylor	85000.00
Finance	Painter	120000.00
History	Painter	50000.00
Music	Packard	80000.00
Physics	Watson	70000.00

Foreign Key Referential Integrity Violations

Instructor

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

- SQL *rejects operations* that would lead to a **violation** of the specified referential integrity constraints e.g.:

- insertions to or updates in the referencing table R (Instructor), e.g.

```
INSERT INTO Instructor VALUES  
(99999, 'Anne', 'Mathematics', 90000.00);
```

- deletion or updates of rows in the referenced table R' (Department), (unless referential actions are specified, see next page), e.g.

```
DELETE FROM Department  
WHERE DeptName = 'Physics';
```

DeptName	Building	Budget
Biology	Watson	90000.00
Comp. Sci.	Taylor	100000.00
Elec. Eng.	Taylor	85000.00
Finance	Painter	120000.00
History	Painter	50000.00
Music	Packard	80000.00
Physics	Watson	70000.00

Foreign Keys: Specification of Referential Actions

In the **CREATE TABLE** command for **R** one can write

- **ON DELETE** *referential-actions*
- **ON UPDATE** *referential-actions*

where *referential-actions* specify modifications to make in **R**,

if a DELETE/UPDATE in **R'** leads to a violation, where rows in **R** have **K** values not existing in **R'**:

- **ON DELETE SET NULL** : set the **K** attributes that do not exist in **R'** to **NULL** in **R**.
- **ON DELETE CASCADE** : delete the problematic rows of **R**.
- **ON UPDATE CASCADE**: make the same updates of **K** values in **R**.

■ Example:

CREATE TABLE Instructor

(InstID VARCHAR(5),
InstName VARCHAR(20) **NOT NULL**,
DeptName VARCHAR(20),
Salary DECIMAL(8,2),
PRIMARY KEY (InstID),
FOREIGN KEY(DeptName) **REFERENCES** Department(DeptName)

ON DELETE SET NULL
ON UPDATE CASCADE);

Referential Actions - Examples

Instructor

- **Example:**

```
CREATE TABLE Instructor
(InstID      VARCHAR(5),
InstName     VARCHAR(20) NOT NULL,
DeptName     VARCHAR(20),
Salary       DECIMAL(8,2),
PRIMARY KEY (InstID),
FOREIGN KEY(DeptName)
  REFERENCES Department(DeptName)
  ON DELETE SET NULL
  ON UPDATE CASCADE);
```

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

- Effect of **ON DELETE SET NULL** on
DELETE FROM Department
WHERE DeptName = 'Physics';
- Effect of **ON UPDATE CASCADE** on
UPDATE Department **SET** DeptName = 'BioSc'
WHERE DeptName = 'Biology';

Department		
DeptName	Building	Budget
Biology	BioSc	Watson
Comp. Sci.	Taylor	100000.00
Elec. Eng.	Taylor	85000.00
Finance	Painter	120000.00
History	Painter	50000.00
Music	Packard	80000.00
Physics	Watson	70000.00

Referential Actions - Examples

Instructor

Example:

CREATE TABLE Instructor

```
(InstID      VARCHAR(5),
InstName    VARCHAR(20) NOT NULL,
DeptName    VARCHAR(20),
Salary      DECIMAL(8,2),
PRIMARY KEY (InstID),
FOREIGN KEY(DeptName)
REFERENCES Department(DeptName)
ON DELETE CASCADE);
```

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

- Effect of **ON DELETE CASCADE** on
DELETE FROM Department
WHERE DeptName = 'Physics';

DeptName	Building	Budget
Biology	Watson	90000.00
Comp. Sci.	Taylor	100000.00
Elec. Eng.	Taylor	85000.00
Finance	Painter	120000.00
History	Painter	50000.00
Music	Packard	80000.00
Physics	Watson	70000.00

Views

- **Restricting information access**
 - In some cases, it is not desirable for all users to be able see all the rows and all columns stored in a database table!
- **Views**
 - Provide a *mechanism to hide certain data* from the view of certain users. Specific columns can be hidden as well as specific rows.
 - A view is a *virtual table*.
 - A view *can be used as a table*,
but it is a stored SQL expression, that is executed each time it is used!

Creating and Using Views

- Command to define a view with name *id*:

CREATE VIEW *id* AS *query-expression*

- *id* can be used in queries as if it had been a table name
- *id* is a short for writing *query-expression*
- A 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 Example that Restricts Access

CREATE VIEW Faculty AS

```
SELECT InstID, InstName, DeptName FROM Instructor  
WHERE DeptName NOT IN ('Finance', 'Music');
```

SELECT * **FROM Instructor;**

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

SELECT * **FROM Faculty;**

InstID	InstName	DeptName
10101	Srinivasan	Comp. Sci.
22222	Einstein	Physics
32343	El Said	History
33456	Gold	Physics
45565	Katz	Comp. Sci.
58583	Califieri	History
76766	Crick	Biology
83821	Brandt	Comp. Sci.
98345	Kim	Elec. Eng.

The above SQL expression is executed as:

SELECT * **FROM**

```
SELECT InstID, InstName, DeptName FROM Instructor  
WHERE DeptName NOT IN ('Finance', 'Music');
```

- Note: In the Faculty View the Salary attribute is excluded as well as the rows with instructors working in Finance and Music.

VIEW Example with Calculations

```
CREATE VIEW DepartmentSalary AS
SELECT DeptName, SUM(Salary) AS DeptSalary
FROM Instructor GROUP BY DeptName;
```

```
SELECT * FROM DepartmentSalary;
```

DeptName	DeptSalary
Biology	72000.00
Comp. Sci.	232000.00
Elec. Eng.	80000.00
Finance	170000.00
History	122000.00
Music	40000.00
Physics	182000.00

- Note: Users can see only the sum of salaries per department, and not the salaries of individual instructors (i.e. assuming no user access to Instructor).

VIEW Example with Joined Tables

CREATE VIEW Studentview **AS**

```
SELECT StudID, StudName, TIMESTAMPDIFF(YEAR, Birth, CURRENT_DATE()) AS
Age, TotCredits, InstID
FROM Student NATURAL LEFT OUTER JOIN Advisor;
```

SELECT * FROM Studentview;

StudID	StudName	Age	TotCredits	InstID
00128	Zhang	23	102	45565
12345	Shankar	19	32	10101
19991	Brandt	22	80	NULL
23121	Chavez	23	110	76543
44553	Peltier	19	56	22222
45678	Levy	19	46	22222
54321	Williams	20	54	NULL
55739	Sanchez	20	38	NULL
70557	Snow	19	0	NULL
76543	Brown	21	58	45565
76653	Aoi	21	60	98345
98765	Bourikas	22	98	98345
98988	Tanaka	23	120	76766

SELECT MIN(Age), MAX(Age), AVG(Age)
FROM Studentview;

MIN(Age)	MAX(Age)	AVG(Age)
19	23	20.8462

- Note: Age can change every day by someone having a birthday, and the users would prefer to have the advisor InstID readily available. Views used as Tables!

VIEW Updates

- Add a new row to the Faculty view

```
INSERT Faculty VALUES ('30765', 'Green', 'Physics');
```

- This insertion will be executed by the DBMS as in the statement:

```
INSERT Instructor VALUES ('30765', 'Green', 'Physics', NULL);
```
- Some view updates can be ambiguous and some can be impossible.

- Most SQL systems only allow updates of simple views satisfying conditions like:

- Any attribute not listed in the view's SELECT clause can be set to NULL.
- The FROM clause has only one table, no JOIN operations!
- The SELECT clause contains only attribute names of the base table, and does not have any expressions, aggregates or DISTINCT specifications.
- The query does not have a GROUP BY or HAVING clause.

Authorization

- Concepts used to control the access to database objects:
 - users
 - roles
 - privileges

Creating and Dropping Users

■ CREATE USER with a password

- As Database Administrator (i.e. when connected/logged in as root) you can create a user *userid* with a password *password* by issuing the command:

CREATE USER *userid* IDENTIFIED BY *password*;

- Creation of users Thomas, Peter and Bill:

CREATE USER 'Thomas'@'localhost' IDENTIFIED BY '1984';

CREATE USER 'Peter'@'localhost' IDENTIFIED BY '1979';

CREATE USER 'Bill'@'localhost' IDENTIFIED BY '1980';

- Thomas, Peter and Bill can now connect/login to the database server, but they cannot access any databases or tables before privileges are granted

SHOW GRANTS FOR 'Thomas'@'localhost';

Grants for Thomas@localhost
GRANT USAGE ON *.* TO 'Thomas'@'localhost'

■ RENAME USER

- **RENAME USER 'Bill'@'localhost' TO 'William'@'localhost';**

■ DROP USER

- **DROP USER 'William'@'localhost';**

■ SHOW created users (in MySQL and MariaDB):

- **SELECT user FROM mysql.user;**

user
root
Thomas
Peter

Changing Password

- If DBA is connected as root:

SET PASSWORD FOR 'Thomas'@'localhost' = Password ('Sec4525');

- If connected as Thomas:

SET PASSWORD = Password ('Sec4525');

GRANT and REVOKE Privileges

Syntax

The syntax for granting privileges on a table is:

```
GRANT privileges ON object TO user;
```

```
REVOKE privileges ON object FROM user;
```

privileges

It can be any of the following values:

Privilege	Description
SELECT	Ability to perform SELECT statements on the table.
INSERT	Ability to perform INSERT statements on the table.
UPDATE	Ability to perform UPDATE statements on the table.
DELETE	Ability to perform DELETE statements on the table.
INDEX	Ability to create an index on an existing table.
CREATE	Ability to perform CREATE TABLE statements.
ALTER	Ability to perform ALTER TABLE statements to change the table definition.
DROP	Ability to perform DROP TABLE statements.
GRANT OPTION	Allows you to grant the privileges that you possess to other users.
ALL	Grants all permissions except GRANT OPTION.

GRANT Privileges - Examples

- When users are created they can be granted privileges

GRANT SELECT (InstID, InstName, DeptName) ON University.Instructor TO 'Thomas'@'localhost';

Note: Thomas can only read from the named database, table and attributes!

GRANT SELECT ON University.* TO 'Thomas'@'localhost';

Note: Thomas can read all tables and attributes from the University Database!

GRANT SELECT ON *.* TO 'Thomas'@'localhost';

Note: Thomas can read all databases, tables and attributes on the localhost!

GRANT ALL ON University.* TO 'Thomas'@'localhost';

SHOW GRANTS FOR 'Thomas'@'localhost';

```
Grants for Thomas@localhost
GRANT USAGE ON *.* TO 'Thomas'@'localhost'
GRANT ALL PRIVILEGES ON `university`.* TO 'Thomas'@'localhost'
```

Note: Thomas can do whatever he likes to the University database, since he has all privileges!

The creator of an object holds all privileges on that object, incl. the privileges to grant privileges to other users.

If **WITH GRANT OPTION** is appended at the end of a **GRANT** command, the user will be allowed to grant the same privileges to other users.

Roles

- One can create *roles*, e.g:
 - **CREATE ROLE** TeachingAssistant;
- Roles can (just like users) be granted privileges, e.g:
 - **GRANT SELECT ON** University.Takes **TO** TeachingAssistant;
- Roles can be granted to users (as well as to other roles), e.g:
 - **GRANT** TeachingAssistant **TO** 'Thomas'@'localhost';
by this Thomas is granted the privileges granted to TeachingAssistant
- Roles can be dropped
 - **DROP ROLE** TeachingAssistant;
- NOTE: these commands are not supported by all DBMS (e.g. not for some MySQL versions), but are supported by MariaDB.



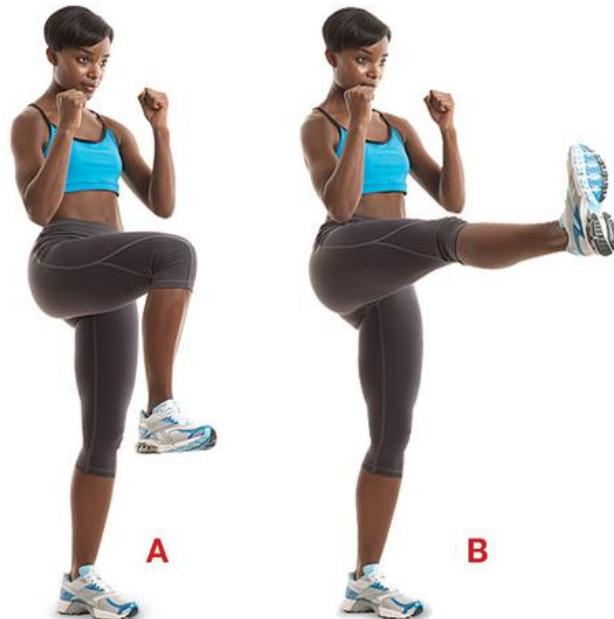
Summary

- Joins based on Cartesian product variations.
- Integrity Constraints to ensure a consistent data model.
- Views of tables to hide and protect data.
- Authorization by creating users & roles and granting & revoking privileges.

Readings

- In Database Systems Concepts please read Chapter 4.
- Pay special attention to the Summaries

Demo Exercises



Demo Exercises clarify ideas and concepts from the Lecture to provide you with good Database Skills.

Do the Demo Exercises. Solutions can be found on later slides.

JOINS

4.1.1 NATURAL JOIN

Use the University Database and display a list of all students with ID, name, department and building. The list should be ordered by the students IDs.

Note:

Student(StudID, StudName, Birth, DeptName,
TotCredits)

Department(DeptName, Building, Budget)

JOINS

4.1.2 NATURAL LEFT OUTER JOIN

Display a list of all instructors, showing their ID, name and the number of sections that they have taught. Argue for the choice of join type.

Note:

Instructor(InstID, InstName, DeptName, Salary)
Teaches(InstID, CourseID, SectionID, Semester,
StudyYear)

4.1.3 Nested SELECT alternative

Write the same query as in the Exercise above, but use a scalar subquery (i.e. a SELECT statement that returns a single value for a number of rows) within another SELECT statement.

VIEWs

4.1.4 Create a VIEW

Called

SeniorStudents(StudID, StudName)
of students with a TotCredits > 100.

Note:

Student(StudID, StudName, Birth, DeptName,
TotCredits)

4.1.5 Create a VIEW

Called

Creditview(StudyYear, SumCredits)
giving the total number of credits taken by
students in each year.

Note:

Takes(StudID, CourseID, SectionID, Semester,
StudyYear, Grade)
Course(CourseID, Title, DeptName, Credits)

Solutions to Demo Exercises



JOINS

4.1.1 NATURAL JOIN

Use the University Database and display a list of all students with ID, name, department and building. The list should be ordered by the students IDs.

Note:

`Student(StudID, StudName, Birth, DeptName, TotCredits)`

`Department(DeptName, Building, Budget)`

4.1.1 NATURAL JOIN

```
SELECT StudID, StudName, DeptName, Building
FROM Student NATURAL JOIN Department
ORDER BY StudID;
```

StudID	StudName	DeptName	Building
00128	Zhang	Comp. Sci.	Taylor
12345	Shankar	Comp. Sci.	Taylor
19991	Brandt	History	Painter
23121	Chavez	Finance	Painter
44553	Peltier	Physics	Watson
45678	Levy	Physics	Watson
54321	Williams	Comp. Sci.	Taylor
55739	Sanchez	Music	Packard
70557	Snow	Physics	Watson
76543	Brown	Comp. Sci.	Taylor
76653	Aoi	Elec. Eng.	Taylor
98765	Bourikas	Elec. Eng.	Taylor
98988	Tanaka	Biology	Watson

JOINS

4.1.2 NATURAL LEFT OUTER JOIN

Display a list of all instructors, showing their ID, name and the number of sections that they have taught. Argue for the choice of join type.

Note:

Instructor(InstID, InstName, DeptName, Salary)
Teaches(InstID, CourseID, SectionID, Semester,
StudyYear)

4.1.2 NATURAL LEFT OUTER JOIN

```
SELECT InstID, InstName, COUNT(SectionID) AS  
SectionsTought  
FROM Instructor NATURAL LEFT OUTER JOIN Teaches  
GROUP BY InstID, InstName;
```

InstID	InstName	SectionsTought
10101	Srinivasan	3
12121	Wu	1
15151	Mozart	1
22222	Einstein	1
32343	El Said	1
33456	Gold	0
45565	Katz	2
58583	Califieri	0
76543	Singh	0
76766	Crick	2
83821	Brandt	3
98345	Kim	1

LEFT OUTER is needed, otherwise instructors who do not teach would not appear in the result (with a 0).

4.1.3 Nested SELECT alternative

Write the same query as in the Exercise above, but use a scalar subquery (i.e. a SELECT statement that returns a single value for a number of rows) within another SELECT statement.

4.1.3 Nested SELECT alternative

```
SELECT InstID, InstName,  
(SELECT COUNT(*) FROM Teaches WHERE Teaches.InstID  
= Instructor.InstID) AS SectionsTought FROM Instructor;
```

VIEWS

4.1.4 Create a VIEW

Called

SeniorStudents(StudID, StudName)
of students with a TotCredits > 100.

Note:

Student(StudID, StudName, Birth, DeptName,
TotCredits)

4.1.5 Create a VIEW

Called

Creditview(StudyYear, SumCredits)
giving the total number of credits taken by
students in each year.

Note:

Takes(StudID, CourseID, SectionID, Semester,
StudyYear, Grade)
Course(CourseID, Title, DeptName, Credits)

4.1.4 Create a VIEW

```
CREATE VIEW SeniorStudents AS
SELECT StudID, StudName FROM Student
WHERE TotCredits > 100;
```

```
SELECT * FROM SeniorStudents;
```

StudID	StudName
00128	Zhang
23121	Chavez
98988	Tanaka

4.1.5 Create a VIEW

```
CREATE VIEW Creditview(StudyYear, SumCredits) AS
(SELECT StudyYear, SUM(Credits) FROM Takes
NATURAL JOIN Course GROUP BY StudyYear);
```

```
SELECT * FROM Creditview;
```

StudyYear	SumCredits
2009	49
2010	29

Exercises



Please answer all exercises
to demonstrate your
Database Skills.

MySQL Exercises
Solutions are available at 11:45

Join, Constraints, View, and Authorization

4.2.1 JOINS

Display the list of all departments, with the total number of instructors in each department.

Note:

Department(DeptName, Building, Budget)
Instructor(InstID, InstName, DeptName, Salary)

4.2.2 JOINS

Display the list of active students, along with titles of the courses they take. The list should be sorted by the student names. (A student is active means: is taking a course.)

4.2.3 Referential Actions

Consider the CREATE TABLE commands for the university database in the appendix. Discuss why the referential ON DELETE actions have been chosen as they are.

Consider the database instance in the appendix. Which table(s) are changed by the following command:

DELETE FROM COURSE WHERE CourseID = 'BIO-301';

Hint: First answer this without using MySQL.

Afterwards, you can check your answer by executing the command.

4.2.4 CREATE a View

Called

SeniorInstructors(InstID, InstName, DeptName)
of instructors with a salary > 80000

Note:

Instructor(InstID, InstName, DeptName, Salary)

4.2.5 Authorization

a) Connect to the database as Database Administrator (root) and create the users **Karen**, **Linda** and **Susan** with the generic password **SetPassword**. Please observe that user names are case sensitive!

b) Then grant SELECT to **Karen** and ALL to **Linda** and **Susan** to a database under your DBA control.

c) Then close your connection to the server (on Workbench under Windows this can be done choosing File->Close Connection Tab).

d) Then add a connection for **Karen**. From the welcome page of Workbench this is done as follows:

1. Click on the + icon and a window will pop up.
2. In that window choose a name (e.g. **KarenConnection**) for the connection, set the username to **Karen**, check port is 3306, and click **Ok**. Then you will be returned to the welcome page where you can see the new connection **KarenConnection**.

e) Connect as **Karen**:

1. Click on the icon with the new connection name (**KarenConnection**) and a window will pop up.
2. Fill out the password chosen for **Karen** in step a): **SetPassword**. It will then probably give a connection warning and you should just click on **Continue Anyway**.

f) Change Karen's password to **KarenSecret**

g) Try (still as **Karen**) to execute some select statements and some table modifications and see what happens.

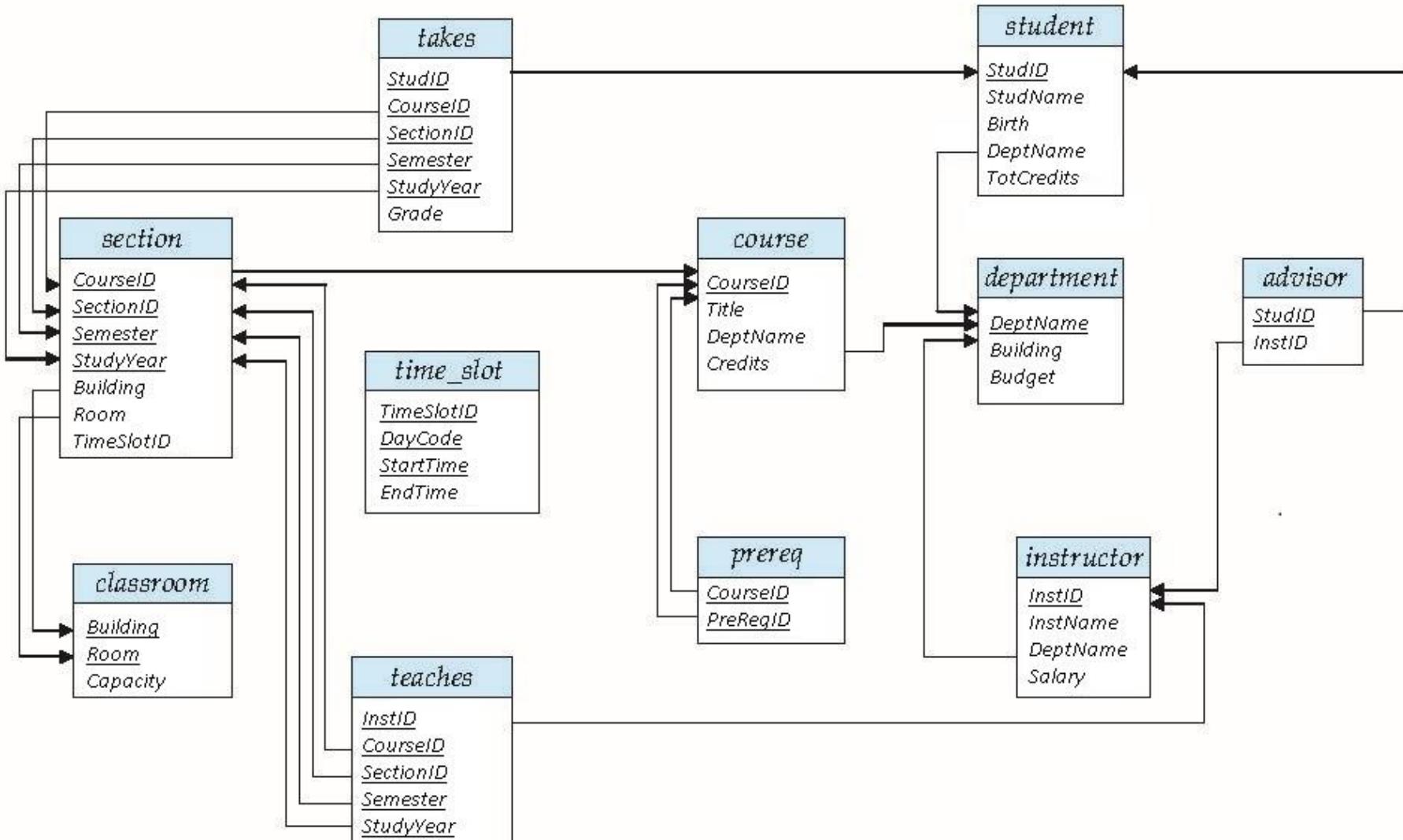
h) Close the connection and then connect as DBA (root) and drop users **Karen**, **Linda** and **Susan**.

i) Close the connection. Delete **KarenConnection** by clicking on the tools icon on the Workbench welcome page, then clicking on **KarenConnection**, and finally clicking on **delete** and then **close**.

University Database, revisited

- Database example used throughout the textbook and in this course!
 1. Database Schema Diagram
 2. Database Schema (CREATE TABLE Commands)
 3. Database Instance

Database Schema Diagram

 from the book, but with new names

University Database Schema (1 of 4)

```
CREATE TABLE Instructor
  (InstID          VARCHAR(5),
   InstName        VARCHAR(20) NOT NULL,
   DeptName        VARCHAR(20),
   Salary          DECIMAL(8,2),
   PRIMARY KEY (InstID),
   FOREIGN KEY(DeptName) REFERENCES Department(DeptName) ON DELETE SET NULL
  );

CREATE TABLE Student
  (StudID          VARCHAR(5),
   StudName        VARCHAR(20) NOT NULL,
   Birth           DATE,
   DeptName        VARCHAR(20),
   TotCredits      DECIMAL(3,0),
   PRIMARY KEY(StudID),
   FOREIGN KEY(DeptName) REFERENCES Department(DeptName) ON DELETE SET NULL
  );

CREATE TABLE Advisor
  (StudID          VARCHAR(5),
   InstID          VARCHAR(5),
   PRIMARY KEY(StudID),
   FOREIGN KEY(InstID) REFERENCES Instructor(InstID) ON DELETE SET NULL,
   FOREIGN KEY(StudID) REFERENCES Student(StudID) ON DELETE CASCADE
  );

CREATE TABLE Department
  (DeptName        VARCHAR(20),
   Building         VARCHAR(15),
   Budget           DECIMAL(12,2),
   PRIMARY KEY(DeptName)
  );
```

University Database Schema (2 of 4)

```
CREATE TABLE Course
  (CourseID          VARCHAR(8),
   Title             VARCHAR(50),
   DeptName          VARCHAR(20),
   Credits           DECIMAL(2,0),
   PRIMARY KEY(CourseID),
   FOREIGN KEY(DeptName) REFERENCES Department(DeptName) ON DELETE SET NULL
  );

CREATE TABLE PreReq
  (CourseID          VARCHAR(8),
   PreReqID          VARCHAR(8),
   PRIMARY KEY(CourseID, PreReqID),
   FOREIGN KEY(CourseID) REFERENCES Course(CourseID) ON DELETE CASCADE,
   FOREIGN KEY(PreReqID) REFERENCES Course(CourseID)
  );
```

University Database Schema (3 of 4)

```
CREATE TABLE Teaches
  (InstID          VARCHAR(5),
   CourseID        VARCHAR(8),
   SectionID       VARCHAR(8),
   Semester        ENUM('Fall','Winter','Spring','Summer'),
   StudyYear       YEAR,
   PRIMARY KEY(InstID, CourseID, SectionID, Semester, StudyYear),
   FOREIGN KEY(CourseID, SectionID, Semester, StudyYear)
     REFERENCES Section(CourseID, SectionID, Semester, StudyYear) ON DELETE CASCADE,
   FOREIGN KEY(InstID) REFERENCES Instructor(InstID) ON DELETE CASCADE
);

CREATE TABLE Takes
  (StudID          VARCHAR(5),
   CourseID        VARCHAR(8),
   SectionID       VARCHAR(8),
   Semester        ENUM('Fall','Winter','Spring','Summer'),
   StudyYear       YEAR,
   Grade           VARCHAR(2),
   PRIMARY KEY(StudID, CourseID, SectionID, Semester, StudyYear),
   FOREIGN KEY(CourseID, SectionID, Semester, StudyYear)
     REFERENCES Section(CourseID, SectionID, Semester, StudyYear) ON DELETE CASCADE,
   FOREIGN KEY(StudID) REFERENCES Student(StudID) ON DELETE CASCADE
);
```

University Database Schema (4 of 4)

```
CREATE TABLE Section
  (CourseID          VARCHAR(8),
  SectionID          VARCHAR(8),
  Semester           ENUM('Fall','Winter','Spring','Summer'),
  StudyYear          YEAR,
  Building           VARCHAR(15),
  Room                VARCHAR(7),
  TimeSlotID         VARCHAR(4),
  PRIMARY KEY(CourseID, SectionID, Semester, StudyYear),
  FOREIGN KEY(CourseID) REFERENCES Course(CourseID) ON DELETE CASCADE,
  FOREIGN KEY(Building, Room) REFERENCES Classroom(Building, Room) ON DELETE SET NULL
);

CREATE TABLE TimeSlot
  (TimeSlotID    VARCHAR(4),
  DayCode        ENUM('M','T','W','R','F','S','U'),
  StartTime      TIME,
  EndTime        TIME,
  PRIMARY KEY(TimeSlotID, DayCode, StartTime)
);

CREATE TABLE Classroom
  (Building          VARCHAR(15),
  Room                VARCHAR(7),
  Capacity           DECIMAL(4,0),
  PRIMARY KEY(Building, Room)
);
```

Database Instance (1 of 4)

Instructor

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

Student

StudID	StudName	Birth	DeptName	TotCredits
00128	Zhang	1992-04-18	Comp. Sci.	102
12345	Shankar	1995-12-06	Comp. Sci.	32
19991	Brandt	1993-05-24	History	80
23121	Chavez	1992-04-18	Finance	110
44553	Peltier	1995-10-18	Physics	56
45678	Levy	1995-08-01	Physics	46
54321	Williams	1995-02-28	Comp. Sci.	54
55739	Sanchez	1995-06-04	Music	38
70557	Snow	1995-11-22	Physics	0
76543	Brown	1994-03-05	Comp. Sci.	58
76653	Aoi	1993-09-18	Elec. Eng.	60
98765	Bourikas	1992-09-23	Elec. Eng.	98
98988	Tanaka	1992-06-02	Biology	120

Advisor

StudID	InstID
12345	10101
44553	22222
45678	22222
00128	45565
76543	45565
23121	76543
98988	76766
76653	98345
98765	98345

Department

DeptName	Building	Budget
Biology	Watson	90000.00
Comp. Sci.	Taylor	100000.00
Elec. Eng.	Taylor	85000.00
Finance	Painter	120000.00
History	Painter	50000.00
Music	Packard	80000.00
Physics	Watson	70000.00

Database Instance (2 of 4)

Teaches

InstID	CourseID	SectionID	Semester	StudyYear
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
10101	CS-101	1	Fall	2009
45565	CS-101	1	Spring	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
10101	CS-315	1	Spring	2010
45565	CS-319	1	Spring	2010
83821	CS-319	2	Spring	2010
10101	CS-347	1	Fall	2009
98345	EE-181	1	Spring	2009
12121	FIN-201	1	Spring	2010
32343	HIS-351	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

Takes

StudID	CourseID	SectionID	Semester	StudyYear	Grade
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	NULL

Database Instance (3 of 4)

Course

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

PreReq

CourseID	PreReqID
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Database Instance (4 of 4)

Section

CourseID	SectionID	Semester	StudyYear	Building	Room	TimeSlotID
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Classroom

Building	Room	Capacity
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

TimeSlot

TimeSlotID	DayCode	StartTime	EndTime
A	M	08:00:00	08:50:00
A	W	08:00:00	08:50:00
A	F	08:00:00	08:50:00
B	M	09:00:00	09:50:00
B	W	09:00:00	09:50:00
B	F	09:00:00	09:50:00
C	M	11:00:00	11:50:00
C	W	11:00:00	11:50:00
C	F	11:00:00	11:50:00
D	M	13:00:00	13:50:00
D	W	13:00:00	13:50:00
D	F	13:00:00	13:50:00
E	T	10:30:00	11:45:00
E	R	10:30:00	11:45:00
F	T	14:30:00	15:45:00
F	R	14:30:00	15:45:00
G	M	16:00:00	16:50:00
G	W	16:00:00	16:50:00
G	F	16:00:00	16:50:00
H	W	10:00:00	12:30:00



Technical University of Denmark

DTU Compute

Department of Applied Mathematics and Computer Science

Advanced SQL Programming

Sections 4.3, 5.2, 5.3 in the textbook

©Anne Haxthausen and Flemming Schmidt

These slides have been prepared by Anne Haxthausen, partly reusing/modifying slides by Flemming Schmidt. Some examples come from Silberschatz, Korth, Sudarshan, 2010.

Contents

- Procedural Statements
- Programming Objects
 - Functions
 - Procedures
 - Triggers
 - Transactions
 - Events
- Errors and Warnings
 - Error Handling
- SQL Access from a Programming Language
 - ODBC, JDBC and Embedded SQL
- Demo Exercises and Exercises
- Appendix: University Database



SQL Statements

- **Declarative statements, e.g.:**
 - DDL statements like CREATE and DROP
 - DML statements like INSERT, DELETE and SELECT
- **Procedural/imperative statements:**
 - Variable assignments
 - If-then-else, case
 - Various kinds of loops
 - Blocks (BEGIN ... END)
 - Function and procedure calls
 - Return statement (inside a function body)
 - ...

Note: Most DBMS implement **non standard syntax** for procedural statements, but the concepts are the same! These slides use the syntax implemented by MySQL/MariaDB (while the text book uses the standard syntax).

Procedural Statements

■ BEGIN-END blocks

[label_name:] BEGIN

local-variable-declarations #scope is within the begin-end block
statements

END [label_name];

- *local-variable-declarations* are of the form:

DECLARE var_name type [DEFAULT value];

■ Variable assignments

SET var_name = expr;

- where *var_name* can be the name of

- a *local variable* declared in an enclosed BEGIN-END statement
- a formal *parameter* of an enclosed procedure (or function)
- a *user-defined variable* (name is of the form @id, no specific declaration, is session specific),
e.g. **SET @x = 1;** **SET @x = @x +1;**

SET [GLOBAL] system_var_name = value;

■ Variable assignments in SELECT clauses

SELECT ..., expr INTO var_name,

Example: **SELECT SUM(Salary) INTO @var FROM Instructor; SELECT @var;**

@var
999000.00

Procedural Statements

- **IF ... THEN ... ELSEIF ... THEN ... ELSE ... END IF;**

IF *condition₁* **THEN** *statements₁*

[ELSEIF *condition₂* **THEN** *statements₂* **]**

...

[ELSE *statements_n* **]**

END IF;

Procedural Statements

- CASE using an expression or conditions

CASE *expression*

WHEN *value₁* **THEN**

statements₁ #to execute when expression equals value₁

...

WHEN *value_n* **THEN**

statements_n #to execute when expression equals value_n

[ELSE

statements #to execute when no values matched]

END CASE;

CASE

WHEN *condition₁* **THEN**

statements₁ #to execute when condition₁ is TRUE

...

WHEN *condition_n* **THEN**

statements_n #to execute when condition_n is TRUE

[ELSE

statements #to execute when all conditions were FALSE]

END CASE;

Procedural Statements

- **LOOP:**

```
[ label_name: ] LOOP  
  statements          #can be terminated by a LEAVE or RETURN statement  
END LOOP [ label_name ];
```

- **WHILE DO:**

```
[ label_name: ] WHILE condition DO  
  statements  
END WHILE [ label_name ];
```

- **REPEAT UNTIL:**

```
[ label_name: ] REPEAT  
  statements  
UNTIL condition  
END REPEAT [ label_name ];
```

- **LEAVE:** **LEAVE** *label_name* # to exit a labelled loop

- **ITERATE:** **ITERATE** *label_name* # to start a labelled loop again

Programming Objects

■ A Function

- is a stored program for which parameters can be passed in, and then it will return a value:

```
CREATE FUNCTION function_name (parameter1 datatype1,...,parametern datatypen)  
RETURNS return_datatype  
routine_body
```

#Must include a 'RETURN value' statement

■ A Procedure

- is a stored program for which parameters can be passed in and out:

```
CREATE PROCEDURE procedure_name ([IN | OUT | INOUT] parameter1 datatype1 ,  
...)  
routine_body
```

■ A Trigger

- is a stored program that is executed BEFORE/AFTER an INSERT, UPDATE or DELETE operation:

```
CREATE TRIGGER trigger_name      #Use naming standard like: Table_name_Before_Insert  
{ BEFORE | AFTER } { INSERT | UPDATE | DELETE }  
ON table_name FOR EACH ROW  
actions
```

Programming Objects

▪ A Transaction

- Is a group of SQL statements that executes as one entity. If one or more statements fails to execute successfully, then the successfully executed statements are undone, and the database is returned to the initial state before the transaction was started. In practice, all changes are stored in temporary storage, until they are either committed and stored permanently, or discarded (i.e. rolled back) and deleted from temporary storage.
- Transactions can be executed concurrently by the DBMS without disturbing each other.

▪ An Event

- Is a stored program that is executed at a given time or at given time intervals:
CREATE EVENT event_name
ON SCHEDULE schedule
DO statement

▪ SQL Access from a Programming Language

- Normally the application user interface and application logic are programmed using a common programming language like C, Java or Cobol.
- Application Programs written in a programming language can interact with a database using an Application Program Interface (API) for example ODBC and JDBC.
- The SQL standard also defines embedding's of SQL in a variety of programming languages such as C, Java, and Cobol, for example like inserting in the programming language code:
EXEC SQL <embedded SQL statement > END_EXEC

Functions

- **Create a function**

CREATE FUNCTION *function_name* (*parameter₁* *datatype₁*, ..., *parameter_n* *datatype_n*)
 RETURNS *return_datatype*

routine_body #must include a 'RETURN value' statement

- A *routine_body* is a *statement* (can e.g. be a block).
- Must not be recursive.

- **Call a function**

function_name(*e₁*, ..., *e_n*)

- *e₁*, ..., *e_n* are expressions matching the formal parameters.
- Is syntactically a *value expression*.

- **Drop a function**

DROP FUNCTION *function_name*;

- Note, if binary logging is enabled in your MySQL/MariaDB server (you can check this with the command **SHOW VARIABLES LIKE 'log_bin'**; which replies 'ON' if it is enabled) you may need to execute **SET GLOBAL log_bin_trust_function_creators = 1**; in order to be able use functions. Otherwise (when the reply is 'OFF'), it should not be needed.

Functions – Example

- Create a function calculating Age from Date of Birth

```
CREATE FUNCTION Age (vDate DATE) RETURNS INTEGER  
RETURN TIMESTAMPDIFF(YEAR, vDate, CURDATE());
```

- Test the function

```
SELECT Age(20000720);
```

Age(20000720)
20

```
SELECT StudID, StudName, Birth, Age(Birth) FROM Student;
```

StudID	StudName	Birth	Age(Birth)
00128	Zhang	1992-04-18	23
12345	Shankar	1995-12-06	20
19991	Brandt	1993-05-24	22
23121	Chavez	1992-04-18	23
44553	Peltier	1995-10-18	20
45678	Levy	1995-08-01	20
54321	Williams	1995-02-28	20

Functions - Example

■ Create a function and test it

- Given a department name, the function returns the number of instructors

```
DELIMITER //
```

```
CREATE FUNCTION DeptInstCount (vDeptName VARCHAR(20)) RETURNS INT
```

```
BEGIN
```

```
    DECLARE vDeptInstCount INT;
```

```
    SELECT COUNT(*) INTO vDeptInstCount FROM Instructor
```

```
    WHERE DeptName = vDeptName;
```

```
    RETURN vDeptInstCount;
```

```
END//
```

```
DELIMITER ;
```

The SQL DELIMITER is temporarily changed from ";" to "://" in order to allow ";" in the function body!

- SELECT** DeptName, **DeptInstCount(DeptName)** **AS** Instructors **FROM** Department;
- Find department name and budget of all departments with two or more instructors.

```
SELECT DeptName, Budget FROM Department
```

```
WHERE DeptInstCount(DeptName) >= 2;
```

DeptName	Instructors
Biology	1
Comp. Sci.	3
Elec. Eng.	1
Finance	2
History	2
Music	1
Physics	2

DeptName	Budget
Comp. Sci.	1000000.00
Finance	1200000.00
History	50000.00
Physics	70000.00

Procedures

■ Create a procedure

CREATE PROCEDURE *procedure_name*

 (**[IN|OUT|INOUT]** *parameter₁* *datatype₁* , ... , **[IN|OUT|INOUT]** *parameter_n* *datatype_n*)
 routine_body

- A *routine_body* is a *statement* (typically a block).
- An **IN** parameter passes a value into a procedure.
- An **OUT** parameter passes a value from the procedure back to the caller.
- An **INOUT** parameter passes a value into the procedure and back to the caller.
- Note that a procedure can have several output parameters!

■ Call a procedure

CALL *procedure_name*(*e₁*, ... , *e_n*);

- *e₁*, ... , *e_n* are expressions matching the formal parameters wrt types.
- When *parameter_i* is **OUT** and **INOUT** parameters, *e_i* must be a variable name.
- Is syntactically a *statement*.
- **CALL p();** is equivalent to **CALL p;**

■ Drop a procedure

DROP PROCEDURE *procedure_name*;

Procedures – Examples Without Parameters

- Create a simple procedure and test it

```
CREATE PROCEDURE ShowVersion() SELECT VERSION() AS 'My DBMS Version';
CALL ShowVersion;
```

```
My DBMS Version
10.3.12-MariaDB
```

- Create a more elaborate procedure (providing a lucky number, which is 4 or 7)

```
DELIMITER $$  
CREATE PROCEDURE LuckyNumber()  
BEGIN  
    DECLARE vNumber INTEGER DEFAULT 0;  
    IF DAY(CURRENT_DATE())%2 = 0 THEN SET vNumber = 4; ELSE SET vNumber = 7; END IF;  
    SELECT vNumber AS 'Todays Lucky Number';  
END $$  
DELIMITER ;
```

```
CALL LuckyNumber;
```

Todays Lucky Number
4

Procedures – Example with IN Parameters and Side Effects

■ Create a procedure

- to add a new instructor with the minimum wage 29000 as salary:

```
DELIMITER //
```

```
CREATE PROCEDURE AddInstructor
```

```
  (IN vInstID VARCHAR(5), IN vInstName VARCHAR(20), IN vDeptName VARCHAR(20))
```

```
BEGIN
```

```
  INSERT Instructor(InstID, InstName, DeptName, Salary)
```

```
    VALUES (vInstID, vInstName, vDeptName, 29000); #As a side effect, a table is changed
```

```
END //
```

```
DELIMITER ;
```

- This procedure has: 3 inputs, 0 outputs.
- As a side effect it changes the Instructor table.

■ Use a CALL statement to execute the procedure

```
CALL AddInstructor ('99999', 'Schmidt', 'Comp. Sci.');
```

```
SELECT * FROM Instructor;
```

InstID	InstName	DeptName	Salary
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00
99999	Schmidt	Comp. Sci.	29000.00
NULL	NULL	NULL	NULL

Procedures – Example with IN and OUT parameters

■ Create a procedure

- to calculate the maximum capacity for a building:

```
DELIMITER //
```

```
CREATE PROCEDURE BuildingCapacity
(IN vBuilding VARCHAR(20), OUT vMaxCapacity INT)
BEGIN
    SELECT SUM(Capacity) INTO vMaxCapacity
    FROM Classroom
    WHERE Building = vBuilding;
END //
```

```
DELIMITER ;
```

- This procedure has: 1 input variable, 1 output variable, no side effects on tables.

■ Call the procedure

```
CALL BuildingCapacity ('Watson', @MaxCap);
```

After this call the output variable can be read:

- **SELECT @MaxCap;**
- **SELECT * FROM Classroom WHERE Capacity > @MaxCap;**

```
 @MaxCap
  80
```

Classroom		
Building	Room	Capacity
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Building	Room	Capacity
Packard	101	500

Triggers

■ A Trigger

- Is a set of SQL statements that are *executed automatically* by the DBMS system as *a side effect of a table modification* (i.e. an SQL INSERT, UPDATE or DELETE).
- Triggers can be used to *preprocess* and *post process* table changes.
 - Example: In order to *validate values* of a new row, a trigger can be executed BEFORE a row is inserted in a table, and it might reject insertion.
 - Example: In order to update *log tables*, a trigger can be executed AFTER a row has been inserted in a table.
 - Example: In order to *enforce integrity constraints* triggers may process tables changes.

Triggers

- **Create a trigger:**

```
CREATE TRIGGER trigger_name #Use naming standard like: table_name_Before_Insert
{ BEFORE | AFTER }{ INSERT | UPDATE | DELETE } ON table_name
FOR EACH ROW actions # trigger actions
```

- *actions* is a statement.
- **OLD.A** and **NEW.A** can be used in *actions* to refer to attribute A before an (UPDATE or DELETE) event and after an (UPDATE or INSERT) event, respectively.
- In a BEFORE trigger, you can make assignments **SET NEW.A = ...**. This means you can use a trigger to modify the values to be inserted into a new row or used to update a row.

- **The trigger definition specifies**

- the **actions** to be taken for inserted/updated/deleted rows when the trigger executes
 - **IMPORTANT:** Note that the "FOR EACH ROW" refers only to each row inserted/updated/deleted by the original query, not every row of the table! So if you only inserted one row, your trigger will run once in the context of that row.
- the **event** for which a trigger is to be executed: an INSERT, UPDATE or DELETE
- the **time** at which the trigger should be executed: BEFORE or AFTER the event

- **Drop a trigger:**

- **DROP TRIGGER** *table_name.trigger_name*; # *table_name*. only needed if *trigger_name* is defined on several tables.

- **Show triggers in a database named *db*:**

- **SHOW TRIGGERS IN *db*;**

Triggers – Example With Preprocessing

- Create a trigger that executes **before** a row is Inserted in Instructor
 - The trigger statement below is executed before a row is inserted in table Instructor. Statements with INSERT involving the Instructor table will trigger execution of the trigger statement (see next slide).

```
DELIMITER //
CREATE TRIGGER Instructor_Before_Insert
BEFORE INSERT ON Instructor FOR EACH ROW
BEGIN
    IF NEW.Salary < 29000 THEN SET NEW.Salary = 29000;
    ELSEIF NEW.Salary > 150000 THEN SET NEW.Salary = 150000;
    END IF;
END// 
DELIMITER ;
```
- A trigger can also be used to make a log of inserts since last backup. **After** a row is inserted in the table Instructor the same row is inserted in the table InstLog. Logs are then cleared after backup. See demo exercise #2.
- A trigger that raises a signal (exception) for undesired insertions will be shown under the Error Handling part of these slides.

Trigger Test of Instructor_Before_Insert

```
INSERT Instructor VALUES (99997, 'Anne', 'Comp. Sci.', 500000.00); #too high a salary
```

```
INSERT Instructor VALUES (99998, 'Helen', 'Comp. Sci.', 100000.00);
```

```
INSERT Instructor VALUES (99999, 'Peter', 'Comp. Sci.', 20000.00); #too low a salary
```

```
SELECT InstName, Salary FROM Instructor WHERE DeptName = 'Comp. Sci.');
```

InstName	Salary
Srinivasan	65000.00
Katz	75000.00
Brandt	92000.00
Anne	150000.00
Helen	100000.00
Peter	29000.00

Triggers – Example with Postprocessing

Given `Takes(StudID, CourseID, ..., Grade)`, `Student(StudID, ..., TotCredits)`, `Course(CourseID, ..., Credits)`

- Create a trigger that updates `Student` after a row has been updated in `Takes`: it should bring the total credits (`TotCredits`) of a student up-to-date when the student passes a course.

`DELIMITER $$`

`CREATE TRIGGER Takes_After_Update`

`AFTER UPDATE ON Takes FOR EACH ROW`

`IF NEW.Grade <> 'F' AND NEW.Grade IS NOT NULL` # course is passed after
AND

`(OLD.Grade = 'F' or OLD.Grade IS NULL)` # course was not passed before
`THEN UPDATE Student`

`SET TotCredits = TotCredits +`

`(SELECT Credits FROM Course WHERE Course.CourseID = NEW.CourseID)`
`WHERE Student.StudID = NEW.StudID;`

`END IF $$`

`DELIMITER ;`

Test Trigger Takes_After_Update

SELECT * FROM Takes WHERE StudID = 98988;

StudID	CourseID	SectionID	Semester	StudyYear	Grade
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	NULL

SELECT StudID, TotCredits FROM Student WHERE StudID = 98988;

StudID	TotCredits
98988	120

UPDATE Takes SET Grade = 'A' WHERE StudID = 98988 AND CourseID = 'BIO-301';

SELECT * FROM Takes WHERE StudID = 98988;

StudID	CourseID	SectionID	Semester	StudyYear	Grade
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	A

SELECT StudID, TotCredits FROM Student WHERE StudID = 98988;

StudID	TotCredits
98988	124

When Not To Use Triggers!

- Triggers were used earlier for tasks such as:
 - *Maintaining summary data* tables (e.g., total salary of each department).
 - *Replicating databases* by recording changes to special relations (called change or delta relations) and having a separate process that applies the changes over to a replica.
- There are better ways of doing these now:
 - Databases today provide built-in View facilities to maintain summary data.
 - Databases provide built-in support for replication.

Transactions

- **SQL Transaction**
 - Is a group of SQL statements (like SELECT, INSERT, UPDATE and DELETE), that *executes as one entity*.
 - If just one statement *fails* to execute successfully, then the successfully executed statements are *rolled back*, and the database is returned to the initial state before the transaction was started.
 - Transactions *can be executed concurrently* without disturbing each other.
- **Nature of transactions**
 - All transactions have a beginning and an end.
 - Transaction results are either saved (*committed*) or aborted (*rollback*).
 - If a transaction fails, then no part of the transaction is saved.

Transactions – Motivating Example

- Moving amounts from one bank account to another
 - Amounts need to be moved without being lost or moved twice!

SELECT * FROM Accounts;

ID	Owner	Amount
1001	James Morrison	35000.00
1002	Frank Summers	15000.00
1003	Hank Williamson	40000.00

- Transfer 5.000 from Frank (ID 1002) to Hank (ID 1003)

SET @Transfer = 5000;

SET @OldAmountID1 = (SELECT Amount FROM Accounts WHERE ID = 1002);

UPDATE Accounts SET Amount = (@OldAmountID1 - @Transfer) WHERE ID = 1002;

SET @OldAmountID2 = (SELECT Amount FROM Accounts WHERE ID = 1003);

UPDATE Accounts SET Amount = (@OldAmountID2 + @Transfer) WHERE ID = 1003;

SELECT * FROM Accounts;

ID	Owner	Amount
1001	James Morrison	35000.00
1002	Frank Summers	10000.00
1003	Hank Williamson	45000.00

However, the database would loose 5.000 if the second update failed. If ID 1003 did not exist, or if the system crashed, or if another user or program interfered between the updates!

Controlling Transactions

- **START TRANSACTION;**
 - Will ensure that all row inserts, updates and deletes are stored in a Temporary Buffer and NOT written to disk and permanently stored.
- **COMMIT;**
 - Will ensure that the changes stored in the Temporary Buffer are written to disk and permanently stored.
- **ROLLBACK;**
 - Will ensure that all changes made since the START TRANSACTION will be deleted in the Temporary Buffer and NOT written to disk and permanently stored.

Procedure with a Transaction

- Create a Procedure that includes a Transaction

```
DELIMITER //
CREATE PROCEDURE Transfer (
    IN vTransfer DECIMAL(9,2), vID1 INT, vID2 INT, OUT vStatus VARCHAR(45))
BEGIN
    DECLARE OldAmountID1, NewAmountID1, OldAmountID2, NewAmountID2 INT DEFAULT 0;
    START TRANSACTION;
    SET SQL_SAFE_UPDATES = 0;
    SET OldAmountID1 = (SELECT Amount FROM Accounts WHERE ID = vID1);
    SET NewAmountID1 = OldAmountID1 - vTransfer;
    UPDATE Accounts SET Amount = NewAmountID1 WHERE ID = vID1;
    SET OldAmountID2 = (SELECT Amount FROM Accounts WHERE ID = vID2);
    SET NewAmountID2 = OldAmountID2 + vTransfer;
    UPDATE Accounts SET Amount = NewAmountID2 WHERE ID = vID2;
    IF (OldAmountID1 + OldAmountID2) = (NewAmountID1 + NewAmountID2)
        THEN SET vStatus = 'Transaction Transfer committed!'; COMMIT;
        ELSE SET vStatus = 'Transaction Transfer rollback'; ROLLBACK;
    END IF;
END //
DELIMITER ;
```

Testing Procedure with Transaction

SELECT * FROM Accounts;

ID	Owner	Amount
1001	James Morrison	35000.00
1002	Frank Summers	15000.00
1003	Hank Williamson	40000.00

CALL Transfer (5000, 1002, 1003, @TStatus);
SELECT @TStatus;

@TStatus
Transaction Transfer committed!

SELECT * FROM Accounts;

ID	Owner	Amount
1001	James Morrison	35000.00
1002	Frank Summers	10000.00
1003	Hank Williamson	45000.00

CALL Transfer (5000, 1002, 1015, @TStatus);
SELECT @TStatus;

@TStatus
Transaction Transfer rollback!

CALL Transfer (5000, 1012, 1015, @TStatus);
SELECT @TStatus;

@TStatus
Transaction Transfer rollback!

SELECT * FROM Accounts;

ID	Owner	Amount
1001	James Morrison	35000.00
1002	Frank Summers	10000.00
1003	Hank Williamson	45000.00

Events

- **CREATE** an event

```
CREATE EVENT event_name  
ON SCHEDULE schedule  
DO statement
```

- *schedule* can e.g. be of the form

EVERY *n timeunit*

[STARTS *timestamp***]**

[ENDS *timestamp***]**

- *timeunit* can be: YEAR | QUARTER | MONTH | DAY | HOUR | MINUTE | WEEK | SECOND | YEAR_MONTH | DAY_HOUR | DAY_MINUTE | DAY_SECOND | HOUR_MINUTE | HOUR_SECOND | MINUTE_SECOND
- **SET GLOBAL** *event_scheduler* = 1; # in order for MariaDB/MySQL to execute events
- Events can e.g. be used to schedule database period backups at regular time intervals.

Events - Example

- Setting up a database operation that runs periodically

```
SHOW VARIABLES LIKE 'event_scheduler';      # Initially it is set OFF
```

Variable_name	Value
event_scheduler	OFF

```
SET GLOBAL event_scheduler = 1;      # To set it ON. MySQL will look for/execute events
```

```
CREATE TABLE MarkLog(  
    TS      TIMESTAMP,  
    Message  VARCHAR(100));
```

```
CREATE EVENT MarkInsert  
ON SCHEDULE EVERY 2 MINUTE  
DO INSERT MarkLog VALUES (CURRENT_TIMESTAMP, "-- It's time again--");
```

```
SELECT * FROM MarkLog;
```

TS	Message
2015-07-07 21:14:04	-- It's time again--
2015-07-07 21:16:04	-- It's time again--
2015-07-07 21:18:04	-- It's time again--

Error Handling

- MariaDB raises signals and show **Warnings & Error Messages** for **predefined conditions (cases)**.
- **Examples using the MariaDB Command Line Client:**

```
Enter password: ****
```

```
MariaDB[none]> select * from instructor;  
ERROR 1046 (3D000): No database selected
```

```
MariaDB[none]> use University;  
Database changed
```

```
MariaDB[University]> select * from instructor table;  
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual  
that corresponds to your MySQL server version for the right syntax to use  
near 'table' at line 1
```

```
MariaDB[University]> select * from instructors;  
ERROR 1146 (42S02): Table 'university.instructors' doesn't exist
```

```
MariaDB[University]> select * from instructor;  
+-----+-----+-----+-----+  
| InstID | InstName | DeptName | Salary |  
+-----+-----+-----+-----+  
| 10101 | Srinivasan | Comp. Sci. | 65000.00 |  
| 98345 | Kim | Elec. Eng. | 80000.00 |  
...  
12 rows in set (0.02 sec)
```

Error Messages

- Example: ERROR 1146 (42S02): Table 'university.instructors' doesn't exist
 - 1146 is a MySQL/MariaDB error code. The set of MariaDB error codes is a superset of the set of MySQL error codes.
 - Table 'university.instructors' doesn't exist is the associated error text.
 - The error codes and texts are not portable to other DBMS!
 - 42S02 is the SQLSTATE used by ANSI SQL and ODBC. It gives error information:
 - Not all MariaDB error codes are mapped to SQLSTATE error codes.
 - HY000 is used for a general error for unmapped error numbers.
 - 00000 indicates successful completion of an SQL statement.
 - ...
- For info about the codes, see in the MariaDB documentation:
 - <https://mariadb.com/kb/en/library/mariadb-error-codes/>

User-defined Error Signalling

- It is possible to raise signals for user-defined conditions indicating errors and warnings using the SIGNAL statement. Typical pattern for this:

IF *condition* **THEN SIGNAL SQLSTATE** *state*

SET MYSQL_ERRNO = ..., **MESSAGE_TEXT** = ...;

- Example from the body of the Course_Before_Insert trigger on next page:

```
IF NOT (NEW.Credits BETWEEN 0 AND 5)
THEN SIGNAL SQLSTATE 'HY000'
    SET MYSQL_ERRNO = 1525, MESSAGE_TEXT = 'Invalid Credits; Range is [0,5]!';
END IF;
```

- SQLSTATE HY000 means that a general error has occurred
- MYSQL_ERRNO 1525 means wrong value occurred
- When the signal is raised*, the trigger program is aborted and the related INSERT is not executed!

Example: Trigger With Error Handling and Preprocessing

Trigger with rejection and preprocessing

DELIMITER \$\$

```
CREATE TRIGGER Course_Before_Insert
BEFORE INSERT ON Course FOR EACH ROW
BEGIN
```

```
IF NOT (
```

```
  ( LENGTH(NEW.CourseID)=6 AND LEFT(NEW.CourseID,2) IN ('CS','EE','MU'))
  OR
  (LENGTH(NEW.CourseID)=7 AND LEFT(NEW.CourseID,3) IN ('BIO','FIN','HIS','PHY'))
)
```

```
  AND LOCATE('-',NEW.CourseID) <> 0
```

```
  AND RIGHT(NEW.CourseID,3) BETWEEN 100 AND 999 )
```

```
THEN SIGNAL SQLSTATE 'HY000'
```

```
    SET MYSQL_ERRNO = 1525, MESSAGE_TEXT = 'Invalid CourseID; Format is LL(L)-DDD';
```

```
END IF;
```

```
IF NOT (NEW.Credits BETWEEN 0 AND 5)
```

```
THEN SIGNAL SQLSTATE 'HY000'
```

```
    SET MYSQL_ERRNO = 1525, MESSAGE_TEXT = 'Invalid Credits; Range is [0,5]!';
```

```
END IF;
```

```
SET NEW.Title = TRIM(LEADING ' ' FROM NEW.Title); #change value to be inserted
```

```
END $$
```

```
DELIMITER ;
```

CourseID	Title	DeptName	Credits
BIO-101	Intro. to Biology	Biology	4

Rejection if CourseID format is invalid or Credits not in [0,5].

Preprocessing of Title by removing leading blanks.

CourseID
BIO-101
BIO-301
BIO-399
CS-101
CS-190
CS-315
CS-319
CS-347
EE-181
FIN-201
HIS-351
MU-199
PHY-101

Trigger Test of Course_Before_Insert

```
INSERT Course VALUES ('CS-350','Data Warehouse','Comp. Sci.',3);      # Inserted OK
INSERT Course VALUES ('CS3700','Temporal Databases','Comp. Sci.',5); # Error Code1525
SHOW WARNINGS;
```

Level	Code	Message
Error	1525	Invalid CourseID; Format is LL(L)-DDD

```
INSERT Course VALUES ('CS-3700','Temporal Databases','Comp. Sci.',5); # Error Code1525
SHOW WARNINGS;
```

Level	Code	Message
Error	1525	Invalid CourseID; Format is LL(L)-DDD

```
INSERT Course VALUES ('CS-370','Temporal Databases','Comp. Sci.',7); # Error Code1525
SHOW WARNINGS;
```

Level	Code	Message
Error	1525	Invalid Credits; Range is [0,5]!

```
INSERT Course VALUES ('CS-370','Temporal Databases','Comp. Sci.',5); # Inserted OK
SELECT * FROM Course WHERE DeptName = 'Comp. Sci.';                 # No leading '_'
```

CourseID	Title	DeptName	Credits
CS-350	Data Warehouse	Comp. Sci.	3
CS-370	Temporal Databases	Comp. Sci.	5

User-defined Error Handling

- To raise a signal for a user-defined condition, use the **SIGNAL** statement.
- To declare a handler, use the **DECLARE ... HANDLER** statement.

```
DECLARE handler_action HANDLER
```

```
    FOR condition-value statement
```

- *Condition-value* = *mysql_or_mariadb_error_code* | **SQLSTATE** *sqlstate_value* | **SQLWARNING** | **NOT FOUND** | **SQLEXCEPTION** | ...
- If the *condition-value* occurs, the specified *statement* executes.
- The *handler_action* indicates what action the handler takes after execution of the statement:
 - **CONTINUE**: Execution of the current program continues.
 - **EXIT**: Execution terminates for the BEGIN ... END compound statement in which the handler is declared.

- Example of a handler: see demo exercise 5.1.4

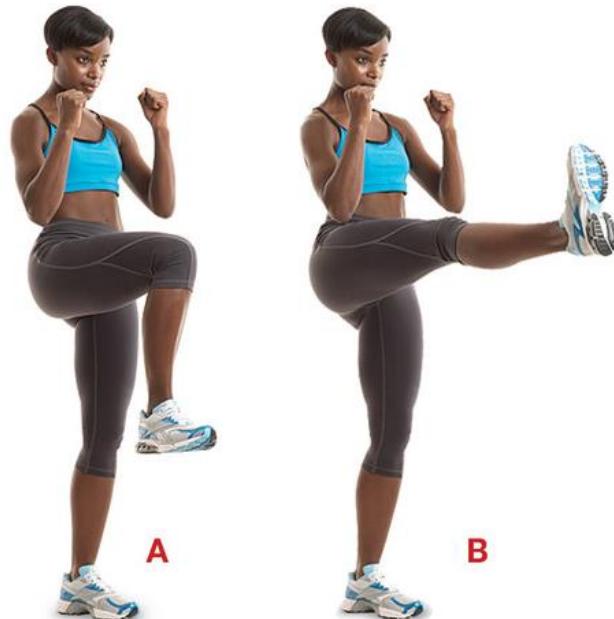
SQL Access from a Programming Language

- **Application Programs**
 - Often the application user interface and application logic are made in a programming language like Java, C, or Cobol.
- **Application Programs can interact with a database**
 - Using an Application Program Interface (API) providing functionality for
 - connecting with the database server
 - sending SQL commands to the database server
 - fetching tuples of a query result one-by-one into program variables
- **Java DataBase Connectivity (JDBC)**
 - API for Java
- **Open DataBase Connectivity (ODBC)**
 - Standard API. Works with a wide range of programming languages like C, C++, C#, Cobol and Visual Basic.

Embedded SQL

- The SQL standard defines embedding's of SQL in a variety of programming languages such as C, Java, and Cobol.
 - A language to which SQL queries are embedded is referred to as a *host language*, and the SQL structures permitted in the host language constitute *embedded SQL*.
 - Special statements are used to identify embedded SQL, like:
 - `EXEC SQL <embedded SQL statement > END_EXEC` for embedded Cobol
 - `# SQL { embedded SQL statement }` for embedded Java
 - A *precompiler* replaces the SQL statements in the application program with appropriate host language declarations and procedure calls that allow runtime execution of the database access.
 - This approach differs from JDBC and ODBC that use a dynamic SQL approach which does not use a precompiler, but allows the application program to construct and submit SQL queries as strings at runtime.

Demo Exercises



Demo Exercises clarify ideas and concepts from the Lecture to provide you with good Database Skills.

Discuss and do the Demo Exercises.
Solutions can be found on later slides.

Functions

5.1.1 Create a Function

Create a function which can check whether a year is a **leap year**, and then test the function.

Hint: Year is a leap year if

$(@Year \% 4 = 0) \text{ AND } ((@Year \% 100 <> 0) \text{ OR } (@Year \% 400 = 0))$,
where $n \% m$ is the remainder of n/m (e.g. $13 \% 5 = 3$)

Examples:

1972 is a leap year as (True AND (True OR False)) evaluates to True;

1900 is not a leap year (True AND (False OR False)) evaluates to False;

2000 is a leap year as (True AND (False OR True)) evaluates to True;

3000 is not a leap year (True AND (False OR False)) evaluates to False;

Triggers

5.1.2 Create a Trigger

named Instructor_After_Insert that after an instructor has been inserted into the Instructor table will insert the same row in a table named InstLog with a timestamp added.

Hint, for a timestamp **NOW(6)**:

SELECT NOW(6);

NOW(6)
2015-09-14 09:48:09.949155

Procedures

5.1.3 Create a Procedure

As a continuation of 5.1.2, create a procedure called InstBackup that will copy all rows from the Instructor table to a table called InstOld, and thereafter also delete all rows in the table InstLog.

The procedure should delete the rows in the old backup table (i.e. InstOld), make a new backup table with the rows of the Instructor table, and delete the rows in InstLog to prepare for recording new inserts into Instructor.

Remark: in MariaDB/MySQL, in order to be allowed to make a DELETE without a where clause, you need to disable safe mode (if not already done): **SET SQL_SAFE_UPDATES = 0;**

Procedures & Transactions

5.1.4 Create a Procedure with a Transaction

As a continuation of 5.1.3, redefine the procedure called InstBackup so that all data manipulations are done within a transaction. Test it.

```
DROP TABLE IF EXISTS InstOld;
DROP TABLE IF EXISTS InstLog;
CREATE TABLE InstOld LIKE Instructor;
CREATE TABLE InstLog LIKE Instructor;
ALTER TABLE InstLog ADD LogTime TIMESTAMP(6);

DELIMITER //
CREATE PROCEDURE InstBackup1 ()
BEGIN
    DECLARE vSQLSTATE CHAR(5) DEFAULT '00000';
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
        # this handler statement below ensures that
        # if an exception is raised by SQL during the transaction
        # then vSQLSTATE will be assigned a value <> '00000'
        # and continue
    BEGIN
        GET DIAGNOSTICS CONDITION 1
        vSQLSTATE = RETURNED_SQLSTATE;
    END;
    ... fill out here
END //
DELIMITER ;
```

... test it ...

Events

5.1.5 Create an Event

As a continuation of 5.1.4, create an event named InstEvent that will execute the called InstBackup every week the night between Saturday and Sunday at 00:00:01, first time 2016-02-21.

Remember to set the GLOBAL Event_Scheduler to 1, to turn the event on.

Events

5.1.6 Create an Event making a dice roll

Design a Gambling Machine which rolls a dice every 5 seconds:



1. Set the GLOBAL Event_Scheduler to 1.
2. Create a table DiceRolls with attributes RollNo and DiceEyes of data type integer.
Hint: If you add AUTO_INCREMENT after the type of the RollNo attribute, then each time you insert a new row in the table, you need only stating the value of DiceEyes – the value of RollNo will automatically be generated: First time it will be 1, then 2, etc.
3. Create an event RollDice that executes every 5 seconds and inserts a random number of 1, 2, 3, 4, 5 or 6 (DiceEyes) into table DiceRolls.

Hint:

RAND() returns a random floating-point value v in the range $0 \leq v < 1.0$.
FLOOR() returns the largest integer value not greater than the argument.

Make a query showing the number of times the six dice values have been played within the first 10 rolls.

Solutions to Demo Exercises



Functions

5.1.1 Create a Function

Create a function which can check whether a year is a **leap year**, and then test the function.

Hint: Year is a leap year if

$(@Year \% 4 = 0) \text{ AND } ((@Year \% 100 <> 0) \text{ OR } (@Year \% 400 = 0))$,
where $n \% m$ is the remainder of n/m (e.g. $13 \% 5 = 3$)

Examples:

1972 is a leap year as (True AND (True OR False)) evaluates to True;
1900 is not a leap year (True AND (False OR False)) evaluates to False;

2000 is a leap year as (True AND (False OR True)) evaluates to True;
3000 is not a leap year (True AND (False OR False)) evaluates to False;

```
CREATE FUNCTION LeapYear ( vYear YEAR ) RETURNS BOOLEAN
RETURN
(vYear % 4 = 0) AND ((vYear % 100 <> 0) OR (vYear % 400 = 0));
```

```
SELECT LeapYear(1964);
```

Leapyear(1964)
1

```
SELECT
```

```
StudID, StudName, Birth, Age(Birth) AS Age, LeapYear(YEAR(Birth))
AS LeapYear
FROM Student;
```

StudID	StudName	Birth	Age	LeapYear
00128	Zhang	1992-04-18	23	1
12345	Shankar	1995-12-06	20	0
19991	Brandt	1993-05-24	22	0
23121	Chavez	1992-04-18	23	1
44553	Peltier	1995-10-18	20	0
45678	Levy	1995-08-01	20	0
54321	Williams	1995-02-28	20	0

Triggers

5.1.2 Create a Trigger

named `Instructor_After_Insert` that after an instructor has been inserted into the `Instructor` table will insert the same row in a table named `InstLog` with a timestamp added.

Hint, for a timestamp `NOW(6)`:

```
SELECT NOW(6);
```

<code>NOW(6)</code>
2015-09-14 09:48:09.949155

```
CREATE TABLE InstLog LIKE Instructor;  
ALTER TABLE InstLog ADD LogTime TIMESTAMP(6);
```

```
DELIMITER //  
CREATE TRIGGER Instructor_After_Insert  
AFTER INSERT ON Instructor FOR EACH ROW  
BEGIN  
    INSERT InstLog VALUES (New.InstID,  
                          New.InsName,  
                          New.DeptName,  
                          New.Salary,  
                          NOW(6));  
END //  
DELIMITER ;
```

```
SELECT * FROM Instructor;  
SELECT * FROM InstLog;
```

```
INSERT Instructor VALUES  
('11001', 'Valdez', 'Comp. Sci.', 36000),  
('11002', 'Koerver', 'Comp. Sci.', 36000);
```

```
SELECT * FROM Instructor;  
SELECT * FROM InstLog;
```

InstID	InstName	DeptName	Salary	LogTime
11001	Valdez	Comp. Sci.	36000.00	2015-09-14 09:59:36.122563
11002	Koerver	Comp. Sci.	36000.00	2015-09-14 09:59:36.122563

Procedures

5.1.3 Create a Procedure

As a continuation of 5.1.2, create a procedure called InstBackup that will copy all rows from the Instructor table to a table called InstOld, and thereafter also delete all rows in the table InstLog.

The procedure should delete the rows in the old backup table (i.e. InstOld), make a new backup table with the rows of the Instructor table, and delete the rows in InstLog to prepare for recording new inserts into Instructor.

Remark: in MariaDB/MySQL, in order to be allowed to make a DELETE without a where clause, you need to disable safe mode (if not already done): **SET SQL_SAFE_UPDATES = 0;**

```
CREATE TABLE InstOld LIKE Instructor;  
  
DELIMITER //  
CREATE PROCEDURE InstBackup ()  
BEGIN  
    DELETE FROM InstOld; # see remark in left column  
    INSERT INTO InstOld SELECT * FROM Instructor;  
    DELETE FROM InstLog;  
END //  
DELIMITER ;  
  
SELECT * FROM Instructor;  
SELECT * FROM InstOld;  
SELECT * FROM InstLog;  
  
SET SQL_SAFE_UPDATES = 0;  
CALL InstBackup;  
SET SQL_SAFE_UPDATES = 1;  
  
SELECT * FROM Instructor;  
SELECT * FROM InstOld; # contains the Instructor rows  
SELECT * FROM InstLog; # no rows
```

Procedures & Transactions

5.1.4 Create a Procedure with a Transaction

As a continuation of 5.1.3, redefine the procedure called InstBackup so that all data manipulations are done within a transaction. Test it.

```
DROP TABLE IF EXISTS InstOld; DROP TABLE IF EXISTS InstLog;
CREATE TABLE InstOld LIKE Instructor; CREATE TABLE InstLog LIKE Instructor;
ALTER TABLE InstLog ADD LogTime TIMESTAMP(6);

DELIMITER //
CREATE PROCEDURE InstBackup1()
BEGIN
    DECLARE vSQLSTATE CHAR(5) DEFAULT '00000';
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
        # this handler statement below ensures that
        # if an exception is raised by SQL during the transaction
        # then vSQLSTATE will be assigned a value <> '00000'
        # and continue
    BEGIN
        GET DIAGNOSTICS CONDITION 1
        vSQLSTATE = RETURNED_SQLSTATE;
    END;
    START TRANSACTION;
    DELETE FROM InstOld;
    INSERT INTO InstOld SELECT * FROM Instructor;
    DELETE FROM InstLog;
    SELECT vSQLSTATE;
    IF vSQLSTATE = '00000' THEN COMMIT;
        ELSE ROLLBACK;
    END IF;
END //
DELIMITER ;

# Test the procedure before and after "DROP TABLE InstLog;".
INSERT Instructor VALUES ('10000', 'Hansen', 'Comp. Sci.', 50000);
SELECT * FROM InstLog; # contains the new row
SET SQL_SAFE_UPDATES = 0;
CALL InstBackup1; # SELECT vSQLSTATE returns 00000 and the transaction is
committed
SELECT * FROM Instructor;
SELECT * FROM InstOld; #same as Instructor
SELECT * FROM InstLog; #no rows

DROP TABLE InstLog;
CALL InstBackup1; # SELECT vSQLSTATE returns 42S02 and the transaction is rolled
back as Instlog does not exist

# Remove "SELECT vSQLSTATE;" inside the procedure when testing has been done!
```

Events

5.1.5 Create an Event

As a continuation of 5.1.4, create an event named InstEvent that will execute the called InstBackup every week the night between Saturday and Sunday at 00:00:01, first time 2016-02-21.

Remember to set the GLOBAL Event_Scheduler to 1, to turn the event on.

#re-create table deleted in 5.1.4 solution

```
CREATE TABLE InstLog LIKE Instructor;  
ALTER TABLE InstLog ADD LogTime TIMESTAMP(6);
```

```
CREATE EVENT InstEvent  
ON SCHEDULE EVERY 1 WEEK  
STARTS '2016-02-21 00:00:01'  
DO CALL InstBackup;
```

```
SET GLOBAL event_scheduler = 1;  
SHOW VARIABLES LIKE 'event_scheduler';
```

Variable_name	Value
event_scheduler	ON

Events

5.1.6 Create an Event making a dice roll

Design a Gambling Machine which rolls a dice every 5 seconds:

1. Set the **GLOBAL Event_Scheduler** to 1.
2. Create a table **DiceRolls** with attributes **RollNo** and **DiceEyes** of data type **integer**.
Hint: If you add **AUTO_INCREMENT** after the type of the **RollNo** attribute, then each time you insert a new row in the table, you need only stating the value of **DiceEyes** – the value of **RollNo** will automatically be generated: First time it will be 1, then 2, etc.
3. Create an event **RollDice** that executes every 5 seconds and inserts a random number of 1, 2, 3, 4, 5 or 6 (**DiceEyes**) into table **DiceRolls**.

Hint:

RAND() returns a random floating-point value **v** in the range $0 \leq v < 1.0$.

FLOOR() returns the largest integer value not greater than the argument.

Make a query showing the number of times the six dice values have been played within the first 10 rolls.



```
SET GLOBAL event_scheduler = 1;
```

```
CREATE TABLE DiceRolls (  
  RollNo INTEGER AUTO_INCREMENT PRIMARY KEY,  
  DiceEyes INTEGER);
```

```
CREATE EVENT RollDice  
ON SCHEDULE EVERY 5 SECOND  
DO  
INSERT DiceRolls (DiceEyes) VALUES  
(1+FLOOR(6*RAND()));
```

```
SELECT DISTINCT DiceEyes, Count(DiceEyes)  
FROM DiceRolls WHERE RollNo <= 10  
GROUP BY DiceEyes;
```

#stop event after use

```
SET GLOBAL event_scheduler = 0;
```

Exercises



Please answer all exercises
to demonstrate your
Database Skills.

Solutions are available at 11:45

Functions

5.2.1 Create a Function

1. Create a function named *BuildingCapacityFct* which takes as input a *Building* of the *Classroom* table in the *University* database and returns the total capacity of the building.
2. Test the function (i.e. execute an SQL command containing a function call of that function).

Triggers, Procedures & Error Signalling

5.2.2 Procedure with Error Signalling

For the *TimeSlot* table of the *University* database, state informally constraints (besides the key constraint) that should hold between the values of attributes in a single row and between values of attributes of two rows. Check your suggestion with a teaching assistant before continuing.

Create a procedure *InsertTimeSlot* for inserting a row into the *TimeSlot* table. It should signal an error in case the insertion of the new tuple leads to a violation of the constraints. (The procedure should not check whether the constraints already hold before the insertion.) To make the procedure more readable, it is advisable to define one or several auxiliary/helper functions to express the error conditions.

Test systematically the auxiliary functions and procedure.

5.2.3 Trigger with Error Signalling

Make a trigger *TimeSlot_Before_Insert*, which automatically raises a signal when inserting a row into *TimeSlot* (directly with an *INSERT* without using the *InsertTimeSlot* procedure), if the insertion of the new row leads to a violation of the constraints identified in the previous exercise.

Test the trigger by making *INSERTs* into *TimeSlot*.

Events

5.2.4 Create an Event for a European Roulette

Design a Gambling Machine which rolls a ball on a European Roulette every 10 seconds and stores the Lucky number.

Create a table called BallRolls with attributes RollNo and LuckyNo.

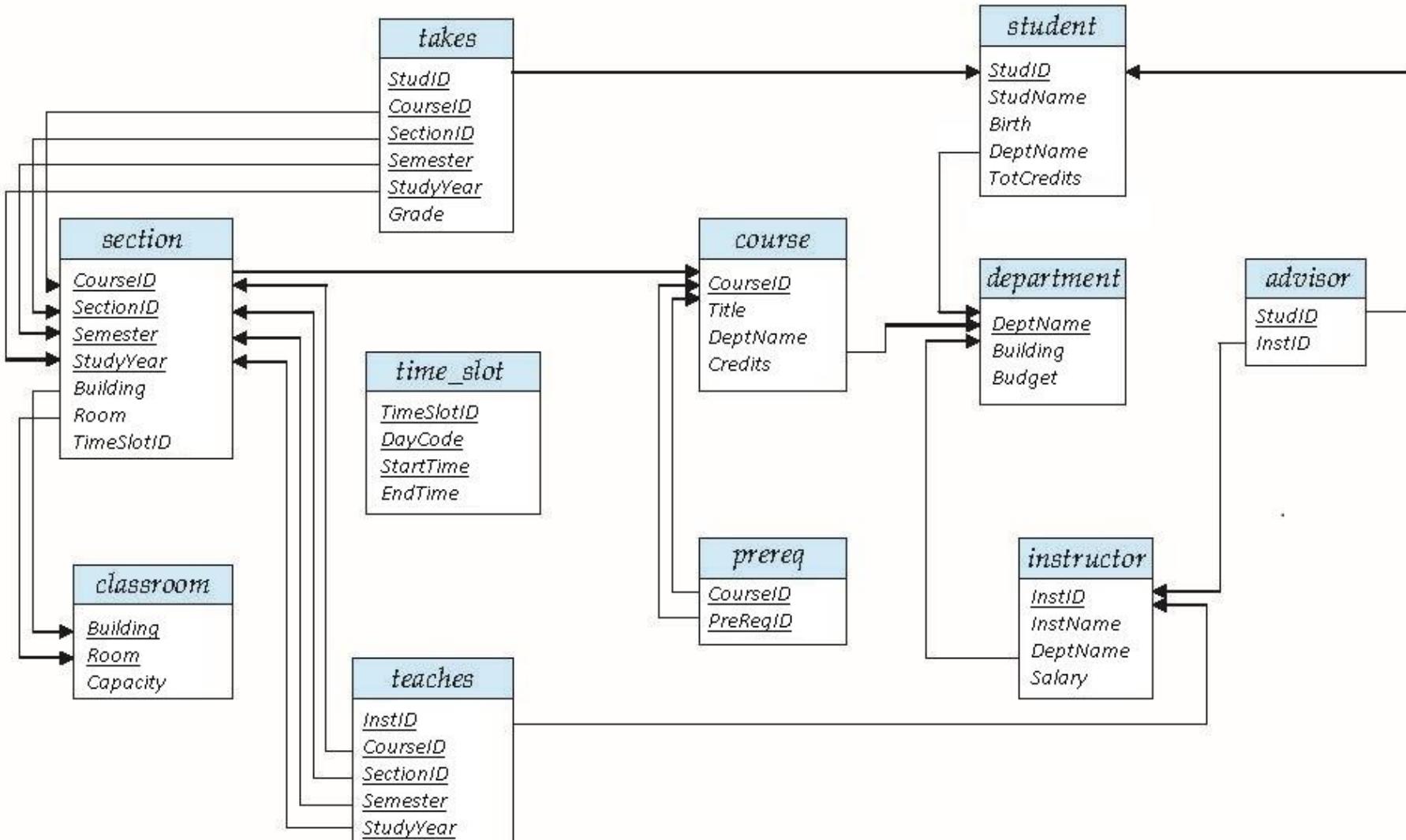
Create an event *RollBall* that executes every 10 seconds and inserts RollNo (automatically counting from 1) and LuckyNo (i.e. random number between 0 and 36) into the table BallRolls.



Appendix: University Database

- Database example used throughout the textbook and in this course!
 1. Database Schema Diagram
 2. Database Schema (CREATE TABLE Commands)
 3. Database Instance

Database Schema Diagram

 from the book, but with new names

University Database Schema (1 of 4)

```
CREATE TABLE Instructor
  (InstID          VARCHAR(5),
   InstName        VARCHAR(20) NOT NULL,
   DeptName        VARCHAR(20),
   Salary          DECIMAL(8,2),
   PRIMARY KEY (InstID),
   FOREIGN KEY(DeptName) REFERENCES Department(DeptName) ON DELETE SET NULL
  );

CREATE TABLE Student
  (StudID          VARCHAR(5),
   StudName        VARCHAR(20) NOT NULL,
   Birth           DATE,
   DeptName        VARCHAR(20),
   TotCredits      DECIMAL(3,0),
   PRIMARY KEY(StudID),
   FOREIGN KEY(DeptName) REFERENCES Department(DeptName) ON DELETE SET NULL
  );

CREATE TABLE Advisor
  (StudID          VARCHAR(5),
   InstID          VARCHAR(5),
   PRIMARY KEY(StudID),
   FOREIGN KEY(InstID) REFERENCES Instructor(InstID) ON DELETE SET NULL,
   FOREIGN KEY(StudID) REFERENCES Student(StudID) ON DELETE CASCADE
  );

CREATE TABLE Department
  (DeptName        VARCHAR(20),
   Building         VARCHAR(15),
   Budget           DECIMAL(12,2),
   PRIMARY KEY(DeptName)
  );
```

University Database Schema (2 of 4)

```
CREATE TABLE Course
  (CourseID          VARCHAR(8),
   Title             VARCHAR(50),
   DeptName          VARCHAR(20),
   Credits           DECIMAL(2,0),
   PRIMARY KEY(CourseID),
   FOREIGN KEY(DeptName) REFERENCES Department(DeptName) ON DELETE SET NULL
  );

CREATE TABLE PreReq
  (CourseID          VARCHAR(8),
   PreReqID          VARCHAR(8),
   PRIMARY KEY(CourseID, PreReqID),
   FOREIGN KEY(CourseID) REFERENCES Course(CourseID) ON DELETE CASCADE,
   FOREIGN KEY(PreReqID) REFERENCES Course(CourseID)
  );
```

University Database Schema (3 of 4)

```
CREATE TABLE Teaches
  (InstID          VARCHAR(5),
   CourseID        VARCHAR(8),
   SectionID       VARCHAR(8),
   Semester        ENUM('Fall','Winter','Spring','Summer'),
   StudyYear       YEAR,
   PRIMARY KEY(InstID, CourseID, SectionID, Semester, StudyYear),
   FOREIGN KEY(CourseID, SectionID, Semester, StudyYear)
     REFERENCES Section(CourseID, SectionID, Semester, StudyYear) ON DELETE CASCADE,
   FOREIGN KEY(InstID) REFERENCES Instructor(InstID) ON DELETE CASCADE
);

CREATE TABLE Takes
  (StudID          VARCHAR(5),
   CourseID        VARCHAR(8),
   SectionID       VARCHAR(8),
   Semester        ENUM('Fall','Winter','Spring','Summer'),
   StudyYear       YEAR,
   Grade           VARCHAR(2),
   PRIMARY KEY(StudID, CourseID, SectionID, Semester, StudyYear),
   FOREIGN KEY(CourseID, SectionID, Semester, StudyYear)
     REFERENCES Section(CourseID, SectionID, Semester, StudyYear) ON DELETE CASCADE,
   FOREIGN KEY(StudID) REFERENCES Student(StudID) ON DELETE CASCADE
);
```

University Database Schema (4 of 4)

```
CREATE TABLE Section
  (CourseID          VARCHAR(8),
  SectionID          VARCHAR(8),
  Semester           ENUM('Fall','Winter','Spring','Summer'),
  StudyYear          YEAR,
  Building           VARCHAR(15),
  Room                VARCHAR(7),
  TimeSlotID         VARCHAR(4),
  PRIMARY KEY(CourseID, SectionID, Semester, StudyYear),
  FOREIGN KEY(CourseID) REFERENCES Course(CourseID) ON DELETE CASCADE,
  FOREIGN KEY(Building, Room) REFERENCES Classroom(Building, Room) ON DELETE SET NULL
);

CREATE TABLE TimeSlot
  (TimeSlotID    VARCHAR(4),
  DayCode        ENUM('M','T','W','R','F','S','U'),
  StartTime      TIME,
  EndTime        TIME,
  PRIMARY KEY(TimeSlotID, DayCode, StartTime)
);

CREATE TABLE Classroom
  (Building          VARCHAR(15),
  Room                VARCHAR(7),
  Capacity           DECIMAL(4,0),
  PRIMARY KEY(Building, Room)
);
```

Database Instance (1 of 4)

Instructor

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

Student

StudID	StudName	Birth	DeptName	TotCredits
00128	Zhang	1992-04-18	Comp. Sci.	102
12345	Shankar	1995-12-06	Comp. Sci.	32
19991	Brandt	1993-05-24	History	80
23121	Chavez	1992-04-18	Finance	110
44553	Peltier	1995-10-18	Physics	56
45678	Levy	1995-08-01	Physics	46
54321	Williams	1995-02-28	Comp. Sci.	54
55739	Sanchez	1995-06-04	Music	38
70557	Snow	1995-11-22	Physics	0
76543	Brown	1994-03-05	Comp. Sci.	58
76653	Aoi	1993-09-18	Elec. Eng.	60
98765	Bourikas	1992-09-23	Elec. Eng.	98
98988	Tanaka	1992-06-02	Biology	120

Advisor

StudID	InstID
12345	10101
44553	22222
45678	22222
00128	45565
76543	45565
23121	76543
98988	76766
76653	98345
98765	98345

Department

DeptName	Building	Budget
Biology	Watson	90000.00
Comp. Sci.	Taylor	100000.00
Elec. Eng.	Taylor	85000.00
Finance	Painter	120000.00
History	Painter	50000.00
Music	Packard	80000.00
Physics	Watson	70000.00

Database Instance (2 of 4)

Teaches

InstID	CourseID	SectionID	Semester	StudyYear
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
10101	CS-101	1	Fall	2009
45565	CS-101	1	Spring	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
10101	CS-315	1	Spring	2010
45565	CS-319	1	Spring	2010
83821	CS-319	2	Spring	2010
10101	CS-347	1	Fall	2009
98345	EE-181	1	Spring	2009
12121	FIN-201	1	Spring	2010
32343	HIS-351	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

Takes

StudID	CourseID	SectionID	Semester	StudyYear	Grade
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	NULL

Database Instance (3 of 4)

Course

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

PreReq

CourseID	PreReqID
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Database Instance (4 of 4)

Section

CourseID	SectionID	Semester	StudyYear	Building	Room	TimeSlotID
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Classroom

Building	Room	Capacity
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

TimeSlot

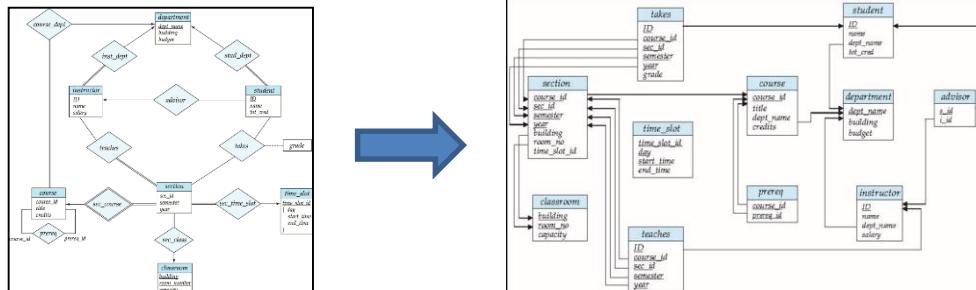
TimeSlotID	DayCode	StartTime	EndTime
A	M	08:00:00	08:50:00
A	W	08:00:00	08:50:00
A	F	08:00:00	08:50:00
B	M	09:00:00	09:50:00
B	W	09:00:00	09:50:00
B	F	09:00:00	09:50:00
C	M	11:00:00	11:50:00
C	W	11:00:00	11:50:00
C	F	11:00:00	11:50:00
D	M	13:00:00	13:50:00
D	W	13:00:00	13:50:00
D	F	13:00:00	13:50:00
E	T	10:30:00	11:45:00
E	R	10:30:00	11:45:00
F	T	14:30:00	15:45:00
F	R	14:30:00	15:45:00
G	M	16:00:00	16:50:00
G	W	16:00:00	16:50:00
G	F	16:00:00	16:50:00
H	W	10:00:00	12:30:00

Database Design and Entity-Relationship Diagrams

Modelling real world systems with Entities and Relationships
 Chapter 7 in the textbook

©Anne Haxthausen and Flemming Schmidt

These slides have been prepared by Anne Haxthausen, partly reusing/modifying slides by Flemming Schmidt. Some examples come from Silberschatz, Korth, Sudarshan, 2010.



Database Design

- Three main phases

1. *Conceptual design:*

- Is the process of constructing a *conceptual model* of the data used in an enterprise, without taking any decisions on how the data should be logically or physically implemented in a DBMS.
- It is common to use *Entity-Relationship (E-R) models* as conceptual models. An ER model describes the entities and relationships that should be captured in the database, but without saying how these entities and their relationships should be captured in tables.

2. *Logical design:*

- Is the process of constructing a *logical model* of the data, for a specific type of databases (e.g. relational, object, or ...), but *independent of the DBMS* (e.g. MySQL, DB2, Oracle etc).
- For relational databases, the logical model is *a relational database schema*, i.e. a collection of relation/table schemas.

3. *Physical design:*

- Is the process of describing how the database *should* be implemented on *a specific DBMS*. The physical features of the database system are specified. These include e.g. the choice of file organization and indexes.

- Objectives today:

1. Show how to use *Entity-Relationship (E-R) modelling* to build a conceptual data model.
2. Show how *to convert* an E-R model into a logical design for the relational model.

E-R Modelling with E-R Diagrams

- Entity Sets and Relationship Sets
- Attributes
- Order, Cardinality and Participation
- Keys for Entities and Relationships
- Entity-Relationship (E-R) Diagrams
- Strong and Weak Entities
- E-R Diagram for the University Database
- Summary of the Textbook Adapted UML Notation
- Other Diagram Notations

Entities and Relationships

The data of an enterprise can be modelled as entities and relationships.

- An entity e
 - Is a thing/object, e.g. a specific student.
- An entity set E
 - Is a set of entities representing objects of the same type (having the same attributes).
 - Examples: the set *student* of all students.
- Entity attributes
 - Each entity has properties called entity attributes.
 - Examples of *student* Attributes: *ID* and *name*.
- A relationship
 - Is an association (e_1, \dots, e_n) among several entities e_i .
- A relationship set among entity sets E_1, \dots, E_n
 - Is a set of relationships of the same type, i.e. a subset $\{(e_1, \dots, e_n) \mid e_i \in E_i\}$.
 - Examples: *advisor*
- Relationship attributes
 - Each relationship can also have properties called relationship attributes.
 - Example of an *advisor* attribute: *date* (for start date).

Two Entity Sets

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

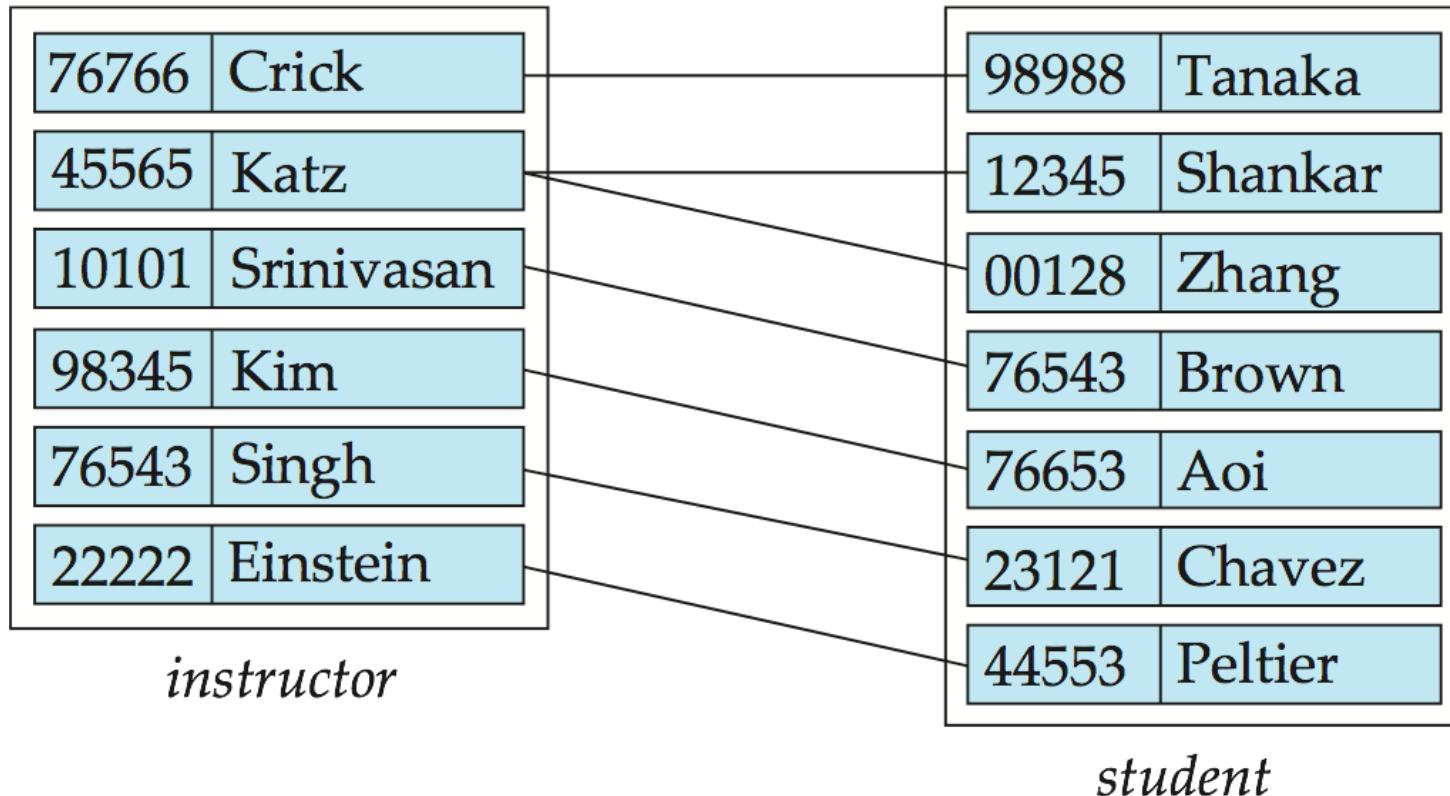
instructor

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

student

PS. In this presentation names in the University Database have been altered slightly to conform with the textbook, e.g. ID instead of InstID and name instead InstName.

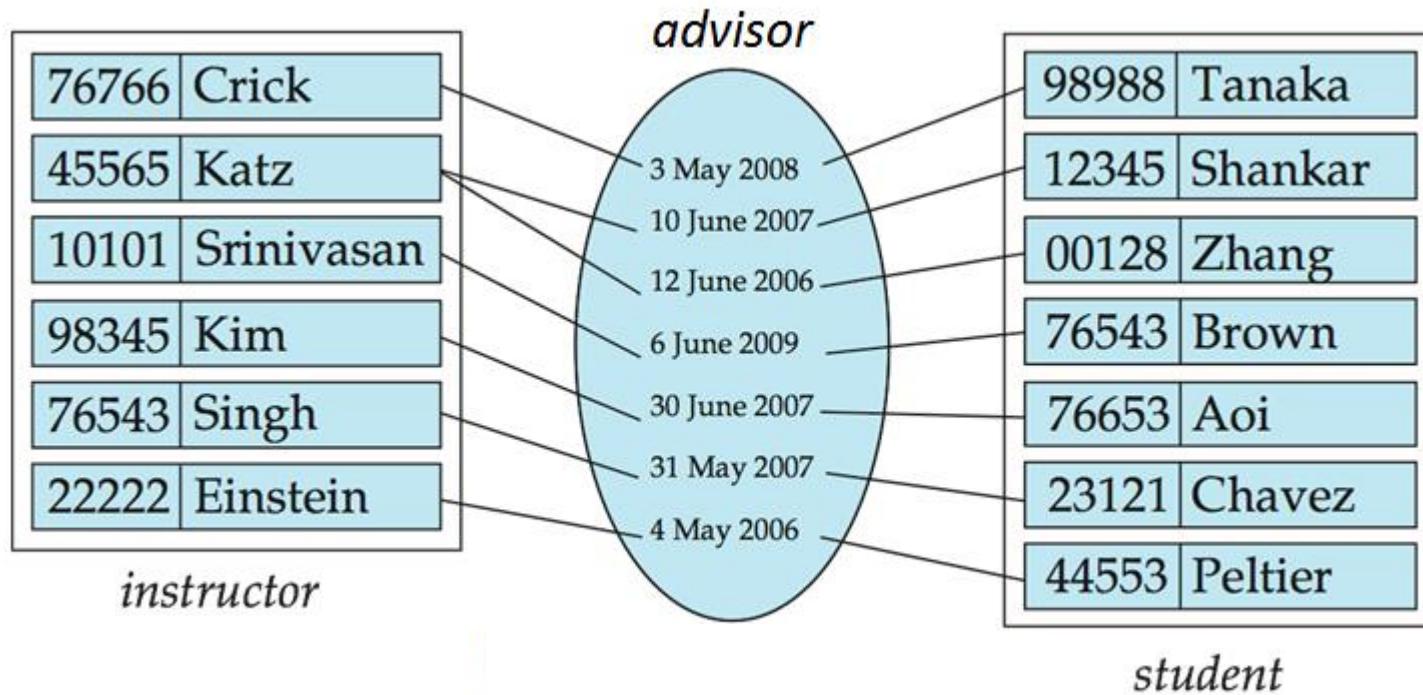
Relationship Set *advisor*



Each line illustrates a relationship.

Relationship Attributes - Example

- An *advisor* relationship between an *instructor* entity and a *student* entity may have the *attribute date*, which tracks, when the relationship was initiated.
- E.g., 4 May 2006 the instructor Einstein became advisor for the student Peltier.



Attribute Types

■ Attributes

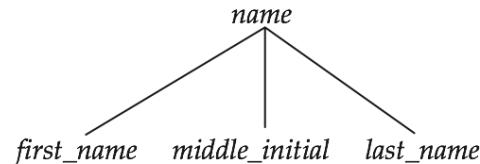
- are descriptive properties possessed by all elements of an entity set or a relationship set.

■ The Domain of an Attribute

- is the set of permitted values for the attribute.

■ Attribute Types

- *Simple (atomic)* attributes.
- *Composite* attributes, which consist of component attributes.

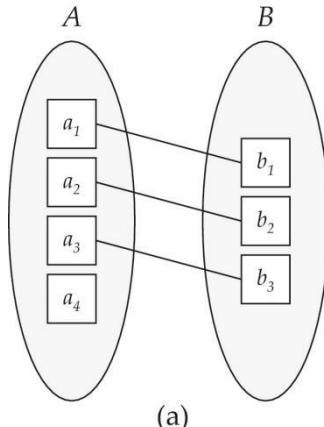


- *Multivalued* attributes, which is a set of attribute values. E.g. *phone_numbers*.
- *Derived* attributes, which are computed from other attributes.

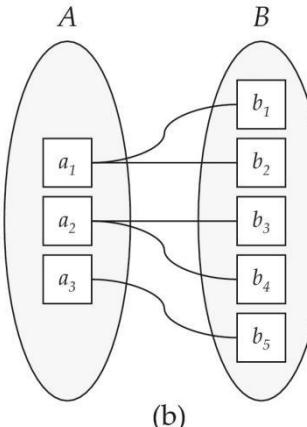
Order, Cardinality and Participation

- Order/Degree for relationship set:
 - Is the number of entities (entity sets) being associated in a relationship (set)
 - If two entities (entity sets) are associated then the relationship (set) is *Binary*
 - If three entities (entity sets) are associated then the relationship (set) is *Ternary* etc.
- Cardinality for binary relationship set between entity sets E_1 and E_2 :
 - **one-to-one**: An element in E_1 can be associated with at most **one** (0..1) element in E_2 , and vice versa.
 - **one-to-many**: An element in E_1 can be associated with **many** (0..*) elements in E_2 , and an element in E_2 can be associated with at most **one** (0..1) element in E_1 .
 - **many-to-one**: An element in E_1 can be associated with at most **one** (0..1) element in E_2 , and an element in E_2 can be associated with **many** (0..*) elements in E_1 .
 - **many-to-many**: An element in E_1 can be associated with **many** (0..*) elements in E_2 , and an element in E_2 can be associated with **many** (0..*) elements in E_1 .
- Participation in binary relationship set between entity sets E_1 and E_2 :
 - **Total Participation** of E_1 in a relationship set between E_1 and E_2 :
All elements of E_1 participate in at least one association with elements in E_2 .
(So associations with 0 elements is not allowed.)
 - **Partial Participation** of E_1 in a relationship set between E_1 and E_2 :
Some elements in E_1 may not participate in any association with elements in E_2 .
(So association with 0 elements is allowed.)

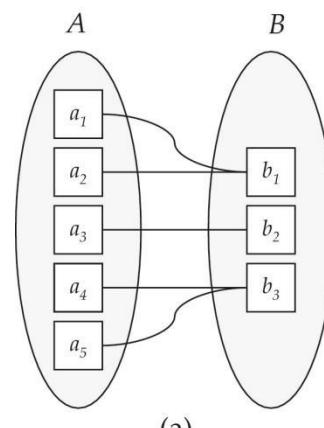
Cardinality and Participation



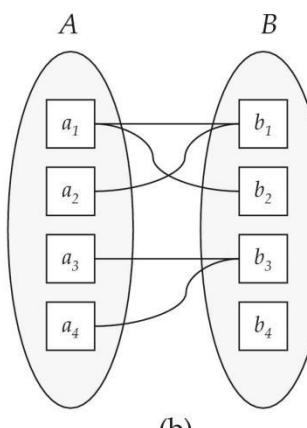
one-to-one
A partial, B total



one-to-many
A total, B total



many-to-one
A total B total



many-to-many
A total, B partial

Key for an Entity Set

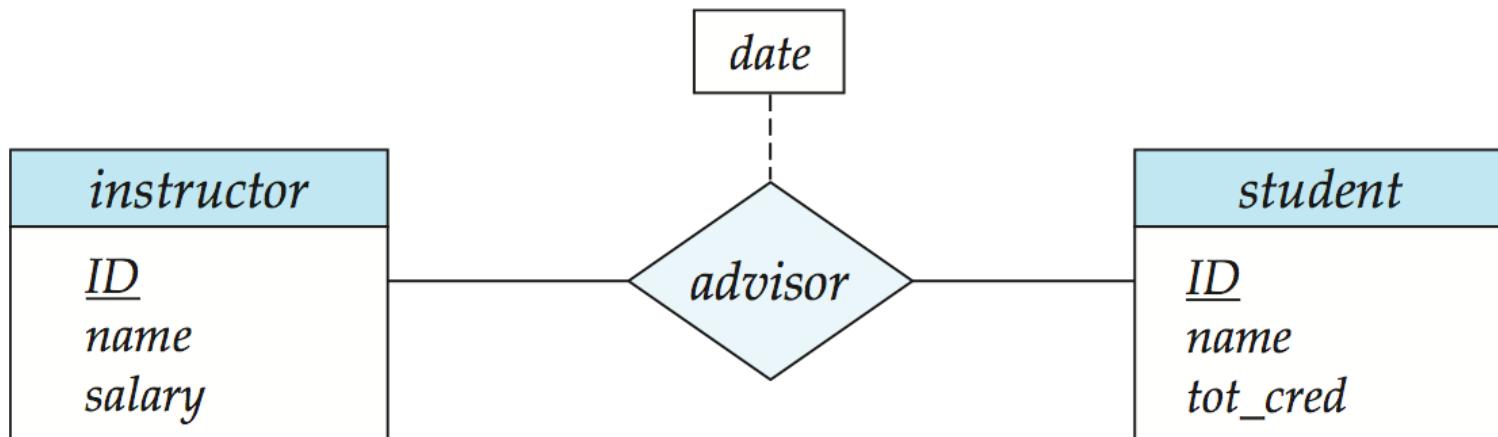
- A (Super) Key of an Entity Set
 - A set of one or more attributes whose values uniquely determine each Entity element in an Entity Set.
- A Candidate Key of an Entity Set
 - Is a Super Key which is minimal (i.e. has no other super key as a proper subset)
 - Example: *ID* is candidate key of *instructor*
- A Candidate Key is selected as Primary Key
 - Although several candidate keys may exist, one of the candidate keys is selected to be the Primary Key.

Key for a Relationship Set

- A Super Key of a Relationship Set
 - The combination of primary keys of the participating entity sets **and the attributes of the relationship set** forms a super key of a relationship set.
 - Example: *(student.ID, instructor.ID)* is a super key of *advisor*.
- Deciding Candidate Keys of a Relationship Set
 - Must consider the semantics of the relationship set to decide whether attributes of the relationship set should contribute to candidate keys. (Usually they do not contribute, but if it is multivalued it contributes, see demo exercise 6.4.)
 - Must consider the cardinality of the relationship set when deciding what are the candidate keys. For instance, for *advisor*
 - *(student.ID, instructor.ID)* is a candidate key, if the *advisor* relation is many-many.
 - *student.ID* is a candidate key, if the *advisor* relation is many-one (with *student* on the many side).
- Selecting the Primary Key
 - Remember to consider the semantics of the relationship set when selecting the primary key, since there can be more than one candidate key.

Entity-Relationship Diagrams

- An E-R diagram is a diagrammatic representation of an E-R model. There are many *different* E-R diagram notations, such the Textbook's **Adapted UML Class Diagram** notation.
- *Entity sets* are represented by rectangles with a name in the blue top.
 - *Entity attributes* are listed inside the entity rectangle.
 - *Primary keys* are underlined.
- *Relationship sets* are represented by a diamond with a name listed inside, and lines connecting to the participating entities.
 - *Relationship attributes* are listed in a box, linked with a dashed line.

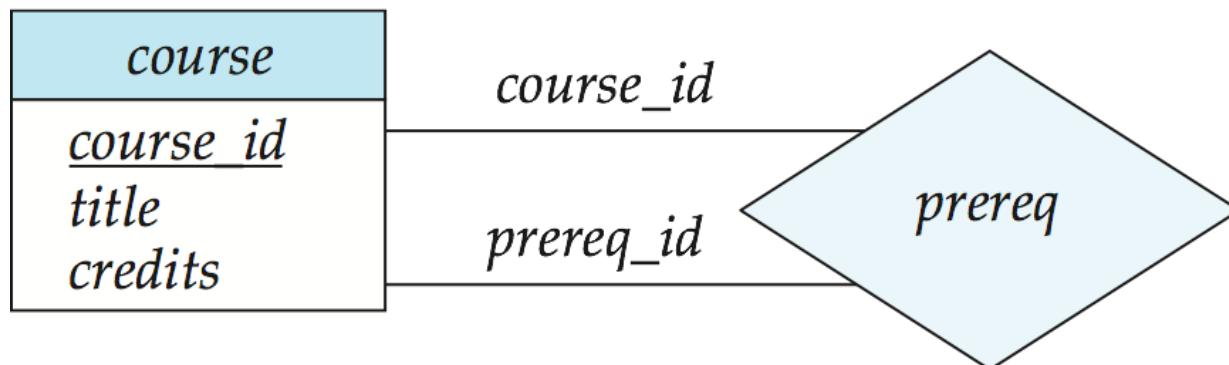


Notation to Express Complex Attributes

<i>instructor</i>	
<i>ID</i>	atomic attribute
<i>name</i>	composite attribute
<i>first_name</i>	
<i>middle_initial</i>	
<i>last_name</i>	
<i>address</i>	composite attribute
<i>street</i>	
<i>street_number</i>	
<i>street_name</i>	
<i>apt_number</i>	
<i>city</i>	
<i>state</i>	
<i>zip</i>	
{ <i>phone_number</i> }	multivalued attribute
<i>date_of_birth</i>	atomic attribute
<i>age</i> ()	derived attribute

Roles of Relationship Sets

- The entity sets of a relationship set need not be distinct
 - Each occurrence of an entity set plays a “role” in the relationship set.
 - The labels “*course_id*” and “*prereq_id*” are called *roles*.



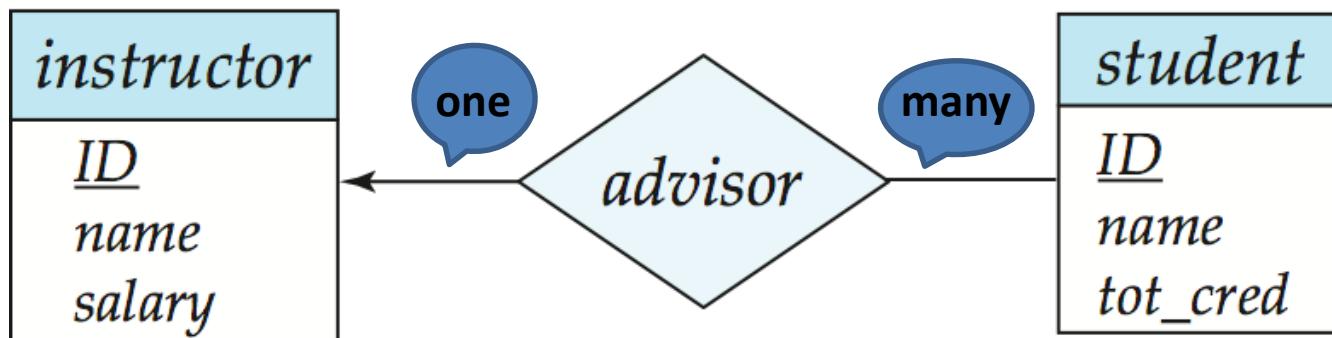
Drawing Cardinality Constraints

Drawing Cardinality Constraints

- We express cardinality constraints by drawing either a directed line (\rightarrow), signifying “one,” or an undirected line ($-$), signifying “many,” between the relationship set and the entity set.

Example:

- One-to-Many** Relationship Set between *instructor* and *student*:
An instructor is associated with several students via *advisor*.
A student is associated with at most one instructor via *advisor*.



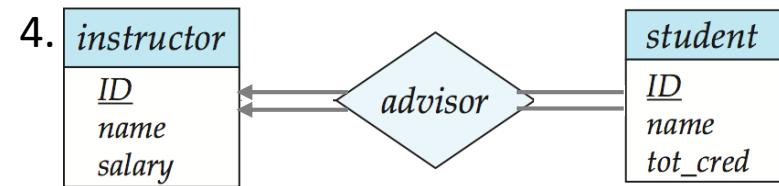
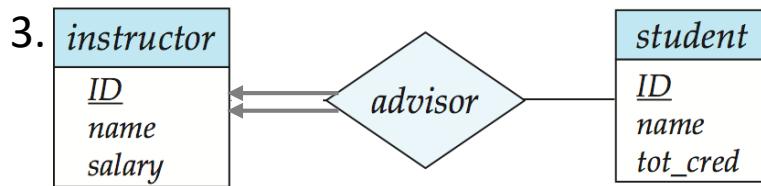
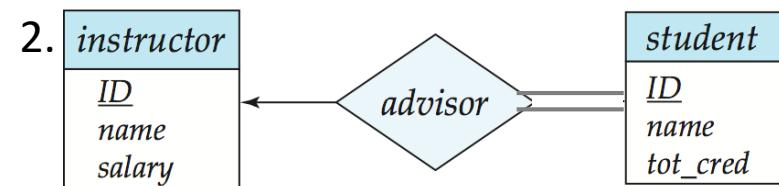
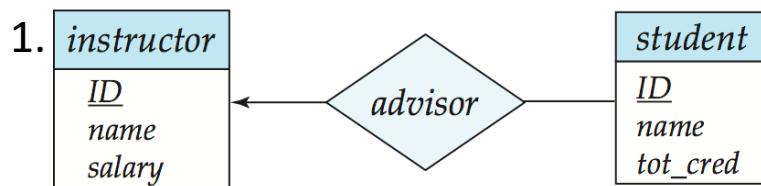
Participation of an Entity Set in a Relationship Set

■ Total Participation

- Every entity element participates in at least one relationship. This is shown by a **double line**.

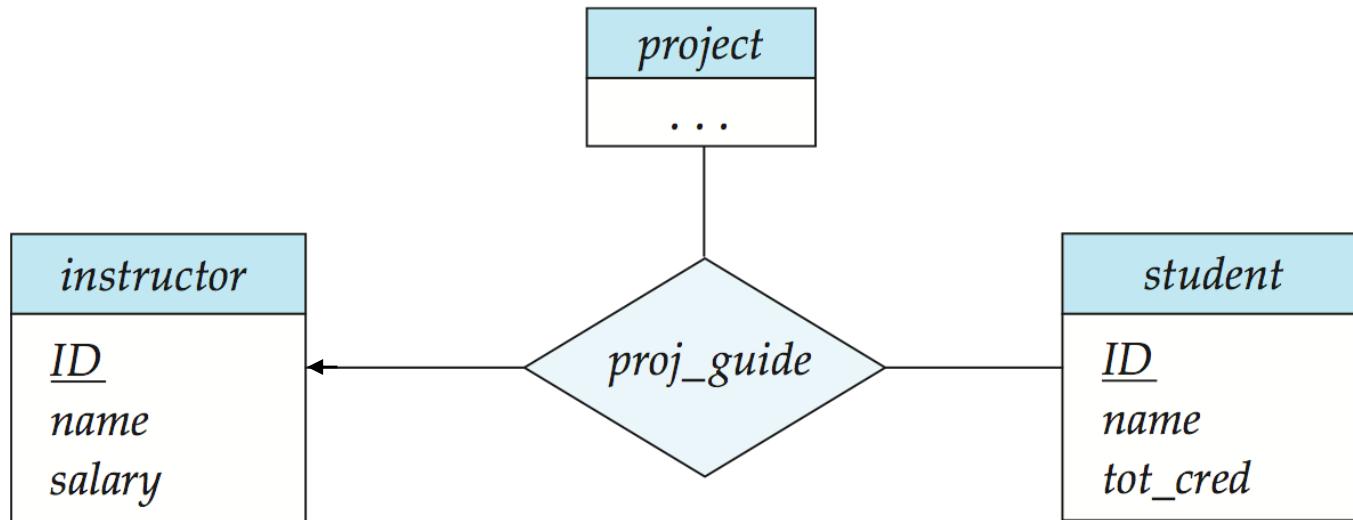
■ Partial Participation

- Some entity elements may not participate in any relationship. This is shown by a **single line**.



1. An instructor might not be advisor; A student might not have an advisor.
2. An instructor might not be advisor; Every student must have an advisor.
3. Every instructor must be advisor; A student might not have an advisor.
4. Every instructor must be advisor; Every student must have an advisor.

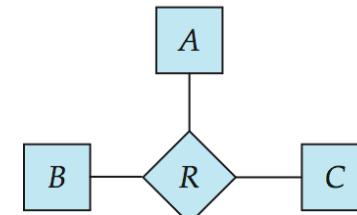
E-R Diagram with a Ternary Relationship Set



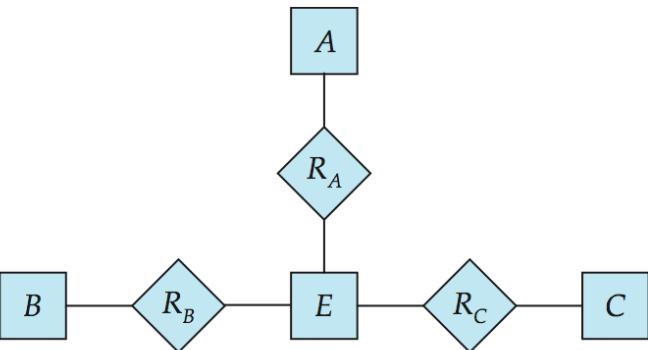
- A **ternary relationship (set)** is a relationship (set) between three entities (entity sets).
- We allow **at most one arrow** out of a ternary (or greater degree) relationship set. E.g. an arrow from *proj_guide* to *instructor* indicates each student has at most one guide for a given project.
- If there is more than one arrow, then there are several ways of defining the meaning. To avoid confusion at most one arrow is allowed.
- To avoid handling the complexities of a ternary relationship set it can be substituted by three binary relationship sets!

Convert Non-Binary Relationships to Binary

- In general, any non-binary relationship can be represented using binary relationships by creating an artificial entity set.
 - Replace R between the entity sets A , B and C by an entity set E and three relationship sets:
 - R_A , relating E and A
 - R_B , relating E and B
 - R_C , relating E and C
 - Create a special identifying attribute for E
 - Add any attributes of R to E
 - For each relationship (a_i, b_i, c_i) in R , create:
 - A new entity e_i in the entity set E
 - Add (e_i, a_i) to R_A
 - Add (e_i, b_i) to R_B
 - Add (e_i, c_i) to R_C



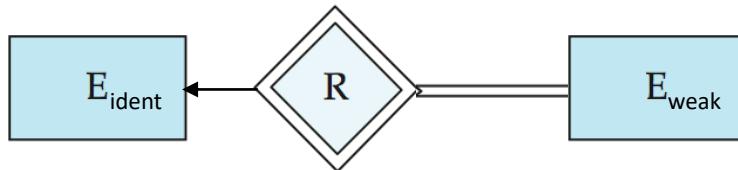
(a)



(b)

Strong and Weak Entity Sets

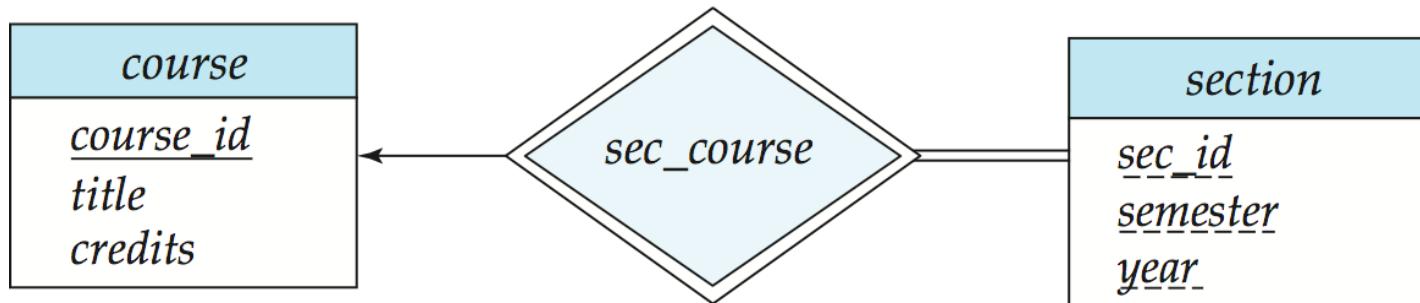
- A strong entity set
 - is one that can form its primary key from its own attributes.
- A weak entity set
 - is one that cannot form its primary key from its own attributes alone.
- Identifying entity set
 - The existence of a weak entity set E_{weak} depends on the existence of an *identifying entity set* E_{ident} . There must be *an identifying relationship set* R between E_{ident} and E_{weak} : a one-to-many relationship set R from E_{ident} to E_{weak} , with total participation of E_{weak} . Identifying relationship sets are shown in double diamonds:



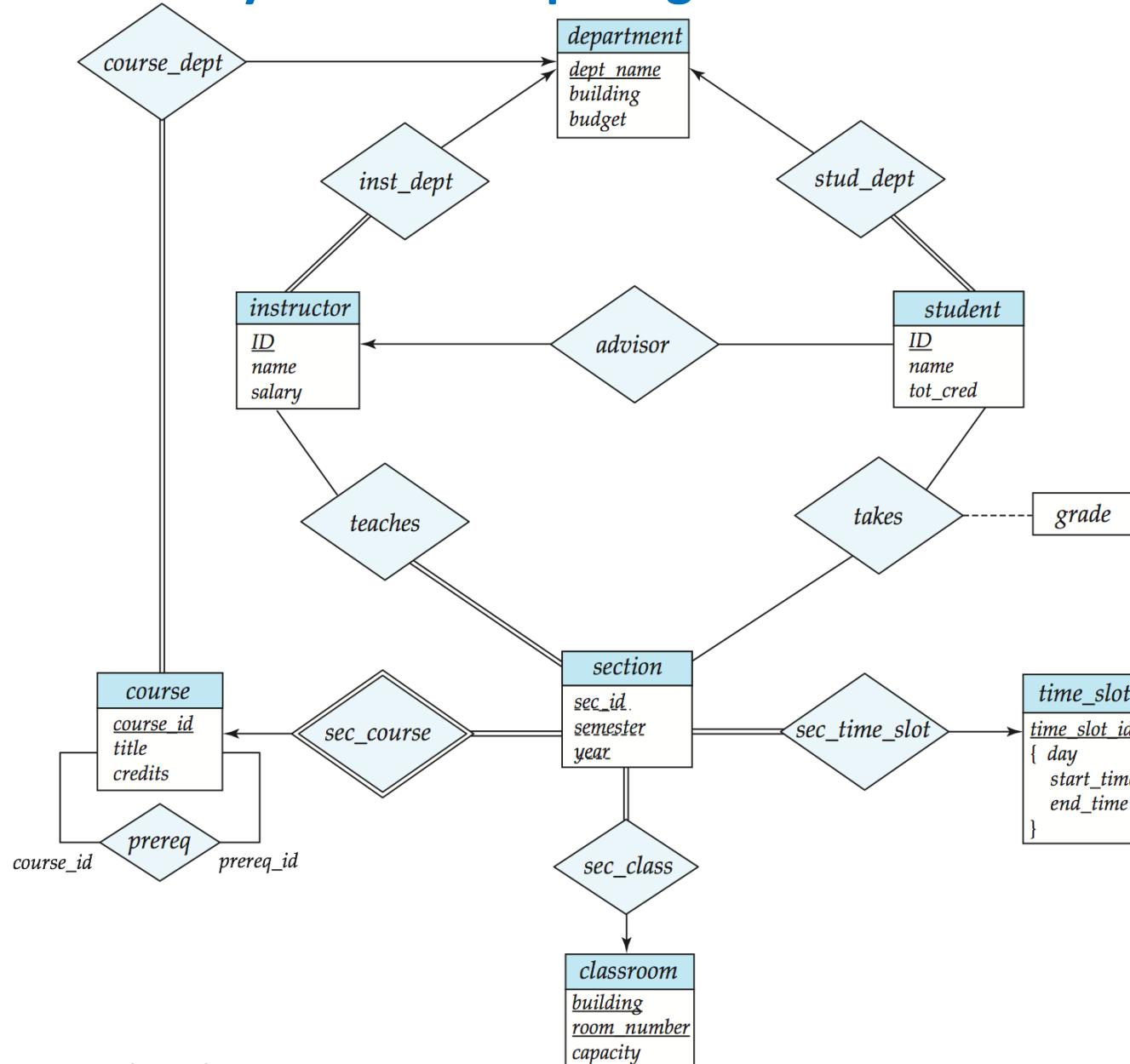
- Discriminator or partial key of E_{weak}
 - is a (sub)set of its attributes that can be used to distinguish an entity that depends on a particular strong entity.
- Primary key of E_{weak}
 - is formed by the primary key of E_{ident} plus the partial key of E_{weak} .

Weak Entity Sets - Example

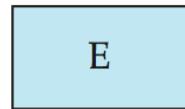
- *section* is a **weak entity set** having *course* as its **identifying entity set** and *sec_course* as its **identifying relationship set**.
- The **partial key** of a weak entity (set) is underlined with a dashed line.
- The **identifying relationship set** of a weak entity set is encapsulated in a double diamond.
- **Primary key** for section consists of: *course_id*, *sec_id*, *semester*, *year*



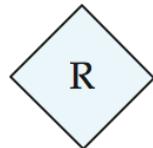
Entity-Relationship Diagram for the University Database



Summary of Textbook Adapted UML Notation



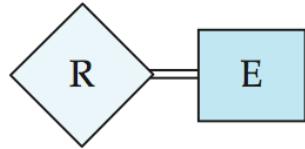
Entity Set



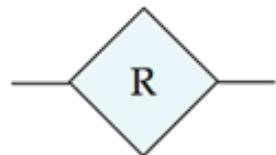
Relationship Set



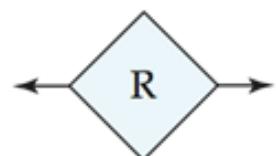
Identifying
Relationship Set of a
Weak Entity Set



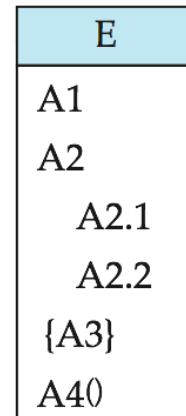
Total Participation
of Entity Set in the
Relationship Set



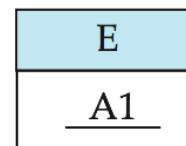
many-to-many
Relationship Set



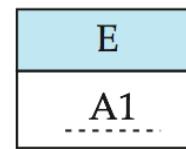
one-to-one
Relationship Set



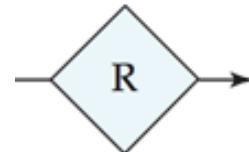
Entity Set with Attributes:
A1 Simple
A2 Composite
A3 Multivalued
A4 Derived



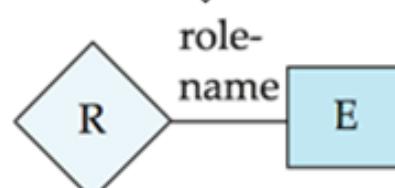
Primary Key Attribute



Discriminating Attribute
of a Weak Entity



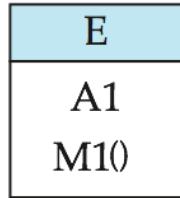
many-to-one
Relationship



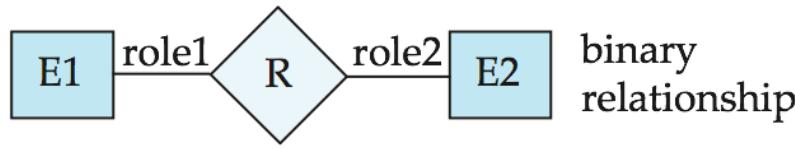
Role Indicator

Other Diagram Notations: ER vs. UML Class Diagrams

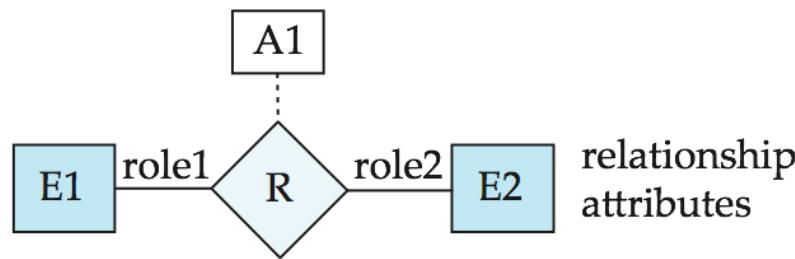
ER Diagram Notation



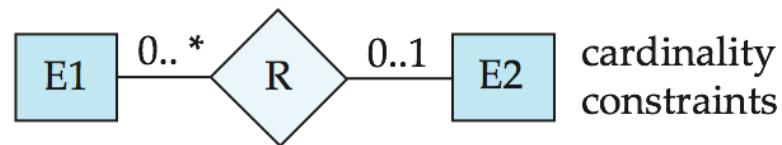
entity with attributes (simple, composite, multivalued, derived)



binary relationship

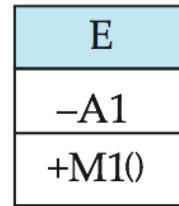


relationship attributes

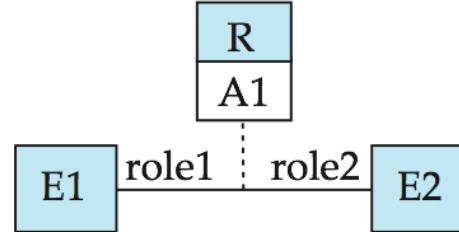


cardinality constraints

Equivalent in UML



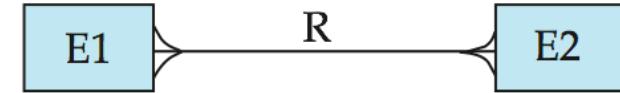
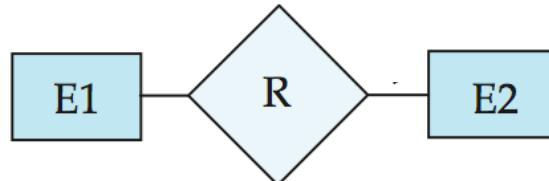
class with simple attributes and methods (attribute prefixes: + = public, - = private, # = protected)



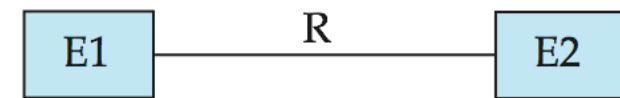
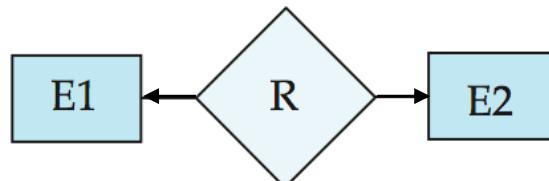
Note reversal of position in cardinality constraint depiction

Other Diagram Notations: ER vs. Crows Feet Notation

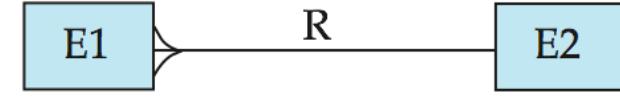
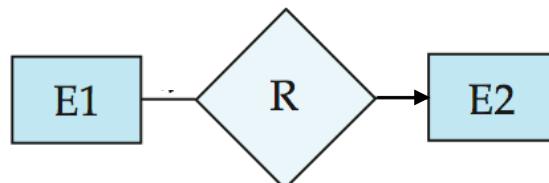
many-to-many
relationship



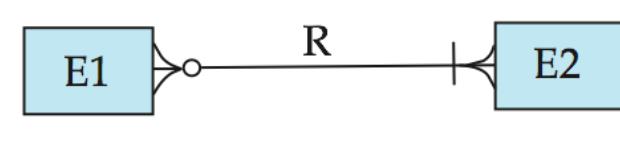
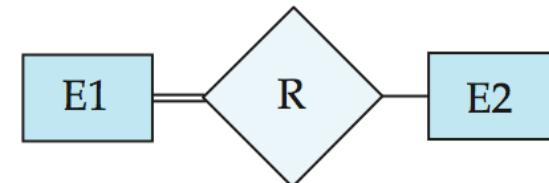
one-to-one
relationship



many-to-one
relationship



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

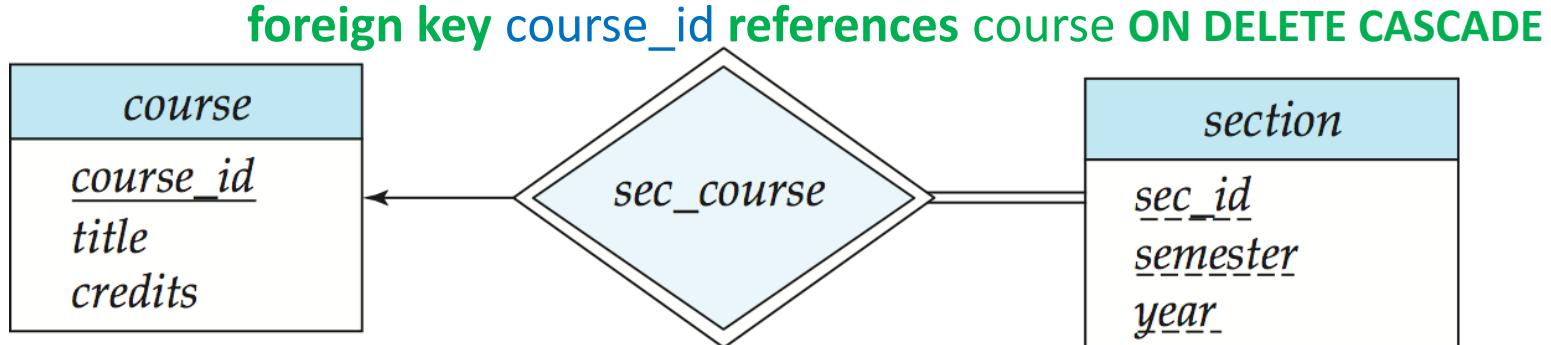


Converting E-R Diagrams to Relation Schemas

- Convert entity sets to relation schemas
 - special treatment of complex attributes
- Convert relationship sets to relation schemas

Convert Entity Sets to Relation Schemas

- First convert strong entity sets, then weak entity sets. If a weak entity set E_1 has another weak entity set E_2 as identifying entity set, then E_2 should be converted before E_1 .
 - A strong entity set (with atomic attributes)
 - Becomes a relation schema with the same attributes and primary key
 - Example: *course(course_id, title, credits)*
 - A weak entity set (with atomic attributes)
 - Becomes a relation schema that additionally includes the primary key of the identifying entity set as a foreign key attribute.
 - Example: *section(course_id, sec_id, semester, year)*



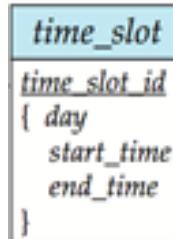
Conversion of Complex Attributes

- **Composite attributes**
 - Example: *name, address*
 - Fix: Store leaf attributes like *street_name* only
- **Multivalued attributes A**
 - Example: $\{ \text{phone_number} \}$, a set of phone numbers.
 - Fix: Store it instead in a separate relation together with the primary key:
 $\text{phones}(\underline{ID}, \underline{\text{phone_number}})$
foreign key *ID* references *instructor*
- **Derived attributes**
 - Example: *age()*, which can be computed from *date_of_birth*.
 - Fix: Store only *date_of_birth*, not *age* – it can be calculated from *date_of_birth* when needed.
- **Conversion of the *instructor* entity set results hence in:**
 $\text{instructor}(\underline{ID}, \underline{\text{first_name}}, \dots, \underline{\text{street_number}}, \dots, \underline{\text{zip}}, \underline{\text{date_of_birth}}),$
 $\text{phones}(\underline{ID}, \underline{\text{phone_number}})$

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

Clean Up: Remove Unnecessary Schemas

Example:



- Conversion of entity set results in:
 - *time_slot1(time slot id)*
 - *time_slot2(time slot id, day, start time, end time)*
- This is first simplified to:
 - *time_slot(time slot id, day, start time, end time)*
- And then to:
 - *time_slot(time slot id, day, start time, end time)*
as a timeslot with the same day and start time can't have different end times

Always remember not to follow conversion rules blindly – remember to clean up.

Convert Binary Relationship Sets to Relation Schemas

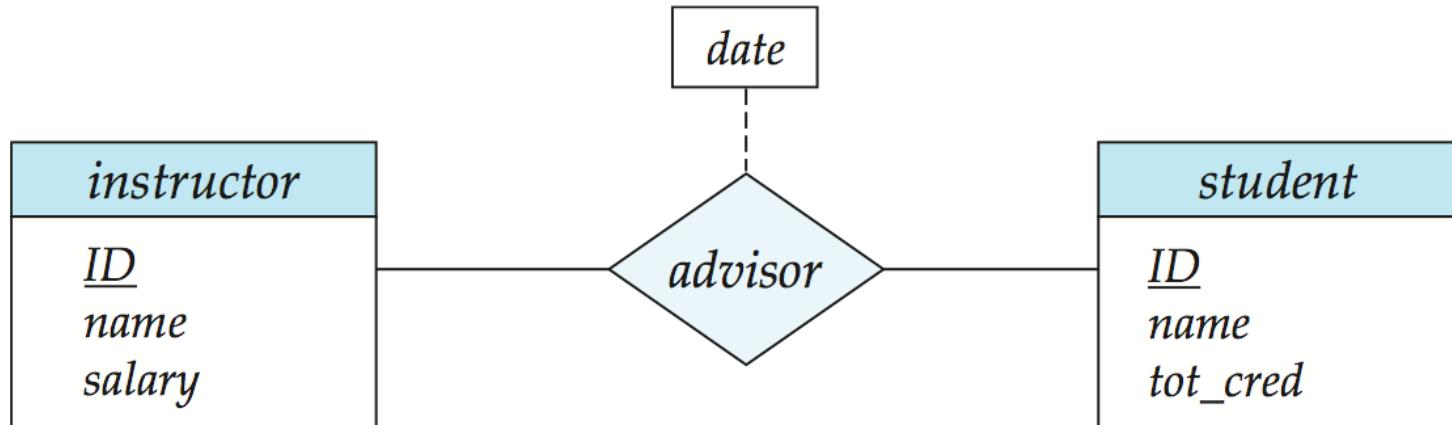
The conversion depends on the cardinality of the relationship set:

- Many-to-Many
- Many-to-One
- One-to-One

Conversion only needed for relationship sets which are not identifying relationship sets **for a weak entity set**.

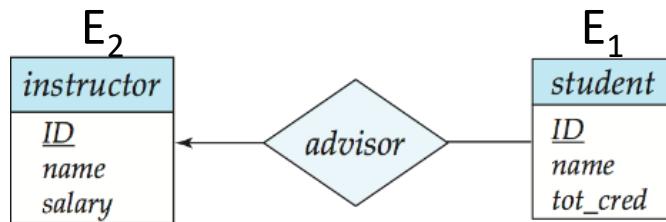
Convert Relationship Sets to Relation Schemas

- Many-to-Many relationship set R between E_1 and E_2
 - Is converted to a relation schema R with **attributes** from the primary keys of E_1 and E_2 and with any attributes of the relationship set.
 - The **primary key** of R is the union of the primary keys of E_1 and E_2 (and in very, very rare cases some of the relationship attributes of R , cf. slide 12).
 - The primary keys of E_1 and E_2 become **foreign keys** in R referencing E_1 and E_2 , respectively.
 - Example: Relation Schema for the relationship *advisor*
 $advisor(\underline{student.ID}, \underline{instructor.ID}, date)$



Convert Relationship Sets to Relation Schemas

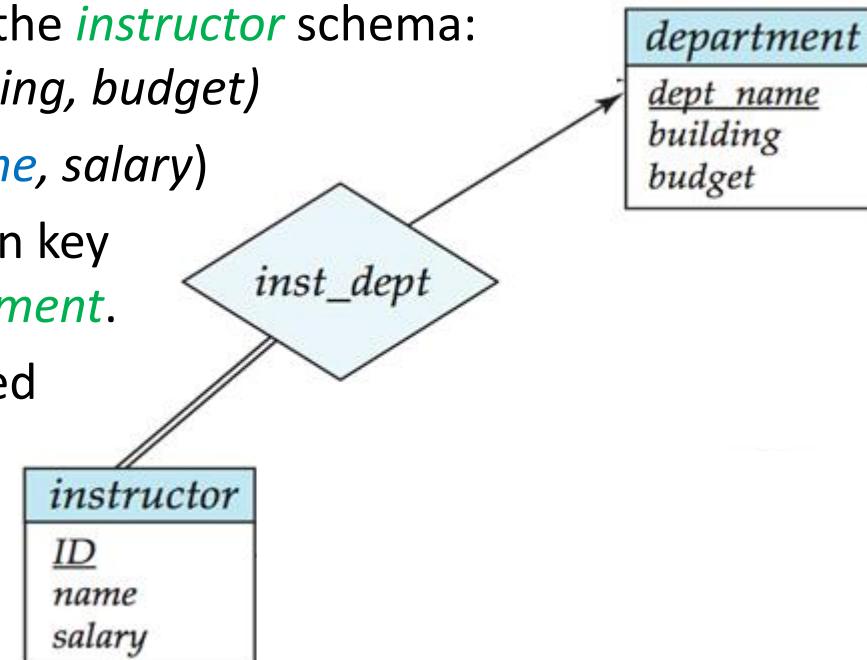
- **Many-to-One relationship sets R between E_1 and E_2**
 - **Approach 1:** Can be converted to a relation schema R in the same way as for Many-to-Many relationship sets R, but the primary key of the resulting relation schema is different: it is the primary key of E_1 .
 - **Approach 2:** It is possible to avoid making the new schema R, by instead modifying the schema of E_1 . This is explained on the following page. (For simplicity, it is assumed that the primary key of relationship set R does not include relationship attributes.)
 - **Approach 2 is usually preferred** unless E_1 has partial participation with only few entities in E_1 taking part in R, as in:



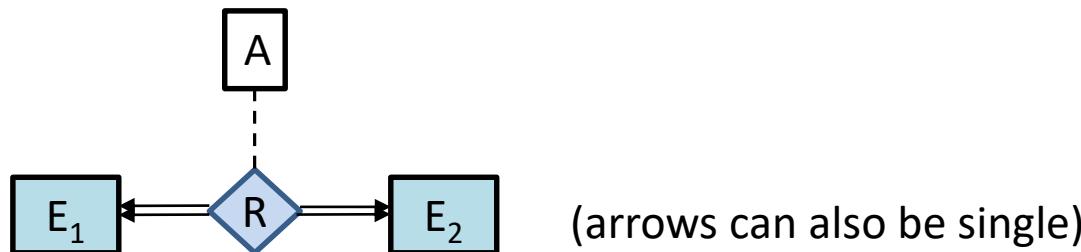
- **One-to-One relationship sets R between E_1 and E_2**
 - Similar comment. Approach 2 is preferred unless both both E_1 and E_2 have partial participation with only few entities taking part in R.

Convert Relationship Sets to Relation Schemas

- Many-to-One relationship sets R from E_1 to E_2
 - Add the attributes A of R and primary key K_2 of the E_2 (one-side) schema to the E_1 (many-side) schema. Hence, K_2 becomes a foreign key in E_1 :
 - $E_1(K_1, \dots, A, K_2), E_2(K_2, \dots)$
 - Example: Instead of creating a schema for the relationship *inst_dept*, add an attribute *dept_name* to the *instructor* schema:
department(dept_name, building, budget)
instructor(ID, name, dept_name, salary)
 - instructor.dept_name* is a foreign key of *instructor* referencing *department*.
 - Total participation can be defined by a "Not Null" constraint for *instructor.dept_name*.



Convert Relationship Sets to Relation Schemas



One-to-One relationship sets R between E_1 and E_2

- The primary key from one of the entity schemas can together with the attributes A of R be added to the other entity schema as a foreign key. This gives two possible solutions:
 - $E_1 (K_1, \dots, A, K_2), E_2 (K_2, \dots)$
 - $E_1 (K_1, \dots), E_2 (K_2, \dots, A, K_1)$
- Choose the solution, which gives the least number of Null Values:
 - For solution 1 it holds that: only if participation of E_1 is partial, then for some rows in E_1 , K_2 will be Null.
 - For solution 2 it holds that: only if participation of E_2 is partial, then for some rows in E_2 , K_1 will be Null.
 - So if e.g. the participation of E_1 is total and E_2 is partial, then go for solution 1.



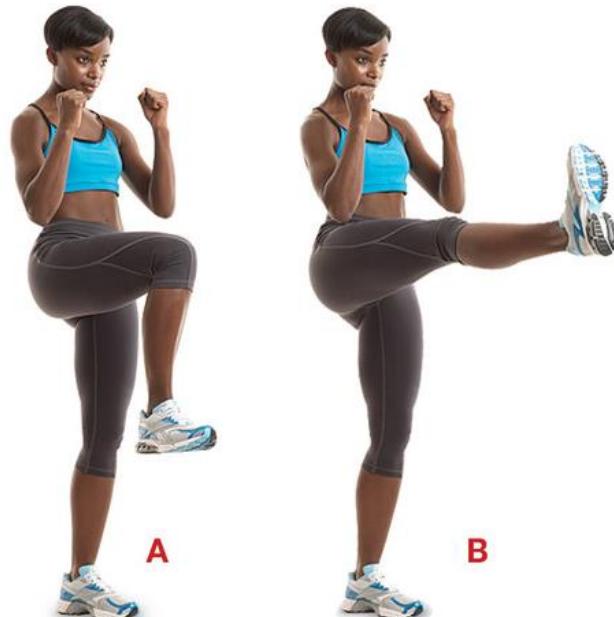
Summary

- E-R Modelling with E-R Diagrams
- Summary of Textbook Adapted UML Notation
- Converting E-R Diagrams to Relation Schemas
- Other diagram notations

Readings

- In Database Systems Concepts please read Chapter 7
- Read the Chapter 7 Summary once again!

Demo Exercises



Demo Exercises clarify ideas and concepts from the Lecture to provide you with good Database Skills.

Do the Demo Exercises.
Solutions are found on later slides.

Super Key, Candidate Key & Primary Key

6.1 The Key concept

Explain the terms *super key*, *candidate key* and *primary key* for entities and relationships.

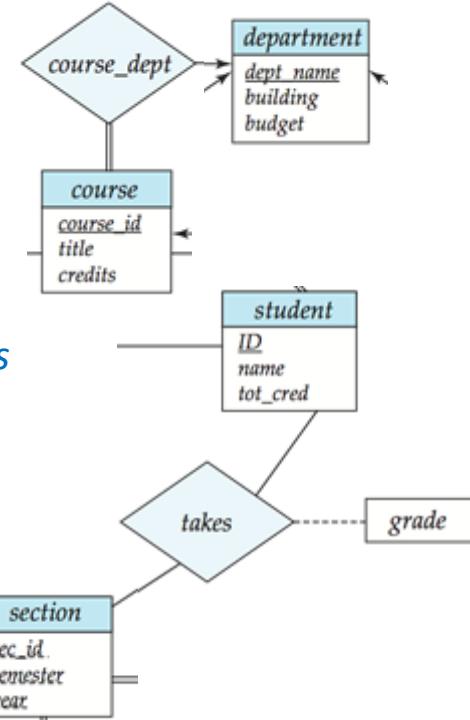
Entity-Relationship Diagram

6.2 Reading an E-R Diagram

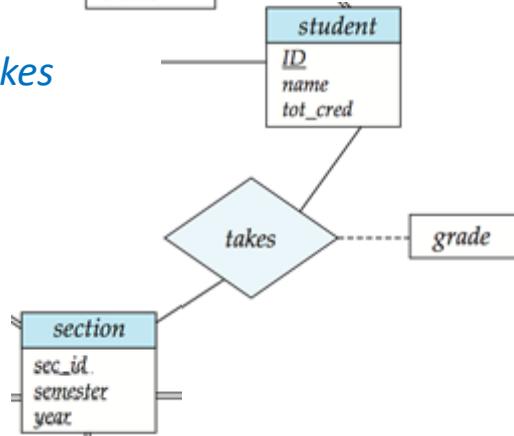
For the University E-R Diagram

a) Explain

course_dept

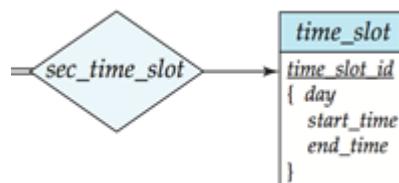


b) Explain *takes*



c)

Explain why the *time_slot* attribute *{day, start_time, end_time}* is multivalued.



E-R Diagrams and Relation Schemas

6.3 From E-R Diagrams to Relation Schemas

An entity set E1 has the attributes A (primary key), B and C. Another entity set E2 has the attributes D (primary key), E and F. A relationship set R with cardinality one-to-many exists between E1 and E2. Both E1 and E2 have partial participation in R.

- a) Draw an E-R Diagram
- b) Convert the E-R Diagram to Relation Schemas
- c) R is now given an attribute G.
Show the updated E-R Diagram and the updated Relation Schemas.

Diagram, Schemas and Instance

6.4 Draw E-R Diagram, Write Relation Schemas, Compose Database Instance

The manager of the Car Repair Shop stated that he has cars registered by license number, owner and model, on which he can provide various services registered by service number, type and price per hour. He tracks all repairs by license number, service number, delivery date and hours spent. The same car can have the same service several times (at different delivery dates).

- a) Construct an Entity-Relationship Diagram and specify the cardinality of the repairs relationship set.
- b) Convert the diagram to Relation Schemas and state any foreign keys.
- c) Construct a Database Instance.

Solutions to Demo Exercises



Super Key, Candidate Key & Primary Key

6.1 The Key concept

Explain the terms *super key*, *candidate key* and *primary key* for entities and relationships.

A **super key** is a set of one or more attributes that, taken collectively, identifies uniquely an entity in the entity set, or a relationship in a relationship set. A super key may contain extraneous attributes. If K is a super key, then so is any superset of K .

A **candidate key** is a super key, for which no proper subset exists (i.e. a minimal super key). It is possible that several distinct sets of attributes could serve as candidate keys.

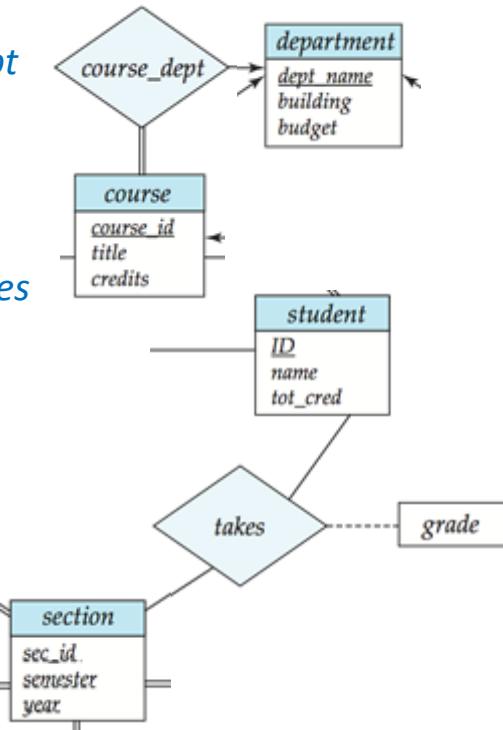
A **primary key** is one of the candidate keys that is chosen by the database designer as the principal means of identifying elements in an entity set or associations in a relationship set.

Entity-Relationship Diagram

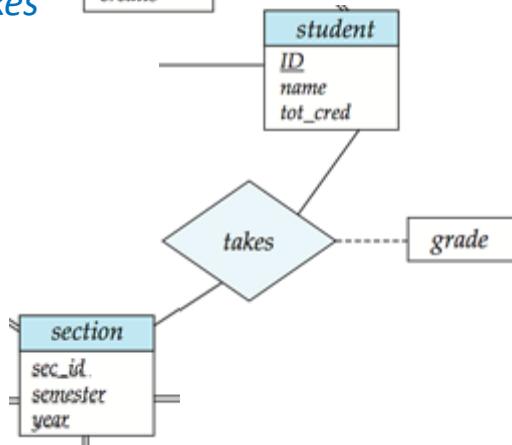
6.2 Reading an E-R Diagram

For the University E-R Diagram

- a) Explain
course_dept

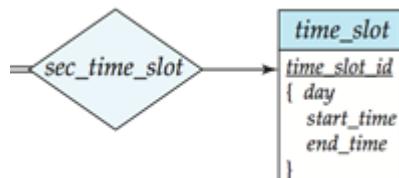


- b) Explain *takes*



c)

Explain why the *time_slot* attribute *{day, start_time, end_time}* is multivalued.



a) *course_dept* defines relationships between courses and departments. It has a **many-to-one** cardinality (shown by an arrowhead towards department), and the participation of course is **total** (is shown by a double line), while the participation of department is **partial** (is shown by a single line). This together means that a *course* is always offered by exactly one *department*, and a *department* can offer from zero to several *courses*.

b) *takes* defines relationships between students and sections. It has a **many-to-many** cardinality (as there are no arrowheads), and the participation is **partial** on both sides. This together means that a *student* may take from zero to several *sections*, and that a *section* may be taken by zero to several students. *takes* has **relationship attribute** *grade* shown by a dotted line.

c) A *time_slot_id* should define a set of weekly meetings (each given by the day, start and end time), for instance Monday, 08.00-10.00 and Thursday, 13.00-15.00. As we need a *set* of meeting times, the attribute is multivalued.

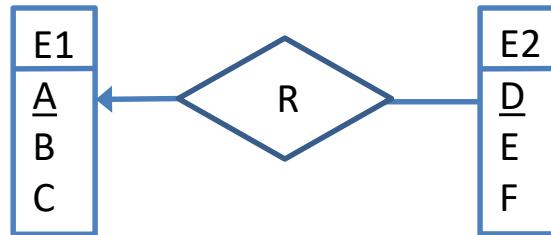
E-R Diagrams and Relation Schemas

6.3 From E-R Diagrams to Relation Schemas

An entity set E1 has the attributes A (primary key), B and C. Another entity set E2 has the attributes D (primary key), E and F. A relationship set R with cardinality one-to-many exists between E1 and E2. Both E1 and E2 have partial participation in R.

- a) Draw an E-R Diagram
- b) Convert the E-R Diagram to Relation Schemas
- c) R is now given an attribute G.
Show the updated E-R Diagram and the updated Relation Schemas.

E-R Diagram



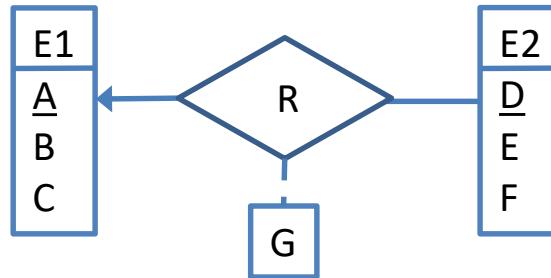
Relation Schemas

R1(A, B, C)

R2(D, E, F, A)

Relationship R is stored in R2 i.e. the 'many' side by adding A as a foreign key.

Updated E-R Diagram



Updated Relation Schemas

R1(A, B, C)

R2(D, E, F, A, G)

The R attribute(s) G are stored in R2.

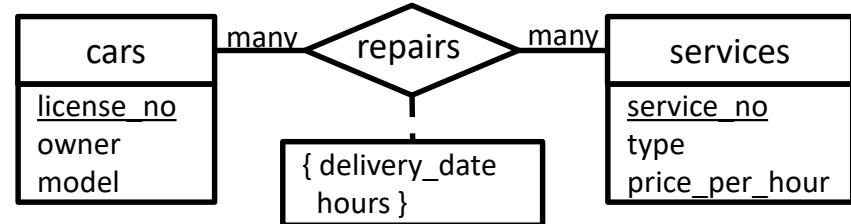
Diagram, Schemas and Instance

6.4 Draw E-R Diagram, Write Relation Schemas, Compose Database Instance

The manager of the Car Repair Shop stated that he has cars registered by license number, owner and model, on which he can provide various services registered by service number, type and price per hour. He tracks all repairs by license number, service number, delivery date and hours spent. The same car can have the same service several times (at different delivery dates).

- Construct an Entity-Relationship Diagram and specify the cardinality of the repairs relationship set.
- Convert the diagram to Relation Schemas and state any foreign keys.
- Construct a Database Instance.

Entity-Relationship Diagram



Relation Schemas

`cars(license_no, owner, model)`
`services(service_no, type, price_per_hour)`
`repairs(license_no, service_no, delivery_date, hours)`
repairs has foreign keys license_no referencing cars and service_no referencing services.

Database Instance

cars	license_no	owner	model
	ZY43816	Adam Asimov	Honda Accord 2,0 Aut
	YE26036	Brian Balter	Audi A4 Limousine 2,0
	UZ58368	Brian Balter	Alfa Romeo 159 2,1

services	service_no	type	price_per_hour
	431001	Bodywork	315,50
	431002	Painting	345,50
	451001	Parts replacement	216,50
	451002	Parts repair	389,50

repairs	license_no	service_no	delivery_date	hours
	ZY43816	431001	2015-02-17	2,5
	ZY43816	431002	2015-02-17	1,5
	UZ58368	51001	2015-02-18	0,5

Exercises



Please answer all exercises
to demonstrate your
Database Skills.

Pencil and Paper Exercises
Solutions are available at 11:45

Diagrams, Schemas and Instance

6.5 Draw Diagram and Write Schemas

An entity set E1 has the attributes A (primary key), B and C. Another entity set E2 has the attributes D (primary key), E and F.

A relationship set R with cardinality many-to-many exists between E1 and E2. Both E1 and E2 have partial participation in R.

a) Draw an E-R Diagram.

b) Convert the E-R Diagram to Relation Schemas and state any foreign key constraints.

c) The relationship is given an attribute G. Show the updated E-R Diagram and the updated Relation Schemas.

6.6 Sets, Diagram, Schemas and Instance

A car insurance company wants a database to

- (1) track clients with name, address and age,
- (2) track cars, with license plate number, model, color and production year, and
- (3) track the car ownerships of clients, each with a start date. It is assumed that a car is owned by exactly one client.

a) List Entity Sets and Relationship Sets.

b) Make an E-R Diagram of the Car Insurance Company.

c) Convert the E-R Diagram to Relation Schemas and state any foreign key constraints.

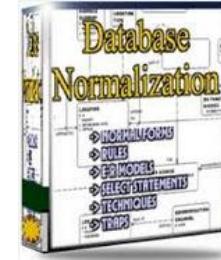
d) Make an example of a Database Instance.

Diagrams, Schemas and Instance

6.7 Draw Diagram and Write Schemas

A company needs help keeping track of the transactions through some bank's ATMs. They have requested a simple database to help with this. An ATM has an ID, a Location and an Associated Bank. A transaction has a transaction number and an amount and is identified by the ID from ATM and the transaction number.

- a) Make an E-R Diagram describing the company.
- b) Convert the E-R Diagram to Relation Schemas and state any foreign key constraints.
- c) Draw the associated database schema diagram.



Normalization

a database schema design technique to
minimize data redundancy and avoid modification anomalies

Chapter 8 of the Textbook

©Anne Haxthausen and Flemming Schmidt

These slides have been prepared by Anne Haxthausen, partly reusing/modifying slides by Flemming Schmidt. Some examples come from Silberschatz, Korth, Sudarshan, 2010.



Contents

- About Normalization
- Theoretical Foundations I: Functional Dependencies
- Normal Forms 1NF-3NF: Original Definitions
- Normal Forms 2NF-3NF, BCNF: General Definitions
- Higher Normal Forms: 4NF, 5NF, ...

About Normalization

- *Normalization* is a simple, practical technique used to minimize data redundancy and avoid modification anomalies.
 - **What:** Normalization involves *decomposing a table into smaller tables* without losing information *and by defining foreign keys*.
 - **The objective** is to isolate data so that additions, updates and deletions can be made in just one place, and then the changes can be propagated through the rest of the database using the defined foreign keys. The **goal** is both to ensure *data consistency* and to *avoid extensive searches*.

Redundancy and Modification Anomalies

- Redundancy is the Evil of Databases
 - *Redundancy* means that the same data is stored in more than one place.
 - **The problem with redundancy is the risk of *modification anomalies*:** i.e. when data are modified (with SQL INSERT, UPDATE and DELETE commands), there is a risk that the data are not modified everywhere making the database *inconsistent*.
 - **The problem with an inconsistent database** is that it might return *different answers* to the same question asked in different ways.
 - If the database is redundancy free, then the DBMS can modify the data efficiently, without having to search the entire database.

Redundancy and Modification Anomalies

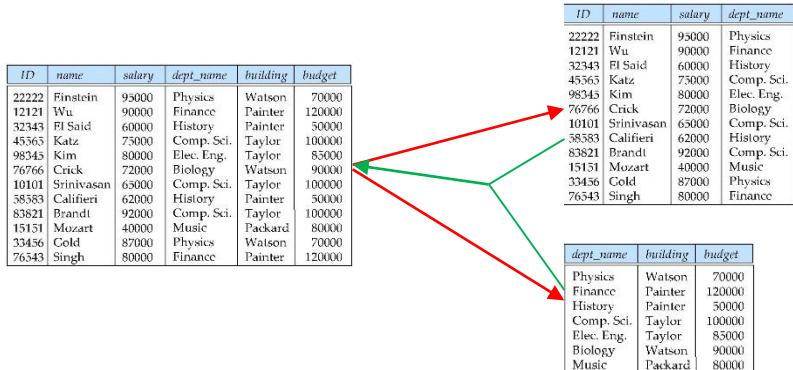
- Suppose we combine *instructor* and *department*:

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

- This design has two problems:
 - It has **redundant information**: *building* and *budget* are repeated for each instructor working in the same department. This can give rise to **modification anomalies** where the data becomes **inconsistent**.
 - It is not possible to store information about a department unless at least one instructor works in the department.

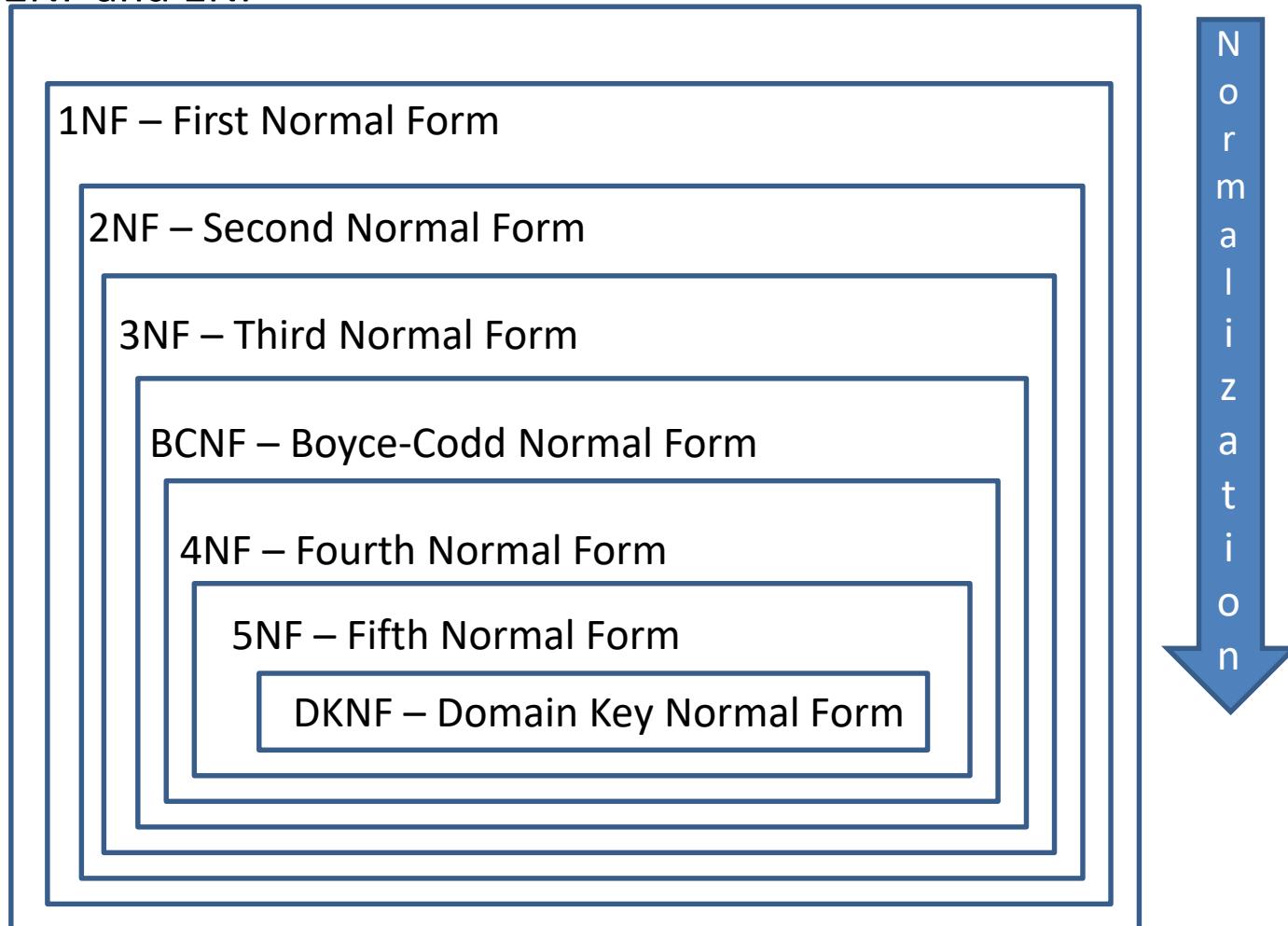
Features of Good Relational Design

- To avoid **modification anomalies**, additional restrictions can be imposed on tables/relation schemas.
- Tables/relation schemas with such restrictions are said to **be in a given normal form** like “Third Normal Form” or 3NF.
- **Normalization** is the process of bringing tables and their relation schemas to higher normal forms by avoiding redundancy. Normally it is done by **projections** of a table into two or more, smaller tables.
- Normalization is **information preserving**: it provides **data lossless decompositions** (usually projections), and is fully reversible, normally by **joining** the normalized tables back into the original table.



Normal Form Hierarchy

- If a table/relation schema is in e.g. 3NF, then by definition it is also in 2NF and 1NF



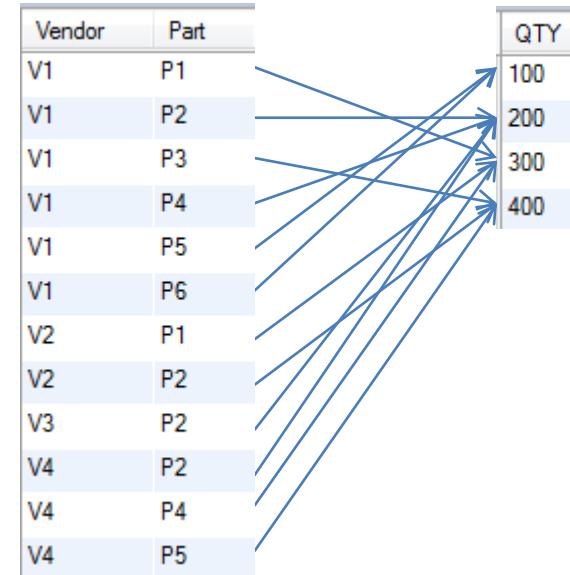
Theoretical Foundations: Functional Dependencies

- Mathematics needed to define 1NF, 2NF, 3NF and BCNF
 - *Functional dependencies*: attribute associations within a relation (schema).
 - Trivial and nontrivial functional dependencies
 - Keys defined in terms of functional dependencies
 - Armstrong's axioms and derived theorems: to find *derived functional dependencies*
- Mathematics needed to define 4NF (later)
 - *Multivalued dependencies*.

Functional Dependencies

- Describe dependencies between attribute sets A and B of a relation schema R:
 - Basically *a many-to-one relation* of associations from one set A of attributes to another set B of attributes within a given relation.
- Example: Consider **Shipments(Vendor, Part, Qty)**:

Vendor	Part	Qty
V1	P1	300
V1	P2	200
V1	P3	400
V1	P4	200
V1	P5	100
V1	P6	100
V2	P1	300
V2	P2	400
V3	P2	200
V4	P2	200
V4	P4	300
V4	P5	400



$\{\text{Vendor}, \text{Part}\} \rightarrow \{\text{Qty}\}$ is *a functional dependency holding for Shipments*.

The attribute set $\{\text{Vendor}, \text{Part}\}$ *functionally determines* the attribute set $\{\text{Qty}\}$.

The attribute set $\{\text{Qty}\}$ is *functionally dependent* of the attribute set $\{\text{Vendor}, \text{Part}\}$.

Formal Definition of Functional Dependency

Let X and Y be subsets of the set of attributes of a relation schema R .

- A **functional dependency** $X \rightarrow Y$ **holds** for R if and only if
 - in every legal instance of R , each X value has associated precisely one Y value (i.e. whenever two rows have the same X value, they also have the same Y values).
- When $X \rightarrow Y$ **holds** for R , we say
 - Y is *functionally dependent of X* and X *functionally determines Y*
 - X is said to be the *determinant* and Y the *dependent*.
 - If $X = \{A_1, \dots, A_n\}$, $Y = \{B_1, \dots, B_m\}$ we sometimes write $A_1, \dots, A_n \rightarrow B_1, \dots, B_m$ or $A_1 \dots A_n \rightarrow B_1 \dots B_m$
- **Trivial Dependencies**
 - $X \rightarrow Y$ is *trivial*, if $Y \subseteq X$.
 - Example 1: $\{Vendor, Part\} \rightarrow \{Vendor\}$
 - Example 2: $\{Part\} \rightarrow \{Part\}$
- **Nontrivial Dependencies**
 - Those FDs which are not trivial.
 - Nontrivial FDs lead to definitions of *integrity constraints* like keys.

Functional Dependencies

- Many functional dependencies can be proposed.
- Example: `Shipment1(Vendor, City, Part, Qty)` (where a vendor only can be in one city)

Vendor	City	Part	Qty
V1	London	P1	100
V1	London	P2	100
V2	Paris	P1	200
V2	Paris	P2	200
V3	Paris	P2	300
V4	London	P2	400
V4	London	P4	400
V4	London	P5	400

No	Determinant	FD	Dependent	Validity	Remark
1	{Vendor, Part}	→	{Qty}	Legal	
2	{Vendor}	→	{City}	Legal	
3	{Qty}	→	{Vendor}	Illegal	
4	{Part}	→	{Qty}	Illegal	
5	{Vendor, Part}	→	{City}	Legal	Derived(2)
6	{Vendor}	→	{Vendor}	Legal	trivial
256		→			

- To decide whether a FD is *valid* (legal for all relation instances), one has to consider the real world.
- Of a set of 4 attributes, it is possible to make $2^4 = 16$ subsets.
- Combining 16 determinants with 16 dependents gives 256 potential FDs.
- Of the 256 potential FDs, some are valid and some are not.
- (Canonical) Cover Set:** A (irreducible/minimal) set of valid functional dependencies from which all valid functional dependencies can be determined. In the example: $\{ \{Vendor, Part\} \rightarrow \{Qty\}, \{Vendor\} \rightarrow \{City\} \}$ is a canonical cover set.
- Closure Set F^+ of a set F of functional dependencies:** The set of all valid functional dependencies that can be logically derived from the F .

Super Key and Functional Dependencies

Let R be a relation schema and let K be a subset of the set A_R of attributes of R .

- Super key, original definition:
 - K is a *superkey* of R , if, in any legal instance of R , for all pairs t_1 and t_2 of tuples in the instance of R if $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$. Hence, a specific K value will uniquely define a specific tuple in R .
- Super key defined by functional dependencies:
 - K is a *superkey* of R , if $K \rightarrow X$ holds for every attribute X of R .
 - Super Key examples for `Shipment1(Vendor, City, Part, Qty)`
 - `{Vendor, City, Part}`
Knowing the values of `Vendor`, `City` and `Part`, the value of `Qty` is defined.
 - However, of interest is a Super Key with a minimum set of attributes:
`{Vendor, Part}`
Knowing the values of `Vendor` and `Part`, the values of `City` and `Qty` is defined.

Candidate Key and Functional Dependencies

- Candidate key is a minimal super key:
 - K is a *candidate key* for R if and only if,
 $K \rightarrow A_R$ and for no $\alpha \subset K$, $\alpha \rightarrow A_R$
 - Candidate key example for `Shipment1(Vendor, City, Part, Qty)`:
 $\{Vendor, Part\}$
 - In some cases R can have several candidate keys!
- Primary key
 - A candidate key is selected by the DBA to be the *primary key* for a relation.

Armstrong's Rules

Let R be a relation schema and let F be a set of functional dependencies on R .

■ Armstrong's Rules

- are some Axioms and Derived Theorems for deriving functional dependences from F
- are
 - **sound**: only valid FDs are derived from valid FDs.
 - **complete**: all valid FDs (the closure set of F) can be derived from a cover set F .
- can e.g. be used for finding candidate keys

Armstrong's Axioms and Derived Theorems

Let X, Y, Z and V be sets of attributes.

■ Armstrong's axioms

1. Reflexivity: If Y is a subset of X , then $X \rightarrow Y$
2. Augmentation: If $X \rightarrow Y$, then $XZ \rightarrow YZ$

Note: XZ is used as a shorthand for $X \cup Z$

Note: $XY = YX$ and $XX = X$; Sets have no order and no repetitions

3. Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

■ Derived theorems

4. Self-determination: $X \rightarrow X$
5. Decomposition: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
6. Union: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
7. Composition: If $X \rightarrow Y$ and $Z \rightarrow V$, then $XZ \rightarrow YV$
8. General Unification: If $X \rightarrow Y$ and $Z \rightarrow V$, then $X \cup (Z - Y) \rightarrow YV$
(where 'u' is Set Union and '-' is Set Difference)

Using Armstrong's Rules

■ To propose candidate keys

- Given $R(A, B, C, D, E)$ with functional dependencies: $A \rightarrow BC$, $CD \rightarrow E$, $B \rightarrow D$, $E \rightarrow A$
 1. $A \rightarrow A$ Rule 4 Self-determination
 2. $A \rightarrow B$ Rule 5 Decomposition of given $A \rightarrow BC$
 3. $A \rightarrow C$ Rule 5 Decomposition of given $A \rightarrow BC$
 4. $A \rightarrow D$ Rule 3 Transitivity of $A \rightarrow B$ and given $B \rightarrow D$
 5. $A \rightarrow CD$ Rule 6 Union $A \rightarrow CD$
 6. $A \rightarrow E$ Rule 3 Transitivity of $A \rightarrow CD$ and given $CD \rightarrow E$
 7. $A \rightarrow ABCDE$ Rule 6 Union
 8. $E \rightarrow ABCDE$ Rule 3 Transitivity of given $E \rightarrow A$ and $A \rightarrow ABCDE$
 9. Candidate Keys: Both A and E! Select one for the Primary Key!
 10. Relation Schema: $R(\underline{A}, B, C, D, E)$ if A is selected as Primary Key or $R(\underline{E}, A, B, C, D)$ if E is selected as Primary key

Normal Forms 1NF-3NF: Original Definitions

- **2NF** and **3NF** are defined using the notion of *functional dependencies*.
The starting point for the definitions is:
 - a relational schema R
 - a cover set FD of functional dependencies for R
- **The original definitions** (here) assume the tables have one candidate key which has been chosen as *primary key*, in the following called the *key*.
- **The generalized definitions** of **2NF** and **3NF** (in the book) generalizes the original definitions by taking *several candidate keys* into account.
- When there is only one candidate key, the original and the generalized definitions are equivalent.

Normal Forms Defined Informally

- 1st normal form
 - All attributes depend on **the key**.
- 2nd normal form
 - All attributes depend on **the whole key**
- 3rd normal form
 - All attributes depend on **nothing but the key**

1NF – First Normal Form

- For a relation to be in First Normal Form 1NF
 - Each attribute value must be a single value (is atomic, not multivalued or composite).

OrderNo	ItemNo	Qty
1000	100	18
1000	102	13
1000	105	24
1001	102	24
1002	100	14
1002	105	21

Normalization to 1NF

- *OrdersTable* below is not in 1NF
as the values of the attribute ItemNo are not atomic.

OrdersTable(OrderNo, ItemNo)

OrderNo	ItemNo
1000	100,102,105
1001	102
1002	100,105

- Normalization to 1NF

Orders1NF(OrderNo, ItemNo)

OrderNo	ItemNo
1000	100
1000	102
1000	105
1001	102
1002	100
1002	105

2NF – Second Normal Form

- For a relation schema R to be in Second Normal Form 2NF
 1. It must be in 1NF.
 2. Each non primary key attribute A must not depend on a strict subset K_{part} of the primary key attribute set K , i.e. $K_{part} \rightarrow A$ and $K_{part} \subset K$ must not be the case. A must depend on the entire primary key K .
- Some special cases where a 1NF relation schema is in 2NF:
 - The primary key consists only of one attribute.
 - The primary key consists of all attributes in the relation.
 - Each non primary key attribute depend on all the attributes of the primary key.
- Normalisation 1NF to 2NF:
 - Move the set of all attributes A which depend on a $K_{part} \subset K$ to a new relation $R2$ together with a copy of K_{part} , which becomes the primary key as well as a foreign key. If there is only one such attribute A we have:
$$R(\underline{K}, \dots, A) \rightarrow R1(\underline{K}, \dots) \text{ foreign key } K_{part} \text{ references } R2, R2(\underline{K}_{part}, A)$$
Key of $R1$: is $K \setminus K_{part}$, if $K \setminus K_{part} \rightarrow K_{part}$, otherwise it is K .
 - Repeat the step above, if $R1$ or $R2$ are not yet in 2 NF.
 - Decomposition is *data lossless*:
$$(\text{select } K, \dots \text{ from } R) \text{ natural join } (\text{select } K_{part}, A \text{ from } R) = \text{select } * \text{ from } R$$

Normalization to 2NF - Example

- Orders1NF is in 1NF, but not 2NF:

OrderNo	ItemNo	ItemName
1000	100	Car Nitromethan
1000	102	Airplane Spitfire
1000	105	Battleship Bismarck
1001	102	Airplane Spitfire
1002	100	Car Nitromethan
1002	105	Battleship Bismarck

- Orders1NF(OrderNo, ItemNo, ItemName)
with FD: ItemNo \rightarrow ItemName
- ItemName depends on ItemNo only, and not on the full primary key. Violation of rule 2 for 2NF.

- Normalization to (Orders2NF and Items in) 2NF

- Orders2NF(OrderNo, ItemNo) foreign Key(ItemNo) references Items(ItemNo)
- Items(ItemNo, ItemName)
- Note that ItemNo is included in the key of Orders2NF, as OrderNo $\text{-/}\rightarrow$ ItemNo

OrderNo	ItemNo	ItemNo	Itemname
1000	100	100	Car Nitromethan
1000	102	102	Airplane Spitfire
1000	105	105	Battleship Bismarck
1001	102		
1002	100		
1002	105		

- Associated normalization of tables is projections:
 $\text{Orders2NF} \equiv \Pi_{\text{OrderNo}, \text{ItemNo}}(\text{Orders1NF})$
 $\text{Items} \equiv \Pi_{\text{ItemNo}, \text{ItemName}}(\text{Orders1NF})$
- A Natural Join brings back the table:
 $\text{Orders1NF} \equiv \text{Orders2NF} \bowtie \text{Items}$
- Normalization is information preserving .

Normalization to 2NF - Example

Teachers in 1NF

InstID	DeptName	InstName	Salary	Building	Budget
10101	Comp. Sci.	Srinivasan	65000.00	Taylor	100000.00
11001	Comp. Sci.	Valdez	36000.00	Taylor	100000.00
11002	Comp. Sci.	Koerver	36000.00	Taylor	100000.00
12121	Finance	Wu	90000.00	Painter	120000.00
15151	Music	Mozart	40000.00	Packard	80000.00
22222	Physics	Einstein	95000.00	Watson	70000.00
32343	History	El Said	60000.00	Painter	50000.00

- Teachers(InstID, DeptName, InstName, Salary, Building, Budget)
- InstID \rightarrow InstName, DeptName, Salary, DeptName \rightarrow Building, Budget
- Violation of 2NF: Building, Budget depend on DeptName only, and not on the full primary key.

Normalizing *Teachers* to *Instructor* and *Department* in 2NF

- Instructor(InstID, InstName, DeptName, Salary)
Foreign Key(DeptName) references Department(DeptName)
- Department(DeptName, Building, Budget)
- Note that DeptName is **not** included in the key of Instructor as InstID \rightarrow DeptName.

InstID	InstName	DeptName	Salary	DeptName	Building	Budget
10101	Srinivasan	Comp. Sci.	65000.00	Biology	Watson	90000.00
11001	Valdez	Comp. Sci.	36000.00	Comp. Sci.	Taylor	100000.00
11002	Koerver	Comp. Sci.	36000.00	Elec. Eng.	Taylor	85000.00
12121	Wu	Finance	90000.00	Finance	Painter	120000.00
15151	Mozart	Music	40000.00	History	Painter	50000.00
22222	Einstein	Physics	95000.00			
32343	El Said	History	60000.00			
33456	Gold	Physics	87000.00			
45565	Katz	Comp. Sci.	75000.00			

- Associate normalization of tables is projections:
 $\text{Instructor} \equiv \prod_{InstID, InstName, DeptName, Salary, ItemName} (\text{Teachers})$
 $\text{Department} \equiv \prod_{DeptName, Building, Budget} (\text{Teachers})$
- A Natural Join brings back the table:
 $\text{Teachers} \equiv \text{Instructor} \bowtie \text{Department}$
- Normalization is information preserving .

3NF – Third Normal Form

- For a relation schema $R(\underline{K}, \dots, A)$ to be in Third Normal Form 3NF
 1. It must be in 2NF.
 2. Each non primary key attribute A must depend ***directly*** on the entire primary key \underline{K} . It must not depend ***transitively*** via other attributes B , like $\underline{K} \rightarrow B \rightarrow A$, where $B \not\rightarrow \underline{K}$ and $B \rightarrow A$ is non-trivial.
- Normalization 2NF to 3NF
 - Move attributes A which are transitively dependent on \underline{K} via B , i.e. $\underline{K} \rightarrow B \rightarrow A$ for some attribute set B , to a new relation $R2$ together with a copy of the dependent attributes in B , which become the primary key and constitute a foreign key:
 $R(\underline{K}, \dots, B, A) \xrightarrow{\hspace{1cm}} R1(\underline{K}, \dots, B)$ foreign key B references $R2$, $R2(B, A)$
 - Repeat the step above, if $R1$ or $R2$ are not yet in 3 NF.
 - Decomposition is ***data lossless***:
 $(\text{select } \underline{K}, \dots, B \text{ from } R) \text{ natural join } (\text{select } B, A \text{ from } R) = \text{select } * \text{ from } R$

Normalization to 3NF - Example

■ Customer2NF in 2NF

CustomerNo	PostNo	CityName
10500	2500	Valby
10501	2500	Valby
10502	2600	Glostrup
10503	2500	Valby
10504	2605	Brøndby
10505	2600	Glostrup

- $\text{Customers2NF}(\underline{\text{CustomerNo}}, \text{PostNo}, \text{CityName})$
- $\text{CustomerNo} \rightarrow \text{PostNo}$, $\text{PostNo} \rightarrow \text{CityName}$
- CityName depends on the full Primary Key, but transitively via PostNo . Violation of rule 2 for 3NF.

■ Normalization to 3NF

CustomerNo	PostNo	PostNo	CityName
10500	2500	2500	Valby
10501	2500	2600	Glostrup
10502	2600	2605	Brøndby
10503	2500		
10504	2605		
10505	2600		

- $\text{Customers3NF}(\underline{\text{CustomerNo}}, \text{PostNo})$,
foreign key(PostNo) references $\text{Post}(\text{PostNo})$
- $\text{Post}(\underline{\text{PostNo}}, \text{CityName})$
- Associated normalization of tables is projections:
 $\text{Customer3NF} \equiv \prod_{\text{CustomerNo}, \text{PostNo}} (\text{Customer2NF})$
 $\text{Post} \equiv \prod_{\text{PostNo}, \text{CityName}} (\text{Customer2NF})$
- A Natural Join brings back the table:
 $\text{Customer2NF} \equiv \text{Customer3NF} \bowtie \text{Post}$
- Normalization is information preserving.

Normal Forms 2NF-3NF, BCNF: General Definitions

- Are defined using the notion of *functional dependencies*.
- **The original definitions** (on previous slides) of 2NF-3NF assume the tables have one candidate key which has been chosen as *primary key*.
- **The generalized definitions** of 2NF-3NF and BCNF (in the book) take *several candidate keys* into account.
- When there is only one candidate key, the original and the generalized definitions of 2NF-3NF are equivalent.
- When there is only one candidate key: 3NF and BCNF are the same.
- 3NF and BCNF are the same (according to Date) unless
 - there are several composite candidate keys CK1 and CK2
 - which are overlapping ($CK1 \cap CK2 \neq \{\}$)

This exception is very rare.

2NF-3NF General Definitions

- For a relation schema R to be in Second Normal Form 2NF
 1. It must be in 1NF.
 2. Each non **candidate** key attribute A must not depend on a strict subset K_{part} of **any candidate** key attribute set K , i.e. $K_{part} \rightarrow A$ and $K_{part} \subset K$ must not be the case. A must depend on the entire K .
- For a relation schema $R(K, \dots, A)$ to be in Third Normal Form 3NF
 1. It must be in 2NF.
 2. No non **candidate** key attribute A depends **transitively** on any **candidate** key K via other attributes B , like $K \rightarrow B \rightarrow A$, where $B \not\rightarrow K$ and $B \rightarrow A$ is non-trivial.

In the book there is another general definition of 3NF. The two definitions are equivalent.

BCNF - Boyce-Codd Normal Form

Boyce-Codd Normal Form BCNF

- A relation schema is in Boyce-Codd Normal Form BCNF, if and only if, every nontrivial, left-irreducible FD $\alpha \rightarrow \beta$ has a candidate key as its determinant α .
 - *Left-irreducible* means that there is no proper subset s of α such that $s \rightarrow \beta$.
- Normalization of $R(A_1, \dots, A_n, B_1, \dots, B_n, C)$ to BCNF:
- Assume $B_1, \dots, B_n \rightarrow C$ is nontrivial, left-irreducible, but B_1, \dots, B_n is not a candidate key.
 - $R(A_1, \dots, A_n, B_1, \dots, B_n, C) \xrightarrow{\hspace{1cm}}$
 $R_1(A_1, \dots, A_n, B_1, \dots, B_n)$ foreign key B_1, \dots, B_n references R_2 and $R_2(B_1, \dots, B_n, C)$
 - Repeat the step above, if $R1$ or $R2$ are not yet in BCNF.
- Normalization to BCNF, example:
- $R(A, B, C)$ with $FD = \{A \rightarrow B, B \rightarrow C\}$
is not in BCNF ($B \rightarrow C$, but B is not candidate key)
 - Decomposition of R : $R_1(A, B)$ and $R_2(B, C)$.

BCNF versus 3NF - Example

Given functional dependencies

$\text{Pizza, ToppingType} \rightarrow \text{Topping}$

$\text{Topping} \rightarrow \text{ToppingType}$

on $R(\text{Pizza, ToppingType, Topping})$

Then R has two candidate keys:

- $\text{Pizza, ToppingType}$
- Pizza, Topping

The first is chosen as primary key.

Then

$\underline{R(\text{Pizza, Topping, ToppingType})}$

is in 3NF, but not in BCNF as $\text{Topping} \rightarrow \text{ToppingType}$ and Topping is not a candidate key.

Pizza	Topping	ToppingType
1	mozzarella	cheese
1	pepperoni	meat
1	olives	vegetable
2	mozzarella	cheese
2	sausage	meat
2	peppers	vegetable

Higher Normal Forms: 4NF, 5NF, ...

- Are defined using the notion of *multivalued dependencies*.
- **Fourth Normal Form**

A relation schema with *multiple unrelated multivalued dependencies* must be broken into relation schemas that separate the unrelated attributes.
- **Fifth Normal Form** neither covered in the book nor here.

Multivalued Dependencies

Let R be a relation schema and α and β be disjoint subsets of the attributes of R and let γ be the remaining attributes of R .

- A *multivalued dependency* $\alpha \twoheadrightarrow \beta$ **holds** for R if and only if, in every legal instance of R , the set of β values matching a given $\alpha \gamma$ value pair depends only on the α value and is independent of the γ value.
- When $\alpha \twoheadrightarrow \beta$ we say α *multivalued determines* β .
- When $\alpha \twoheadrightarrow \beta$ **holds** for R , then $\alpha \twoheadrightarrow \gamma$ also **holds** for R .
- When $\alpha \rightarrow \beta$ **holds** for R , then $\alpha \twoheadrightarrow \beta$ also **holds** for R .

	α	β	γ
t_1	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_r$
t_2	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_r$
t_3	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_r$
t_4	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_r$

Multivalued Dependencies - Example

■ Example

- CTX(Course, Teacher, Text)

Course →→ Teacher ‘Physics’ →→ ‘Prof. Green’ and ‘Prof. Brown’

Course →→ Text ‘Physics’ →→ ‘Principles of Optics’ and ‘Basic Mechanics’

Course	Teacher	Text
Math	Prof. Green	Basic Mechanics
Math	Prof. Green	Trigonometry
Math	Prof. Green	Vector Analysis
Physics	Prof. Brown	Basic Mechanics
Physics	Prof. Brown	Principles of Optics
Physics	Prof. Green	Basic Mechanics
Physics	Prof. Green	Principles of Optics

4NF - Fourth Normal Form

■ Fourth Normal Form 4NF

- A relation schema R is in **Fourth Normal Form 4NF**, if and only if, whenever a non-trivially $\alpha \rightarrow\rightarrow \beta$ holds, then α is a key of R (all attributes of R are also functionally dependent on α).
 $\alpha \rightarrow\rightarrow \beta$ is **trivial** means $\beta \subseteq \alpha$ or $\alpha \cup \beta = \text{set of all attributes in } R$.

■ Normalization from BCNF to 4NF

- Assume $\alpha \rightarrow\rightarrow \beta$ non-trivially holds and α is NOT a key of R .
- This usually arises from many-to-many relationship sets or multivalued entity sets.
- $R(\alpha, \beta, \gamma) \xrightarrow{\hspace{1cm}} R1(\alpha, \beta), R2(\alpha, \gamma)$

■ Decomposition is **data lossless**:

$$(\text{select } \alpha, \beta \text{ from } R) \text{ natural join } (\text{select } \alpha, \gamma \text{ from } R) = \text{select } * \text{ from } R$$

Multivalued Dependencies and 4NF

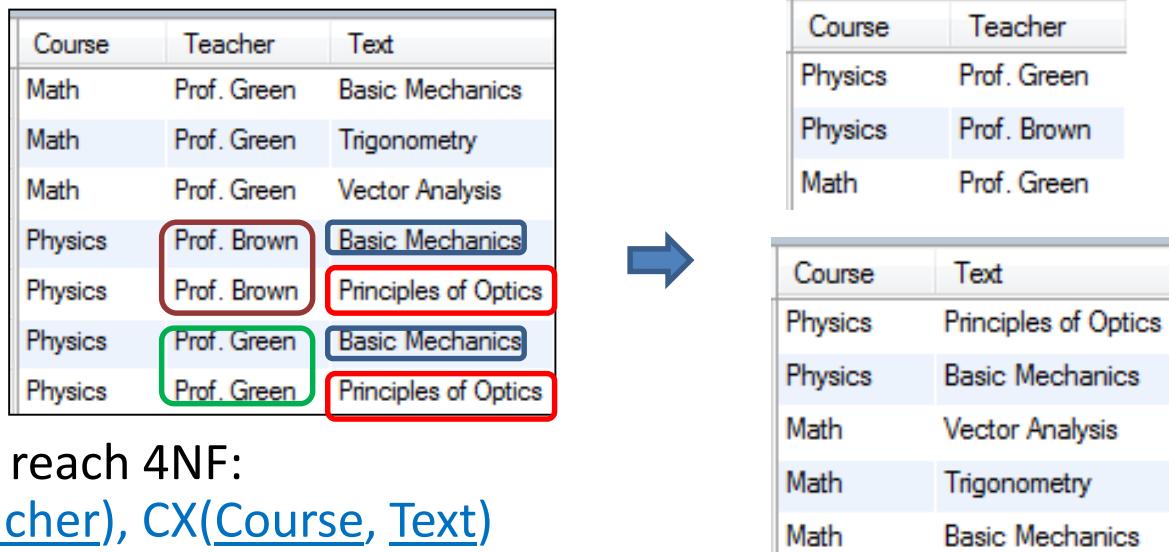
Example

- $\text{CTX}(\text{Course}, \text{Teacher}, \text{Text})$

$\text{Course} \rightarrow\rightarrow \text{Teacher}$ 'Physics' $\rightarrow\rightarrow$ 'Prof. Green' and 'Prof. Brown'

$\text{Course} \rightarrow\rightarrow \text{Text}$ 'Physics' $\rightarrow\rightarrow$ 'Principles of Optics' and 'Basic Mechanics'

- CTX is in 3NF, but not in 4NF as Course is not a key!



- Decompose to reach 4NF:

$\text{CT}(\text{Course}, \text{Teacher})$, $\text{CX}(\text{Course}, \text{Text})$

These are in 4NF, as $\text{Course} \rightarrow\rightarrow \text{Teacher}$ resp. $\text{Course} \rightarrow\rightarrow \text{Text}$ now are trivial

- A Natural Join of CT and CX over Course will restore CTX

Summary

- Redundancy is the evil of all databases leading to modification anomalies (i.e. problems with SQL INSERT, UPDATE and DELETE).
- A good relational design avoids redundancy by decomposition of tables into multiple tables without redundancy.
- Decomposition of tables are done by data lossless projections and a natural join of the decomposed tables will bring back the original table.
- The decomposition process is called normalization, and the tables and their relation schemas gradually continues to improve in quality in higher and higher normal forms.
- **Supplementary Literature:**
An Introduction to Database Systems, C.J. Date, Addison-Wesley, Eight Edition, 2004, Chapters 11, 12 & 13.

Demo Exercises



Demo Exercises clarify ideas and concepts from the Lecture to provide you with good Database Skills.

Discuss and do the Demo Exercises.

Functional Dependencies

7.1.1 Number of Functional Dependencies

How many functional dependencies can be proposed for the relation schema $R(A, B, C)$?

7.1.2 Candidate Keys with Armstrong's Rules

Use Armstrong's rules to make a list of Candidate Keys for the following relation schema $R(A, B, C, D)$ with the following functional dependencies:

$B \rightarrow AD$, $D \rightarrow B$, $A \rightarrow C$. Please specify which of Armstrong's rules are used in each step.
Finally select a Primary Key.

First Normal Form

7.1.3 Violation of First Normal Form 1NF

What is the problem, and how can the table below be normalized?

employee_no	Phone_numbers
67810101	28801633, 28801724
67810210	28801325, 28805647, 28801518
67810324	28801713

Third Normal Form

7.1.4 Violation of Third Normal Form 3NF

What is the problem, and how can the table below be normalized?

Driver_license	License_plate	Full_name	Car_manufacturer
31261435	JW46476	Jens Hansen	Honda
31237248	AX24583	Finn Jensen	Honda
31229753	AZ26184	Anna Flink	Mazda
31237248	KC27534	Finn Jensen	Honda
31231212	JW46538	Karin Holst	Mazda

BCNF

7.1.5 Violation of BCNF

Consider the table

Pizza	Topping	ToppingType
1	mozzarella	cheese
1	pepperoni	meat
1	olives	vegetable
2	mozzarella	cheese
2	sausage	meat
2	peppers	vegetable

7.1.5 Solution:

with relation schema

$R(\underline{\text{Pizza}}, \underline{\text{Topping}}, \underline{\text{ToppingType}})$

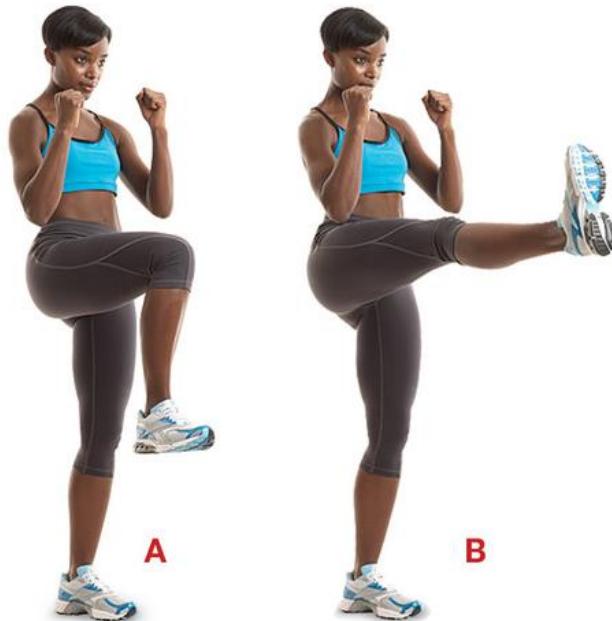
and functional dependencies

$\text{Pizza}, \text{ToppingType} \rightarrow \text{Topping}$

$\text{Topping} \rightarrow \text{ToppingType}$

Normalize the relation schema and table to BCNF.

Solutions to Demo Exercises



Functional Dependencies

7.1.1 Number of Functional Dependencies

How many functional dependencies can be proposed for the relation schema $R(A, B, C)$?

7.1.1 Of a set of 3 attributes, $2^3 = 8$ subsets can be made (i.e. $\{\}, \{A\}, \{B\}, \{C\}, \{A,B\}, \{A,C\}, \{B,C\}$, and $\{A, B, C\}$). Each subset can be a Determinant or Dependent, thus $8 \times 8 = 64$ functional dependencies can be proposed, like $\{A, B\} \rightarrow \{C\}$, $\{A\} \rightarrow \{B\}$.

7.1.2 Candidate Keys with Armstrong's Rules

Use Armstrong's rules to make a list of Candidate Keys for the following relation schema $R(A, B, C, D)$ with the following functional dependencies:

$B \rightarrow AD$, $D \rightarrow B$, $A \rightarrow C$. Please specify which of Armstrong's rules are used in each step.

Finally select a Primary Key.

7.1.2 By using Armstrong's rules:

1. $B \rightarrow B$ by rule 4 Self-determination
 2. $B \rightarrow A$ by rule 5 Decomposition of $B \rightarrow AD$
 3. $B \rightarrow D$ by rule 5 Decomposition of $B \rightarrow AD$
 4. $B \rightarrow C$ given $B \rightarrow A \wedge A \rightarrow C$ by rule 3 Transitivity
 5. $B \rightarrow ABCD$ by rule 6 Union
 6. $D \rightarrow ABCD$ given $D \rightarrow B$ and $B \rightarrow ABCD$ by rule 3
- Candidate Keys are B and D!

I select B to be the Primary Key, but I could have selected D instead.

First Normal Form

7.1.3 Violation of First Normal Form 1NF

What is the problem, and how can the table below be normalized?

employee_no	Phone_numbers
67810101	28801633, 28801724
67810210	28801325, 28805647, 28801518
67810324	28801713

7.1.3 It looks like the table has *employee_no* as primary key.

The normalization could be the one shown below, with a primary key

- $(employee_no, Phone_no)$, if one phone number can be shared between several *employees*.
- $Phone_no$, if a phone number can belong only to one *employee*:

employee_no	Phone_no
67810101	28801633
67810101	28801724
67810210	28801325
67810210	28805647
67810210	28801518
67810324	28801713

As can be seen, there is no limit to the number of phone numbers that an employee can have, and a search of an employee's phone numbers is simple.

Third Normal Form

7.1.4 Violation of Third Normal Form 3NF

What is the problem, and how can the table below be normalized?

Driver_license	License_plate	Full_name	Car_manufacturer
31261435	JW46476	Jens Hansen	Honda
31237248	AX24583	Finn Jensen	Honda
31229753	AZ26184	Anna Flink	Mazda
31237248	KC27534	Finn Jensen	Honda
31231212	JW46538	Karin Holst	Mazda

7.1.4 Inspecting the tables and using domain knowledge we identify the following cover set of functional dependencies:

Driver_license \rightarrow *Full_name*

License_plate \rightarrow *Driver_License*

License_plate \rightarrow *Car_manufacturing*

Using Armstrong's transitivity rule we also have

License_plate \rightarrow *Full_name*

As {*License_plate*} functionally determines all attributes and is minimal, *License_plate* can be chosen as primary key. The Table is in 2NF as all attributes depend fully on the key, but not in 3NF as *Full_name* transitively depends on the key via *Driver_license*. The problem by this is that the information that 'Finn Jensen' has driver license '31237248' is recorded twice (i.e. redundant), and an update of *Driver_license* for 'Finn Jensen' might lead to ambiguity. Normalization gives the two 3NF tables below. The first has *driver_license* and the second has *license_plate* as primary key. A natural join of the two (join over attribute *driver_license*), will bring back the original table.

driver_license	full_name
31229753	Anna Flink
31231212	Karin Holst
31237248	Finn Jensen
31261435	Jens Hansen

license_plate	driver_license	car_manufacturer
AZ26184	31229753	Mazda
JW46538	31231212	Mazda
AX24583	31237248	Honda
KC27534	31237248	Honda
JW46476	31261435	Honda

BCNF

7.1.5 Violation of BCNF

Consider the table

Pizza	Topping	ToppingType
1	mozzarella	cheese
1	pepperoni	meat
1	olives	vegetable
2	mozzarella	cheese
2	sausage	meat
2	pebbers	vegetable

7.1.5 Normalization to BCNF gives tables with schemas R1(Pizza, Topping) and R2(Topping, ToppingType)

Pizza	Topping
1	mozzarella
1	pepperoni
1	olives
2	mozzarella
2	sausage
2	pebbers

with relation schema

R(Pizza, Topping, ToppingType)

and functional dependencies

Pizza, ToppingType → Topping

Topping → ToppingType

Topping	ToppingType
mozzarella	cheese
pepperoni	meat
olives	Vegetable
sausage	meat
pebbers	vegetable

Normalize the relation schema and table to BCNF.

Exercises



Please answer all exercises
to demonstrate your
Database Skills.

Pencil and Paper Exercises
Solutions are available at 11:45

Normalization

7.2.1 Violation of First Normal Form 1NF

Consider the relation schema:

Employees(EmpNo, Jobs)

where Jobs is a multivalued attribute containing one or more jobs.

Make changes needed to ensure 1NF.

7.2.2 Violation of Second Normal Form 2NF

Consider the relation schema:

Clients(ClientNo, SalesRepNo, CName, SName, Date)

where ClientNo and CName is the number and name of a client, SalesRepNo and SName is the number and name of a sales representative, and Date is the date where the sales representative was assigned to the client.

Explain why Clients is not in 2NF. Hint: First decide the minimal, non-trivial functional dependences.

Normalize Clients to 2NF.

7.2.3 Violation of Third Normal Form 3NF

Consider the relation schema:

Winners(Tournament, Year, Winner, Birthday)

where each row in the table tells who was the Winner of a particular Tournament in a particular Year. The Birthday of the Winner is also given.

Explain why Winners is not in 3NF and why that is a problem. Is Winners in 2NF? Normalize Winners to 3NF.

7.2.4 Violation of Fourth Normal Form 4NF

We are told that in a company, an employee can work on many projects and be employed in many departments. Departments as well as projects can have many employees.

Consider the relation schema:

Company(EmpNo, DepNo, ProjectNo)

with the dependency:

$\text{EmpNo} \rightarrow \rightarrow \text{ProjectNo}$

Explain why Company is not in 4NF.

Normalize Company to 4NF.

7.2.5 Functional Dependencies

How many functional dependencies can be proposed for the schema: $R(A, B, C, D, E)$?

7.2.6 Candidate keys with Armstrong's rules

Use Armstrong's rules to propose a primary key for the following Relation Schema:

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

with the following set of functional dependencies:

$\{A \rightarrow BC, E \rightarrow FG, AB \rightarrow D, EG \rightarrow H\}$

Please specify which of Armstrong's rules are used in each step.



Technical University of Denmark

DTU Compute

Department of Applied Mathematics and Computer Science

Formal Query Languages

Mathematically defined Query Languages

Section 6 of the textbook

©Flemming Schmidt and Anne Haxthausen

Mathematical Foundations

- **Formal Relational Query Languages**
 - Relational Algebra
 - Domain Calculus
- **Their Mathematical Foundations**
 - Set Theory
 - Algebra
 - Logic



Set Theory



Georg Cantor 1845-1918

■ Set Theory

- In mathematics *set theory* is one of the most fundamental concepts, and its formalization was a major event in the history of mathematics.
- Set theory is a *foundation* for most mathematical disciplines.
- A *set* is a collection of distinct *elements* considered as a whole, and *element membership* of a set is the fundamental concept.
- Sets can be *represented* in different ways:
 - As a list of individual elements: $\{e_1, e_2, e_3\}$
Ordering and repetitions have no importance, e.g.
 $\{e_1, e_2, e_3\} = \{e_2, e_1, e_3\} = \{e_1, e_2, e_2, e_3\}$
 - By a description of element properties: $\{e \in \text{Nat} \mid e < 4 \vee e = 9\}$ ($= \{0, 1, 2, 3, 9\}$) in such a way, that it can be determined whether a given element e is a member of the set or not.
- *Set operations*: Union, Difference, Intersection, Cartesian Product ...

Algebra

Diophantus of Alexandria 200-284 and Muhammad ibn Musa al-Khwarizmi 780-850



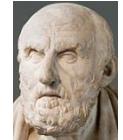
Algebra

- A fundamental discipline in mathematics that arose from the idea that one can perform arithmetic operations on non numerical objects called variables. (Arithmetic: $3+4 = 4+3$; Algebra: $X+Y = Y+X$).
- In its most general form, algebra is the study of mathematical symbols and the rules for manipulating these symbols.

Logic

Aristotle 384–322 BC , ...

Chrysippus of Soli 279-206 BC



- *Propositional Calculus* is a formal system, where formulas are propositions over some variables, like $P(x)$, which for each possible value of x has a truth value of either **True** or **False**. The formulas may contain sub-formulas combined by *logical operators* like \wedge (“and”), \vee (“or”), and \Rightarrow (“implies”).

Example:

$$P(x) \triangleq x > 7 \wedge x < 10;$$

$P(x)$ is **True** for $x = 8$ and for $x = 9$, and **False** for all other values of x .

- *Predicate Calculus* extends propositional calculus by allowing formulas to be quantified with the Existential Quantifier or the Universal Quantifier:
 $\exists x(P(x))$ Reads: There exists some x for which $P(x)$ is True
 $\forall x(P(x))$ Reads: For all x the proposition $P(x)$ is True

Formal Relational Query Languages

- Given a simple Relation Schema and SQL Query

- Relation Schema $R(A, B, C, D)$

SELECT B, A FROM R WHERE $D > 100$



- Formal Relational Query Languages?

- Relational Algebra:** use Relation Variables like R

$\Pi_{B, A}(\sigma_{D > 100}(R))$

- Domain Calculus:** use Domain Variables like a, b, c, d

$\{ \langle b, a \rangle \mid \exists c, d (\langle a, b, c, d \rangle \in R \wedge d > 100) \}$

Relation, Result Attributes
and Selection Predicate!

Relational Algebra

- **Relational Algebra**
 - Was introduced by [Edgar F. Codd](#) while working for IBM in order to provide a procedural query language for the relational model.
 - Is a *procedural language* which provides a selection of operations to generate an answer to a query. Queries are defined in terms of how to compute an answer.
 - Relational Algebra is *the mathematical basis* for the language SQL, and the foundation of Query Processor Engines in some Database Systems.
- **Basic Operations**
 - Is a minimal set of operations.
- **Derived Operations**
 - Additional operations that can be derived from the Basic operations, not for achieving more expressive power, but for expressive convenience.
- **Extended Operations**
 - Give more expressive power - can't be derived from the Basic operations.

Basic Operations in Relational Algebra

- Below R , S are relations, R has attributes A_1, \dots, A_n , $P(A_1, \dots, A_n)$ is a *selection predicate* (a proposition) over A_1, \dots, A_n , and $\{A_i, \dots, A_j\} \subseteq \{A_1, \dots, A_n\}$

	Relational Algebra	Meaning: a relation consisting of
Selection	 $\sigma_{P(A_1, \dots, A_n)}(R)$	Those tuples t of R , for which $P(t)$ is True: $\{ t \mid t \in R \wedge P(t) \}$
Projection	 $\Pi_{A_i, \dots, A_j}(R)$	Delete columns not listed in A_i, \dots, A_j , then delete duplicate tuples: $\{ (A_i, \dots, A_j) \mid (A_1, \dots, A_n) \in R \}$
Set Union	 $R \cup S$	The union of tuples of R and S : $\{ t \mid t \in R \vee t \in S \}$ Only defined when R and S are <i>compatible</i> .
Set Difference	 $R - S$	Those tuples of R that are not in S : $\{ t \mid t \in R \wedge t \notin S \}$ Only defined when R and S are <i>compatible</i> .
Cartesian Product	$R \times S$	All combined tuples: $\{ (r_1, \dots, r_n, s_1, \dots, s_m) \mid (r_1, \dots, r_n) \in R \wedge (s_1, \dots, s_m) \in S \}$
Rename	$\rho_S(R)$ $\rho_{S(L_2)}(R(L_1))$	Rename of R to S . Rename R to S and rename attributes in L_1 to attributes in L_2

- *Compatible* means that R and S must have the same number of attributes with matching, or at least compatible, domains.
- Note: Selection corresponds to SQL's where clause (and not SQL's select clause).

Relational Algebra Basic Operations in SQL

- Below R , S are relations, R has attributes A_1, \dots, A_n , $P(A_1, \dots, A_n)$ is a *selection predicate* (a proposition) over A_1, \dots, A_n , and $\{A_i, \dots, A_j\} \subseteq \{A_1, \dots, A_n\}$

Basic Operations	Relational Algebra	Equivalent SQL Statement
Selection	$\sigma_{P(A_1, \dots, A_n)}(R)$	SELECT * FROM R WHERE P(A ₁ , ..., A _n)
Projection	$\prod_{A_i, \dots, A_j}(R)$	SELECT A _i , ..., A _j FROM R
Set Union	$R \cup S$	(SELECT * FROM R) UNION (SELECT * FROM S)
Set Difference	$R - S$	(SELECT * FROM R) EXCEPT (SELECT * FROM S)
Cartesian Product	$R \times S$	SELECT * FROM R JOIN S
Rename	$\rho_S(R)$ $\rho_{S(B_i, \dots, B_j)}(R(A_i, \dots, A_j))$	SELECT * FROM R AS S SELECT A _i AS B _i , ..., A _j AS B _j FROM R AS S

Basic Operations in Relational Algebra

Using the University Schema



10.1.1 Selection Operation

Find Instructor rows for instructors in the physics department earning more than 90000.

$$\sigma_{\text{DeptName} = 'Physics' \wedge \text{Salary} > 90000}(\text{Instructor})$$

10.1.2 Projection Operation

Find name and IDs of all instructors

$$\Pi_{\text{InstName}, \text{InstID}}(\text{Instructor})$$

10.1.3 Set Union

Find name and IDs of all instructors and students.

$$\Pi_{\text{InstName}, \text{InstID}}(\text{Instructor}) \cup \Pi_{\text{StudName}, \text{StudID}}(\text{Student})$$

10.1.4 Set Difference

Find instructor IDs for instructors that have not taught any courses yet.

$$\Pi_{\text{InstID}}(\text{Instructor}) - \Pi_{\text{InstID}}(\text{Teaches})$$

10.1.5 Cartesian Product

Combine each instructor with each student

Instructor x Student

10.1.6 Rename

Find salary of the instructor with the highest salary. (Use T as a temporary relation).

$$\begin{aligned} & \Pi_{\text{Salary}}(\text{Instructor}) - \\ & \Pi_{\text{Instructor.Salary}} \\ & (\sigma_{\text{Instructor.Salary} < \text{T.Salary}} (\text{Instructor} \times \rho_{\text{T}}(\text{Instructor}))) \end{aligned}$$

Derived Operations in Relational Algebra

- Below R and S are relations, and $\{A_1, \dots, A_n\}$ is the intersection of their attributes.
- Θ is a predicate over attributes in the union of attributes of R and S .

Derived Operations	Relational Algebra	Meaning: a relation consisting of
Natural Join	$R \bowtie S$	Select those tuples from $R \times S$, which have same values for A_1, \dots, A_n , then delete duplicate attributes (A_1, \dots, A_n) from S .
Left Outer Join	$R \bowtie L S$	Adds to $R \bowtie S$: tuples of R that do not match any tuples of S , concatenated with null values for non common attributes in the S relation.
Right Outer Join	$R \bowtie R S$	Adds to $R \bowtie S$: tuples of S that do not match any tuples of R , concatenated with null values for non common attributes in the R relation.
Full Outer Join	$R \bowtie F S$	$(R \bowtie S) \cup (R \bowtie R S)$
Theta join	$R \bowtie_\Theta S$	$\sigma_\Theta(R \times S)$
Set Intersection	$R \cap S$	Tuples that are members of both R and S : $\{ t \mid t \in R \wedge t \in S \}$ Only defined when R and S are compatible.
Assignment	$S \leftarrow R$	Relation variable S is assigned the value of relation R .

Deriving Operations from Basic Operations

- Natural Join can be derived from **Cartesian Product**, **Selection** and **Projection**
 - Schemas $R(A, B, C, D)$, $S(B, D, E)$ and Natural Join Schema $T(A, B, C, D, E)$
 - $T \equiv R \bowtie S \equiv \prod_{R.A, R.B, R.C, R.D, S.E} (\sigma_{R.B=S.B \wedge R.D=S.D} (R \times S))$
- Left Outer Join can be derived from **Cartesian Product**, **Selection**, **Projection**, **Set Difference** and **Set Union**
 - Schemas $R(A, B, C)$, $S(A, C, D, E)$ and Left Outer Join Schema $T(A, B, C, D, E)$
 - $T \equiv R \bowtie L \equiv (R \bowtie S) \cup ((R - \prod_{R.A, R.B, R.C} (R \bowtie S)) \times \{(Null, Null)\})$
(Here $R - \prod_{R.A, R.B, R.C} (R \bowtie S)$ contains those tuples of R that did not match any tuple in S .)
 $R \bowtie S$ can be further decomposed using **Cartesian Product**, **Selection** & **Projection**
- Set Intersection can be derived from **Set Difference**
 - $R \cap S \equiv R - (R - S)$

Relational Algebra Derived Operations in SQL

- Below **R** and **S** are relations.

Derived Operations	Relational Algebra	Equivalent SQL Statement
Natural Join	$R \bowtie S$	SELECT * FROM R NATURAL JOIN S
Left Outer Join	$R \bowtie\! S$	SELECT * FROM R NATURAL LEFT OUTER JOIN S
Right Outer Join	$R \bowtie\! S$	SELECT * FROM R NATURAL RIGHT OUTER JOIN S
Full Outer Join	$R \bowtie\! S$	SELECT * FROM R NATURAL FULL OUTER JOIN S
Set Intersection	$R \cap S$	(SELECT * FROM R) INTERSECT (SELECT * FROM S)
Assignment	$S \leftarrow R$	CREATE TABLE S SELECT * FROM R

Derived Operations in Relational Algebra

10.1.7 Natural Join

Find instructor names and course IDs they have taught. (Join on attribute InstID).

$$\Pi_{\text{InstName, CourseID}}(\text{Instructor} \bowtie \text{Teaches})$$

10.1.8 Natural Join

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

$$\Pi_{\text{InstName, Title}} (\sigma_{\text{DeptName}=\text{'Comp. Sci.'}} (\text{Instructor} \bowtie \text{Teaches} \bowtie \Pi_{\text{CourseID, Title}} (\text{Course})))$$

10.1.9 Left Outer Join

Find student names and their advisor IDs.

$$\Pi_{\text{StudName, InstID}} (\text{Student} \bowtie \text{Advisor})$$

10.1.10 Set Intersection

Find courses that were taught both in Fall 2009 and in Spring 2010.

$$\Pi_{\text{CourseID}} (\sigma_{\text{Semester}=\text{'Fall'}} \wedge \text{StudyYear}=2009 (\text{Section})) \cap \Pi_{\text{CourseID}} (\sigma_{\text{Semester}=\text{'Spring'}} \wedge \text{StudyYear}=2010 (\text{Section}))$$

10.1.11 Assignment

Make a relation called Teacher from Instructor.

Teacher \leftarrow Instructor

Extended Operations in Relational Algebra

Arithmetic Attribute Calculations and Aggregation

Extended Operations	Relational Algebra	Equivalent SQL
Generalized Projection	$\Pi_{E_1, E_2, \dots, E_n}(R)$	SELECT E ₁ , E ₂ , ..., E _n FROM R
Aggregate Functions	AVG, MIN, MAX, SUM, COUNT	AVG, MIN, MAX, SUM, COUNT
Distinct	F-DISTINCT(A)	F (DISTINCT A)
Aggregate Operation	$\mathcal{G}_{F_1(A_1), \dots, F_n(A_n)}(R)$ $A_x \mathcal{G}_{F_1(A_1), \dots, F_n(A_n)}(R)$	SELECT F ₁ (A ₁), ..., F _n (A _n) FROM R SELECT A _x , F ₁ (A ₁), ..., F _n (A _n) FROM R GROUP BY A _x

- E₁, E₂, ..., E_n are arithmetic expressions involving constants and attribute names of R.
- AVG, MIN, MAX, SUM and COUNT are aggregate functions taking a multiset of values as input and returning a single value.
- AVG-DISTINCT, ..., COUNT-DISTINCT are similar, but ignore duplicates in the multiset argument.
- F is an aggregate function and A an attribute.
- F_i are aggregate functions, and A_i is an attribute name of R.
- A_x is a list of attributes on which to group; Can be empty.

Extended Operations in Relational Algebra

10.1.12 Generalized Projection

For all instructors increase salary with 3% and show monthly instead of yearly salary.

$$\Pi_{\text{InstID}, \text{InstName}, \text{DeptName}, \text{Salary} * 1.03/12} (\text{Instructor})$$

10.1.13 Aggregate Operation

Find the maximum, minimum, average and sum of all instructor salaries.

$$G_{\text{MAX}(\text{Salary}), \text{MIN}(\text{Salary}), \text{AVG}(\text{Salary}), \text{SUM}(\text{Salary})} (\text{Instructor})$$

10.1.14 Aggregate Operation with Grouping

Find the maximum and average salary in each department.

$$\text{DeptName } G_{\text{MAX}(\text{Salary}), \text{AVG}(\text{Salary})} (\text{Instructor})$$

Using Relational Algebra for Language Semantics

- Relational Algebra can be used to specify the meaning of relational languages.
- Example:

- The meaning of

SELECT A_1, \dots, A_n FROM R_1, \dots, R_n WHERE P

is

$$\prod_{A_1, \dots, A_n} (\sigma_P(R_1 \times \dots \times R_n))$$

Using Relational Algebra for Query Optimizing

- Relational Algebra can be used for query optimization, see separate slide set 10.5.

Domain Calculus

- **Domain Calculus**
 - Introduced by Michel Lacroix and Alain Pirotte in order to provide a declarative query language based on mathematical logic for the relational model.
 - Domain Calculus, like Tuple Calculus, but in contrast to Relational Algebra, is a non procedural or **declarative language** that describes the answer, without giving specific procedures for how to obtain the answer.
 - Domain Calculus is the **mathematical basis for** the language **QBE**.
- **Domain Calculus versus Relational Algebra**
 - Domain Calculus restricted to Safe Expressions (i.e. expressions with a finite number of tuples in the result), **is equivalent to** Relational Algebra in expressive power **with regard to the Basic operations and derived operations**.
 - Domain Calculus **can be extended** to handle arithmetic expressions as well as aggregation operations. However, that is outside the scope of this lecture.

Domain Calculus

- Each query is of the form: $\{ \langle x_1, x_2, \dots, x_n \rangle \mid P(x_1, x_2, \dots, x_n) \}$
 - x_1, x_2, \dots, x_n represent domain variables (for attributes in the relation)
 - $\langle x_1, x_2, \dots, x_n \rangle$ represents a tuple
 - $P(x_1, x_2, \dots, x_n)$ is a Predicate Calculus formula in the domain variables
- A Predicate Calculus formula
 - Membership test: $\langle v_1, v_2, \dots, v_n \rangle \in R$ and $\langle v_1, v_2, \dots, v_n \rangle \notin R$, where R is a relation having n attributes and each v_i is either a domain variable x_i or a constant c .
 - Comparisons:
 - $x_i \text{ op } x_j$ or $x_i \text{ op } c$,
where op is an operator (like $<$, \leq , $=$, \neq , $>$, \geq), x_i and x_j are domain variables and c is a constant.
 - $f_1 \wedge f_2, f_1 \vee f_2, f_1 \Rightarrow f_2, \neg f_1$, where f_1 and f_2 are formula
 - Quantified expressions:
 $\exists x_1, \dots, x_n (P(x_1, x_2, \dots, x_n))$ There exists values for x_1, \dots, x_n , for which $P(x_1, x_2, \dots, x_n)$ is true.
 $\forall x_1, \dots, x_n (P(x_1, x_2, \dots, x_n))$ For all values of x_1, \dots, x_n , $P(x_1, x_2, \dots, x_n)$ is true.
 - Free and bound variables x_i :
 - Are defined in a similar (usual) way as for Tuple Calculus.
 - For $\{ \langle x_1, x_2, \dots, x_n \rangle \mid P(x_1, x_2, \dots, x_n) \}$ only x_1, x_2, \dots, x_n are allowed to be free in $P(x_1, x_2, \dots, x_n)$

Basic Operations of Domain Calculus

■ Equivalent Operations in Relational Algebra and Domain Calculus

Basic Operations	Relational Algebra	Domain Calculus Basic Operations	
Selection	$\sigma_{B>3, C=B}(R)$	$\{< a, b, c > \mid < a, b, c > \in R \wedge b > 3 \wedge c = b\}$	Given $R(A, B, C)$
Projection	$\Pi_{B,A}(R)$	$\{< b, a > \mid \exists c (< a, b, c > \in R)\}$	Given $R(A, B, C)$
Set Union	$R \cup S$	$\{< a, b, c > \mid < a, b, c > \in R \vee < a, b, c > \in S\}$	Given R and S are compatible
Set Difference	$R - S$	$\{< a, b, c > \mid < a, b, c > \in R \wedge < a, b, c > \notin S\}$	Given R and S are compatible
Cartesian Product	$R \times S$	$\{< a, b, c, d > \mid < a, b > \in R \wedge < c, d > \in S\}$	Given $R(A, B)$ and $S(C, D)$

Basic Operations in Domain Calculus

Using the University Schema



10.3.1 Selection Operation

Find Instructor rows for instructors in the physics department earning more than 90000.

$$\{<i,n,d,s> \mid <i,n,d,s> \in \text{Instructor} \wedge d = \text{'Physics'} \wedge s > 90000\}$$

10.3.2 Projection Operation

Find name and IDs of all instructors

$$\{<n,i> \mid \exists d, s (<i,n,d,s> \in \text{Instructor})\}$$

10.3.3 Set Union

Find name and IDs of all instructors and students.

$$\{<n,i> \mid \exists d, s (<i,n,d,s> \in \text{Instructor}) \vee \exists b, d, t (<i,n,b,d,t> \in \text{Student})\}$$

10.3.4 Set Difference

Find instructor IDs for instructors that have not taught any courses yet.

$$\{<i> \mid \exists n, d, s (<i,n,d,s> \in \text{Instructor}) \wedge \neg \exists c, sec, sem, sy (<i,c,sec,sem,sy> \in \text{Teaches})\}$$

10.3.5 Cartesian Product

Combine each instructor with each student

$$\{<i,n,d,s,si,sn,b,sd,t> \mid (<i,n,d,s> \in \text{Instructor}) \wedge (<si,sn,b,sd,t> \in \text{Student})\}$$

Derived Operations of Domain Calculus

■ Equivalent Operations in Relational Algebra and Domain Calculus

Derived Operations	Relational Algebra	Domain Calculus Derived Operations	
Natural Join	$R \bowtie S$	$\{ \langle a, b, c \rangle \mid \langle a, b \rangle \in R \wedge \langle b, c \rangle \in S \}$	Given $R(A, B)$ and $S(B, C)$
Left Outer Join	$R \bowtie L S$	Self Study	
Right Outer Join	$R \bowtie R S$	Self Study	
Full Outer Join	$R \bowtie F S$	Self Study	
Set Intersection	$R \cap S$	$\{ \langle a, b, c \rangle \mid \langle a, b, c \rangle \in R \wedge \langle a, b, c \rangle \in S \}$	Given R and S are compatible
Assignment	$S \leftarrow R$	NA	

Derived Operations in Domain Calculus

10.3.7 Natural Join Find instructor names and course IDs they have taught. (Join on attribute InstID).

10.3.8 Natural Join

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

10.3.9 Left Outer Join (skipped)

Find student names and their advisor IDs

10.3.10 Set Intersection

Find courses that were taught both in Fall 2009 and in Spring 2010.

```
Course(CourseID, Title, DeptName, Credits)
Instructor(InstID, InstName, DeptName, Salary)
Section(CourseID, Section, Semester, StudyYear, Building,
Room, TimeSlotID)
Teaches(InstID, CourseID, SectionID, Semester, StudyYear)
```

$$\{<\mathbf{n},\mathbf{c}> \mid \exists i,d,s(<\mathbf{i},\mathbf{n},\mathbf{d},\mathbf{s}> \in \text{Instructor} \wedge \exists \mathbf{sec},\mathbf{sem},\mathbf{y}(<\mathbf{i},\mathbf{c},\mathbf{sec},\mathbf{sem},\mathbf{y}> \in \text{Teaches})) \}$$
$$\{<\mathbf{n},\mathbf{t}> \mid \exists i,d,s(<\mathbf{i},\mathbf{n},\mathbf{d},\mathbf{s}> \in \text{Instructor} \wedge d = \text{'Comp. Sci.'} \wedge \exists \mathbf{c},\mathbf{sec},\mathbf{sem},\mathbf{y}(<\mathbf{i},\mathbf{c},\mathbf{sec},\mathbf{sem},\mathbf{y}> \in \text{Teaches} \wedge \exists \mathbf{d},\mathbf{cr}(<\mathbf{c},\mathbf{t},\mathbf{d},\mathbf{cr}> \in \text{Course}))) \}$$

Self Study

$$\begin{aligned} &\{<\mathbf{ci}> \mid \\ &(\exists \mathbf{sec},\mathbf{sem},\mathbf{y},\mathbf{b},\mathbf{r},\mathbf{tsi}(<\mathbf{ci},\mathbf{sec},\mathbf{sem},\mathbf{y},\mathbf{b},\mathbf{r},\mathbf{tsi}> \in \text{Section} \wedge \\ &\text{sem} = \text{'Fall'} \wedge \mathbf{y} = 2009)) \\ &\wedge \\ &(\exists \mathbf{sec},\mathbf{sem},\mathbf{y},\mathbf{b},\mathbf{r},\mathbf{tsi}(<\mathbf{ci},\mathbf{sec},\mathbf{sem},\mathbf{y},\mathbf{b},\mathbf{r},\mathbf{tsi}> \in \text{Section} \wedge \\ &\text{sem} = \text{'Spring'} \wedge \mathbf{y} = 2010)) \\ &\} \end{aligned}$$



Summary

- Mathematical Foundation
- Relational Algebra
- Domain Calculus

Readings

- In Database Systems Concepts please read Chapter 6.
(Remember to check the book typo list – there are several typos.)
- Pay special attention to the Summary

Demo Exercises



Demo Exercises clarify ideas and concepts from the Lecture to provide you with good Database Skills.

Make the
Demo Exercises.

Formal Query Languages

10.4.1 Selection and Projection Query

Find name and IDs of all instructors in the physics department earning more than 90000.

- a) Make the query in SQL
- b) Make the query in Relational Algebra
- c) Make the query in Domain Calculus

Formal Query Languages

10.4.2 Query using aggregation

Find the number of courses provided by the Biology department (in the university database).

- a) Make the query in SQL
- b) Make the query in Relational Algebra

Solutions to Demo Exercises



Formal Query Languages

10.4.1 Selection and Projection Query

Find name and IDs of all instructors in the physics department earning more than 90000.

- a) Make the query in SQL
- b) Make the query in Relational Algebra
- c) Make the query in Domain Calculus

a) SQL:

```
SELECT InstName, InstID FROM Instructor  
WHERE DeptName = 'Physics' AND Salary >  
90000;
```

b) Relational Algebra:

$$\prod_{\text{InstName, InstID}} (\sigma_{\text{DeptName} = \text{'Physics'}} \wedge \text{Salary} > 90000)(\text{Instructor})$$

c) Domain Calculus:

$$\{ \langle n, i \rangle \mid \exists d, s (\langle i, n, d, s \rangle \in \text{Instructor} \wedge d = \text{'Physics'} \wedge s > 90000) \}$$

Formal Query Languages

10.4.2 Query using aggregation

Find the number of courses provided by the Biology department (in the university database).

a) Make the query in SQL

b) Make the query in Relational Algebra

a) SQL:

```
SELECT COUNT(CourseID) FROM Course  
WHERE DeptName = 'Biology';
```

b) Relational Algebra:

$$G_{\text{COUNT}(\text{CourseID})} (\sigma_{\text{DeptName} = \text{'Biology'}} (\text{Course}))$$

Exercises



Please answer all exercises
to demonstrate your
Database Skills.

Pencil, Paper & MySQL
Exercises

Solutions are available at 11:45

Formal Query Languages

10.5.1 Selection and Projection Query

Consider the relation schema:

Employee(Eid, Ename, Profession, Rate)

and consider the query: Find the employees Eid and Ename for employees with Profession 'Carpenter', and whose Rate is less than 100 \$/Hour.

- a) Make the query in SQL.
Create the relation, populate with some example data, and make the query.

- b) Make the query in Relational Algebra

- c) Make the query in Domain Calculus

10.5.2 Join Query

As continuation of 10.5.1, consider the Relation Schemas:

Employee(Eid, Ename, Profession, Rate)

Projects(Pid, Pname)

Staffing(Pid, Eid, Hours)

and consider the query:

Find for each staffed project and each of the employees in its staff: Pid, Pname, Eid, Ename, Hours and Rate.

- a) Make the query in SQL.
Create the relations, populate with some example data, and make the query.

- b) Make the query in Relational Algebra

- c) Make the query in Domain Calculus

Formal Query Languages

10.5.3 Aggregation and Grouping

Consider the University database. Find for each course offered in autumn 2009 the number of students who have taken that course.

- a) Make the query in Relational Algebra.
- b) Make the query in SQL.



Technical University of Denmark

DTU Compute

Department of Applied Mathematics and Computer Science

Query Optimization

©Giovanni Meroni

Query optimization

- **Queries consist of several atomic operations**
 - Selection, projection, join, etc....
- **Operations can be executed in different order, as long as the result remains the same**
 - E.g., selection then join vs join then selection
- **The order in which operations must be performed is called execution plan**
 - Although leading to the same result, execution plans can have substantial differences in processing time and memory usage
- **Optimization finds the best execution plan**
 - Completely carried out by the DBMS
 - Users do not have to optimize their queries

From SQL to relational algebra

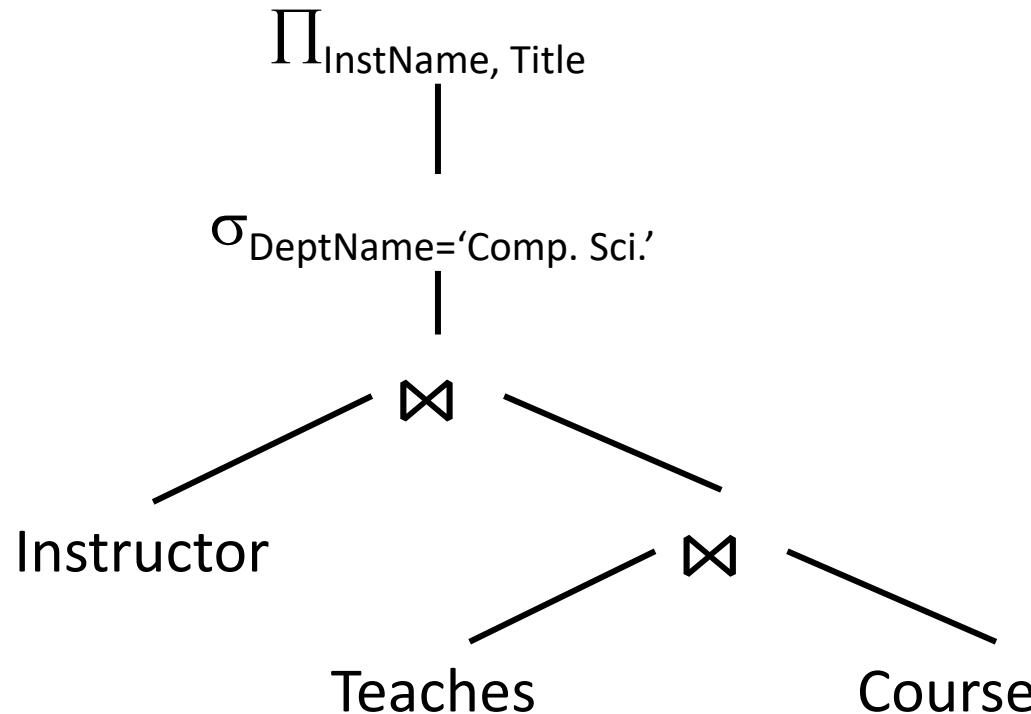
- Find the names of instructors in the Comp. Sci. department together with the course titles of the courses that the instructors teach.
 - `SELECT InstName, Title FROM Instructor NATURAL JOIN Teaches NATURAL JOIN Course WHERE deptname = 'Comp. Sci.'`
- This SQL query can be rewritten in relational algebra:
 - $\prod_{\text{InstName, Title}} (\sigma_{\text{DeptName} = \text{'Comp. Sci.'}} (\text{Instructor} \bowtie \text{Teaches} \bowtie \text{Course}))$
- From this expression, the DBMS applies transformation rules to calculate equivalent expressions.
- The DBMS then selects the one that minimizes the processing time
 - Some transformation rules always produce more efficient expressions
 - In the other cases, the DBMS estimates the cost of each operation, and selects the expression that minimizes the total cost.

Syntax tree

- Relational algebra expressions can be represented as a syntax tree:
 - Leaf nodes represent the relations
 - All other nodes represent the operations
 - The closer a node is from the leaves, the sooner the corresponding operation will be performed

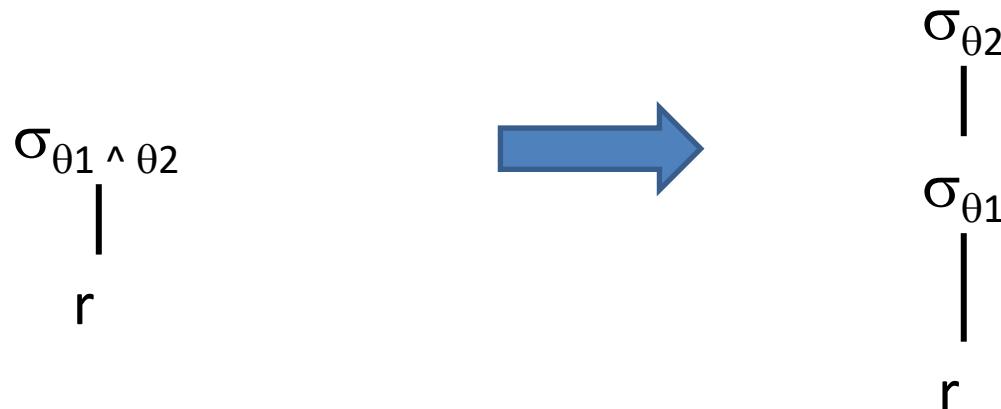
Syntax tree

- Find the names of instructors in the Comp. Sci. department together with the course titles of the courses that the instructors teach.
 - $\prod_{\text{InstName, Title}} (\sigma_{\text{DeptName}=\text{'Comp. Sci.'}} (\text{Instructor} \bowtie \text{Teaches} \bowtie \text{Course}))$



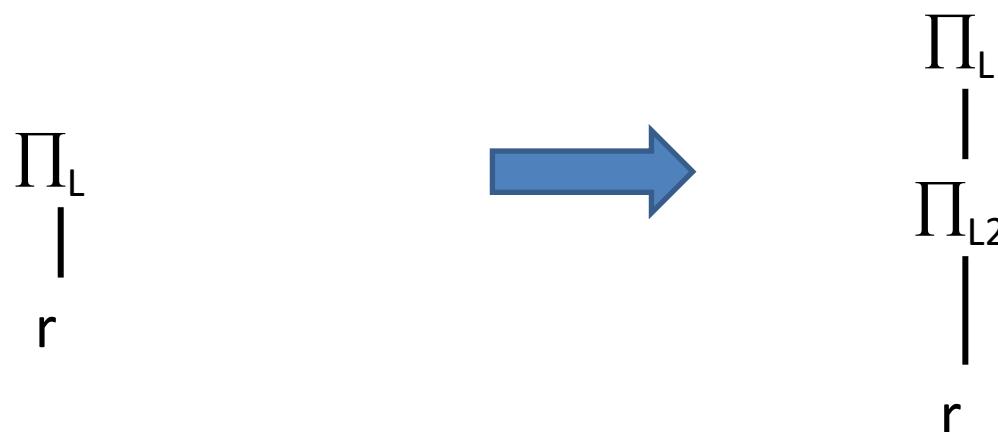
Equivalence of expressions

- Making selection operations atomic
 - Can be done only if θ_1 and θ_2 are in conjunction (\wedge).



Equivalence of expressions

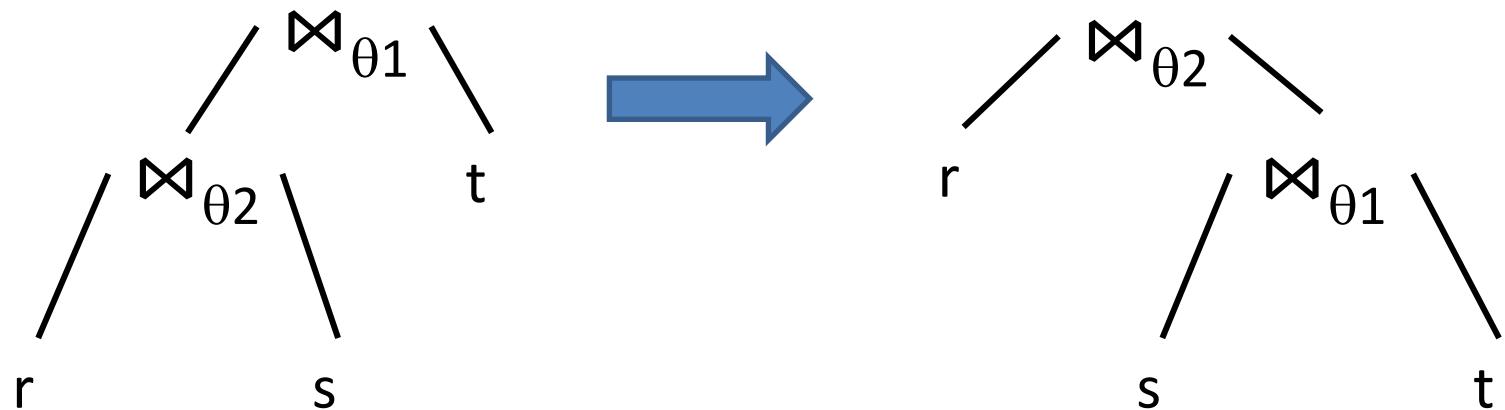
- Expanding projection operations
 - Can be done only if $L \subseteq L_2$.



Equivalence of expressions

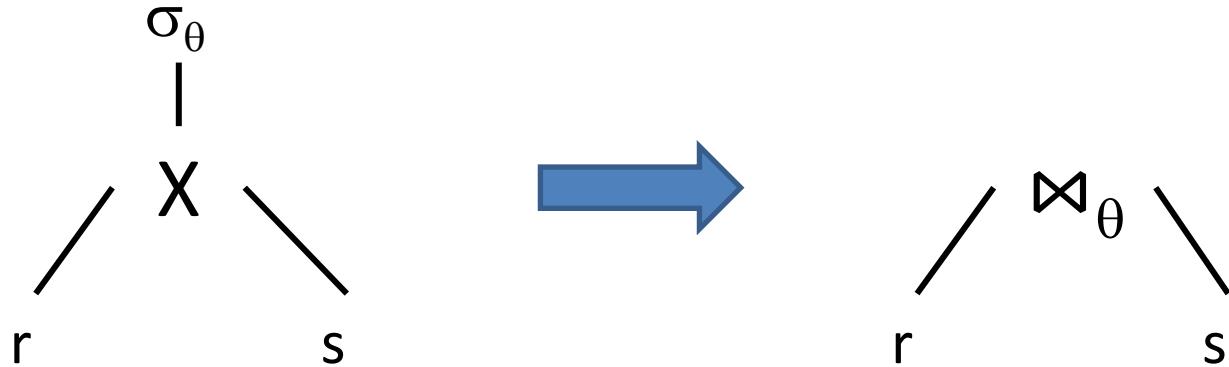
■ Switching natural joins

- Can be done only if θ_1 does **not** refer to attributes of r



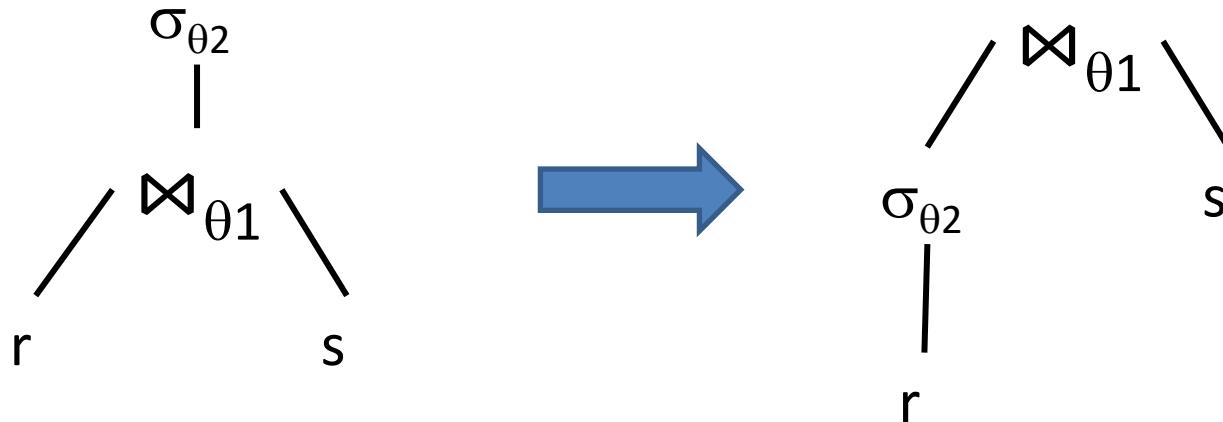
Query optimizations

- Removing cartesian product operations



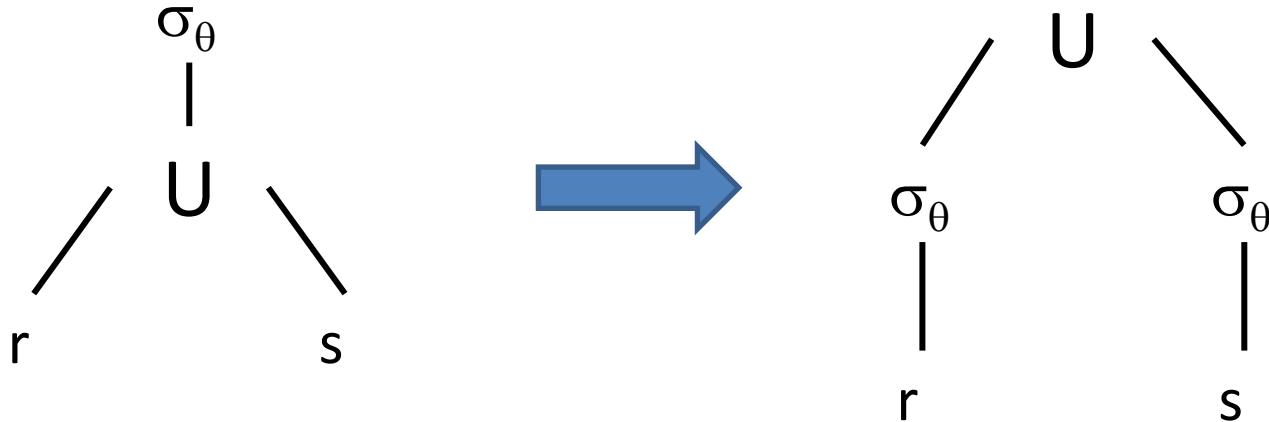
Query optimizations

- Pushing selection operations before joins
 - Can be done only if θ_2 refers only to attributes of r .



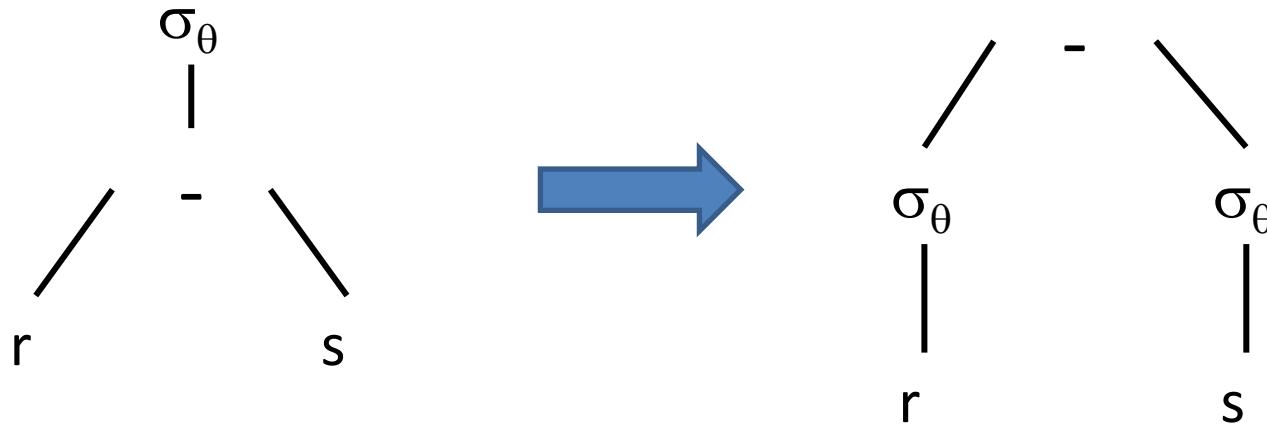
Query optimizations

- Pushing selection operations before unions



Query optimizations

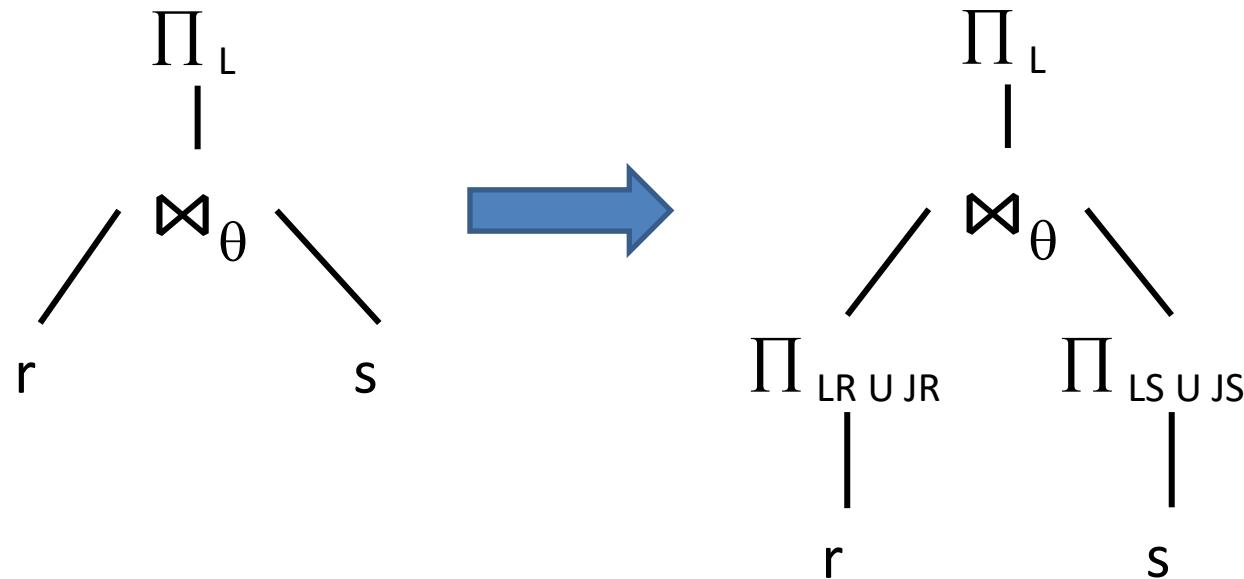
- Pushing selection operations before differences



Query optimizations

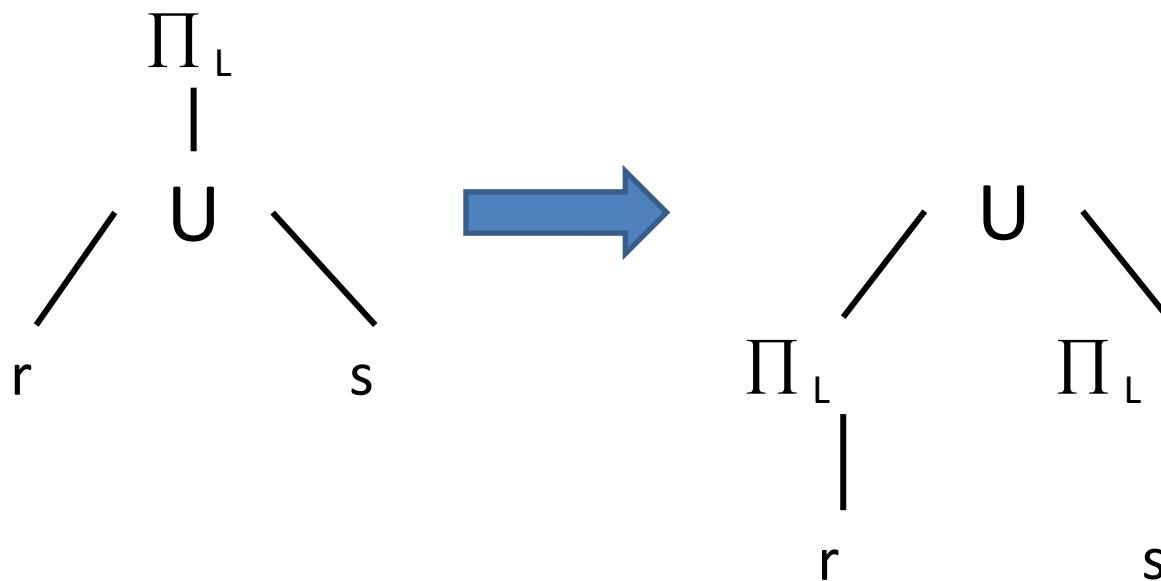
Pushing projection operations before joins

- $LR = L - \text{Schema}(s)$
- $LS = L - \text{Schema}(r)$
- $J = JR \cup JS$ are the attributes referenced in θ



Query optimizations

- Pushing projection operations before unions



Back to the example

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

InstName	Title
Srinivasan	Intro. to Computer Science
Srinivasan	Robotics
Srinivasan	Database System Concepts
Katz	Intro. to Computer Science
Katz	Image Processing
Brandt	Game Design
Brandt	Game Design
Brandt	Image Processing

$\prod_{\text{InstName}, \text{Title}}$

InstID	DeptName	CourseID	SectionID	Semester	StudyYear	Title	Credits	InstName	Salary
76766	Biology	BIO-101	1	Summer	2009	Intro. to Biology	4	Crick	72000.00
76766	Biology	BIO-301	1	Summer	2010	Genetics	4	Crick	72000.00
10101	Comp. Sci.	CS-101	1	Fall	2009	Intro. to Computer Science	4	Srinivasan	65000.00
45565	Comp. Sci.	CS-101	1	Spring	2010	Intro. to Computer Science	4	Katz	75000.00
83821	Comp. Sci.	CS-190	1	Spring	2009	Game Design	4	Brandt	92000.00
83821	Comp. Sci.	CS-190	2	Spring	2009	Game Design	4	Brandt	92000.00
10101	Comp. Sci.	CS-315	1	Spring	2010	Robotics	3	Srinivasan	65000.00
45565	Comp. Sci.	CS-319	1	Spring	2010	Image Processing	3	Katz	75000.00
83821	Comp. Sci.	CS-319	2	Spring	2010	Image Processing	3	Brandt	92000.00
10101	Comp. Sci.	CS-347	1	Fall	2009	Database System Concepts	3	Srinivasan	65000.00
98345	Elec. Eng.	EE-181	1	Spring	2009	Intro. to Digital Systems	3	Kim	80000.00
12121	Finance	FIN-201	1	Spring	2010	Investment Banking	3	Wu	90000.00
32343	History	HIS-351	1	Spring	2010	World History	3	El Said	60000.00
15151	Music	MU-199	1	Spring	2010	Music Video Production	3	Mozart	40000.00
22222	Physics	PHY-101	1	Fall	2009	Physical Principles	4	Einstein	95000.00

$\sigma_{\text{DeptName} = \text{'Comp. Sci.'}}$

InstID	InstName	DeptName	Salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

Instructor

InstID	CourseID	SectionID	Semester	StudyYear
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

Teaches

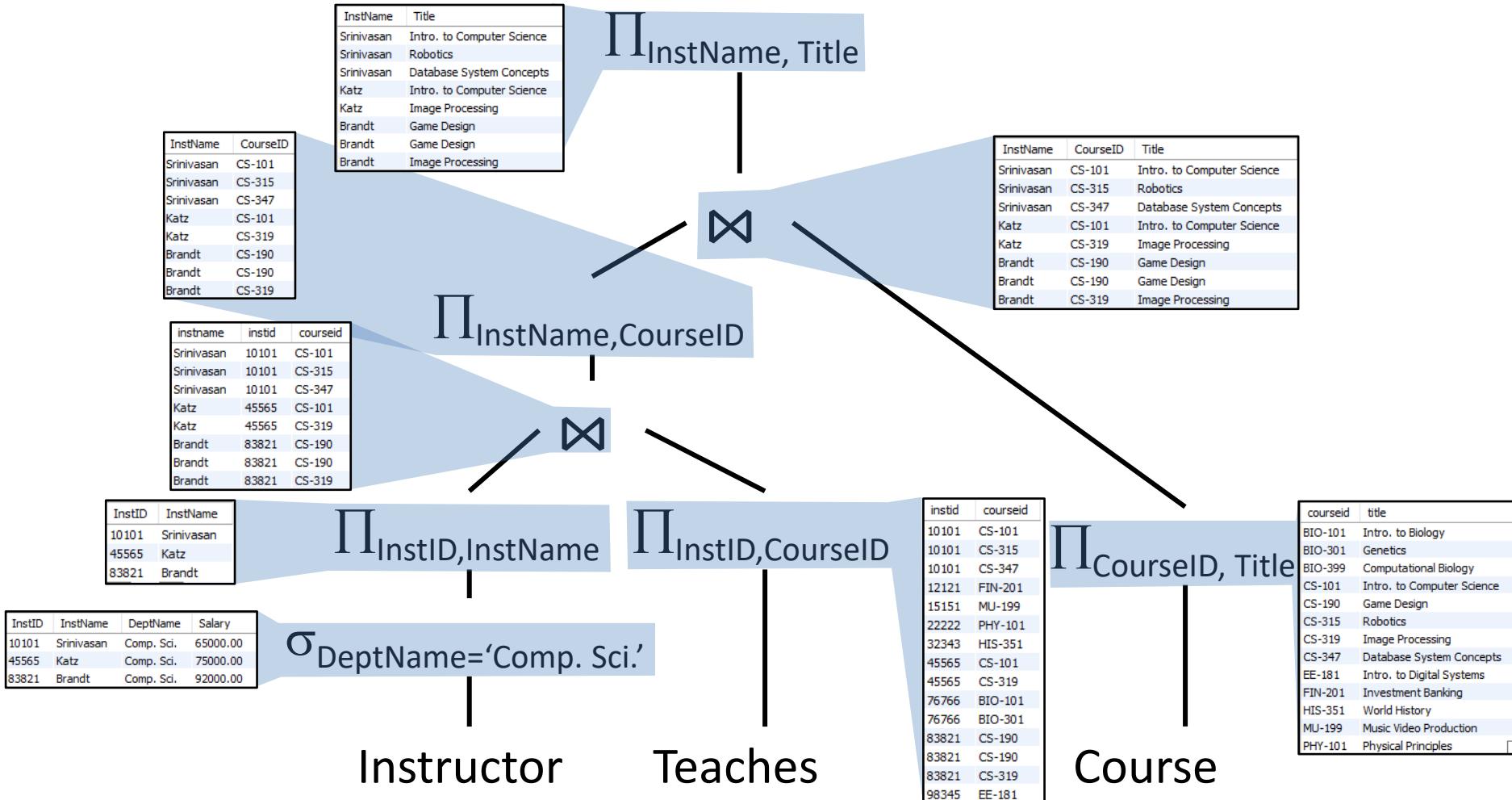
InstID	DeptName	CourseID	SectionID	Semester	StudyYear	Title	Credits	InstName	Salary
10101	Comp. Sci.	CS-101	1	Fall	2009	Intro. to Computer Science	4	Srinivasan	65000.00
10101	Comp. Sci.	CS-315	1	Spring	2010	Robotics	3	Srinivasan	65000.00
10101	Comp. Sci.	CS-347	1	Fall	2009	Database System Concepts	3	Srinivasan	65000.00
45565	Comp. Sci.	CS-101	1	Spring	2010	Intro. to Computer Science	4	Katz	75000.00
45565	Comp. Sci.	CS-319	1	Spring	2010	Image Processing	3	Katz	75000.00
83821	Comp. Sci.	CS-190	1	Spring	2009	Game Design	4	Brandt	92000.00
83821	Comp. Sci.	CS-190	2	Spring	2009	Game Design	4	Brandt	92000.00
83821	Comp. Sci.	CS-319	2	Spring	2010	Image Processing	3	Brandt	92000.00

Course

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

Applying optimizations

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





Summary

- Query syntax tree
- Equivalent expressions
- Query optimizations



Technical University of Denmark

DTU Compute

Department of Applied Mathematics and Computer Science

Indexing & Hashing

Performance Optimization of Relational File Systems

Chapter 11 of the Textbook

©Anne Haxthausen

These slides have been prepared by Anne Haxthausen, partly reusing/modifying some slides by Flemming Schmidt.
Some examples come from Silberschatz, Korth, Sudarshan, 2010.

Contents

- Background Knowledge: File Structure and Organization (book chapter 10)
- Indexing
- Hashing

File Structure, Organization and Storage

File Structure

- A *file* is a sequence of records.
- A *record* is a sequence of *fields*.

Storage of a database

- A *database* is stored in *files* on a *disk*:
 - Each *table (relation)* is stored in a *file*:
 - Each *row* is stored in a *record* of the file:
 - Each *attribute value* is stored in a *field* in the record.
 - The files are partitioned into fixed-length storage units called *blocks*.

- Example:

Block 1

record 0
record 1
record 2

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000

Block 2

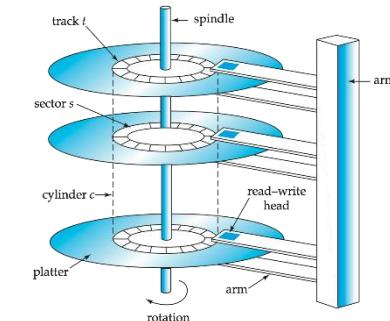
record 3
record 4
record 5

22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000

Block 3

record 6

45565	Katz	Comp. Sci.	75000
-------	------	------------	-------



File Organization

- **File organization** refers to the way records are organized in a file.
 - **Heap File**
 - A record can be placed anywhere in the file where there is space.
 - **Sequential File**
 - Records are stored *in sequential order*, based on the value of the *search key* of each record.
 - Example - Instructor table with InstID as search key:
 - **Hash File**
 - A hash function computed on some attribute of each record; The result specifies in which block of the file the record should be placed.

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	

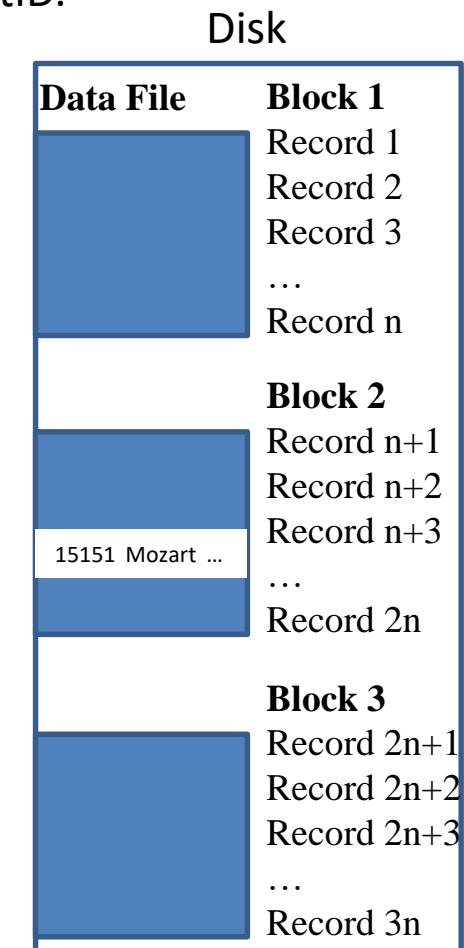
How to Find a Record Fast?

- **SELECT** statements give rise to searching for Records.

Example: **SELECT** InstName **FROM** Instructor **WHERE** InstID = 15151;

- Find the Record with *Search Key* 15151 for Instructor.InstID.
- The Instructor file is in 3 Blocks on the Disk.
- One way:
Read the file sequentially, loading blocks one by one into Memory until the record is found.

- Can this be done faster?



How to Find a Record Fast for a Given Search Key?

1. Use an Index File of Search Keys

1st field: Search Key

2nd field: Pointer

The Pointer points to the start of the Block where the Record is stored, or directly to the record.

Index File

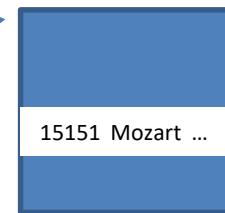
Search_Key	Pointer
Value 1	Block 1
Value 2	Block 1
Value 3	Block 3
...	
15151	Block 2
...	

Data File



Block 1

Record 1
Record 2
Record 3
...
Record n



Block 2

Record n+1
Record n+2
Record n+3
...
Record 2n



Block 3

Record 2n+1
Record 2n+2
Record 2n+3
...
Record 3n

2. Use a Hashing Algorithm on the Search Key

Search Key = 15151

Hash Algorithm: Search Key % 3 +1

Block Address: $15151 \% 3 + 1 = 2$

- The x % y modulus operation is the remainder of x/y

Using hashing saves to have the index file.

Summary

- A File can be read sequentially, one block and one record after another!
 - But indexing mechanisms can be used to **speed up** access dramatically.
- A Search Key can be defined to look up records in a file
 - Search Keys can be stored in an Index File with pointers.
 - First attribute is the Search Key, and the second attribute is a Pointer.
- Index Files are much smaller than the original file
 - An Index File is loaded from disk and often fits into memory.
- Alternatively a Hashing Algorithm can be defined
 - Based on a Search Key it calculates in which Block a record has to be read from or written to.
- Index Evaluation Metrics
 - Access time, insertion time, deletion time and space overhead.

Indexing

- Concepts for Ordered Index Files
 - Primary versus Secondary Indexes
 - Dense versus Sparse Indexes
 - Multi-column Indexes
 - Multilevel Indexes
- Index File Organization:
 - Sequential Files
 - B⁺-Tree
 - ...

Index Definition in SQL

- **Create an index**

CREATE [UNIQUE] INDEX <index-name> ON <table-name>(<attribute-list>)

Like: **CREATE INDEX NameIndex ON Instructor(InstName);**

Most database systems allow a specification of **the type** of index.

Like: **CREATE INDEX NameIndex ON Instructor(InstName) USING BTREE;**

CREATE INDEX NameIndex ON Instructor(InstName) USING HASH;

Each storage engine supports some or all index types. The default storage engine INNODB only allows BTREE.

MariaDB uses BTREE as **default**, if no type is specified.

- ✓ MySQL and MariaDB will **automatically** for each table define a Primary Index on the Primary Key and Secondary Indexes for all Foreign Keys.

- **To drop an index**

DROP INDEX <index-name> ON <table-name>;

Like: **DROP INDEX NameIndex ON Instructor;**

- **To show the index set for a table**

SHOW INDEX FROM <table-name>;

Like: **SHOW INDEX FROM Instructor;**

Ordered Index Files

■ (Sequentially) Ordered Index

- Index File is sorted on the Search Key which is one or several attributes from the data file.
- Since it is sorted we can use binary searching to find an index.

Index File	
Search Key	Pointer
1	
2	
3	

Data File (for a relation)			
A1	A2	A3	A4

■ Primary Index

- Index File and Data File are both sorted on the same Search Key.
- It is the Primary Index that determines the order of the records in the Data File. The Index is also called a **Clustering Index**, since it determines the clustering of records in blocks.
- A relation has at most one Primary Index. It is usually (but not necessarily) defined on the primary key attribute. In many DBMS the primary key automatically gets an index.
- Note: some literature/books use the terms primary index and clustering index with different meanings!

■ Secondary Index = Index which is not Primary

- Index File and the Data File are sorted on different Search Keys.
- A Secondary Index does not determine any order on the records in the Data File.
- Exists only, if there is a Primary Index.
- A relation can have several Secondary Indexes.

Dense Versus Sparse Index Files

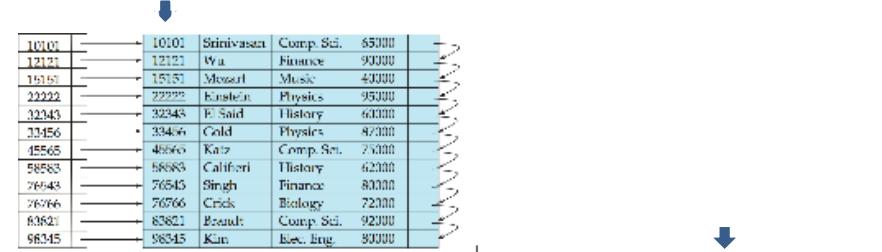
- **Dense Index**
 - The Index File has a record for every Search Key value in the Data File
- **Sparse Index = Non-Dense Index**
 - The Index File has **not** a record for every Search Key value in the Data File

Examples of Ordered Indexes

- Examples of ordered indexes on following slides

1. Primary indexes:

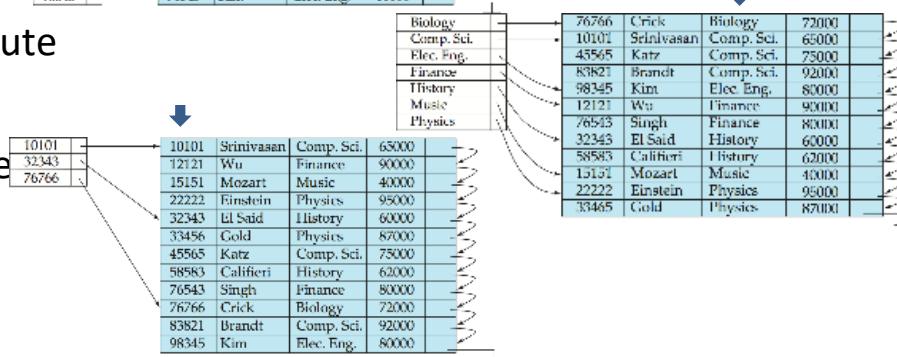
1. Dense index on (primary) key attribute



2. Dense index on non primary key attribute

3. Sparse index on (primary) key attribute

4. Sparse index on non key attribute

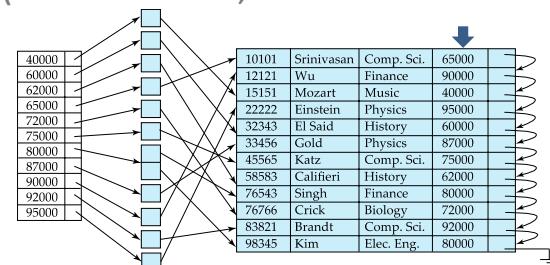


2. Secondary indexes:

1. Dense index on key attribute: no example (similar to 1.1, but with arrows crossing)

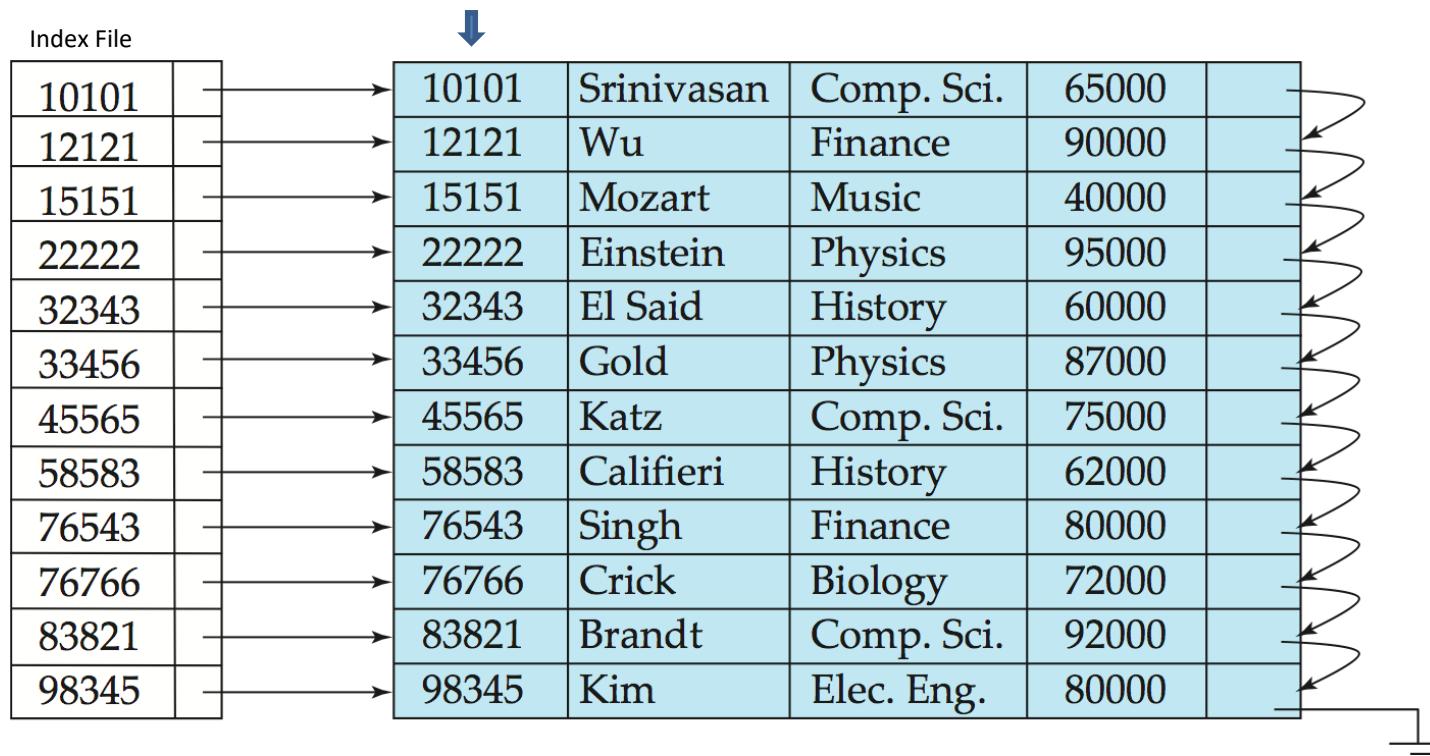
2. Dense index on non primary key attribute

3. Sparse index is **not** an option



Primary Index which is Dense

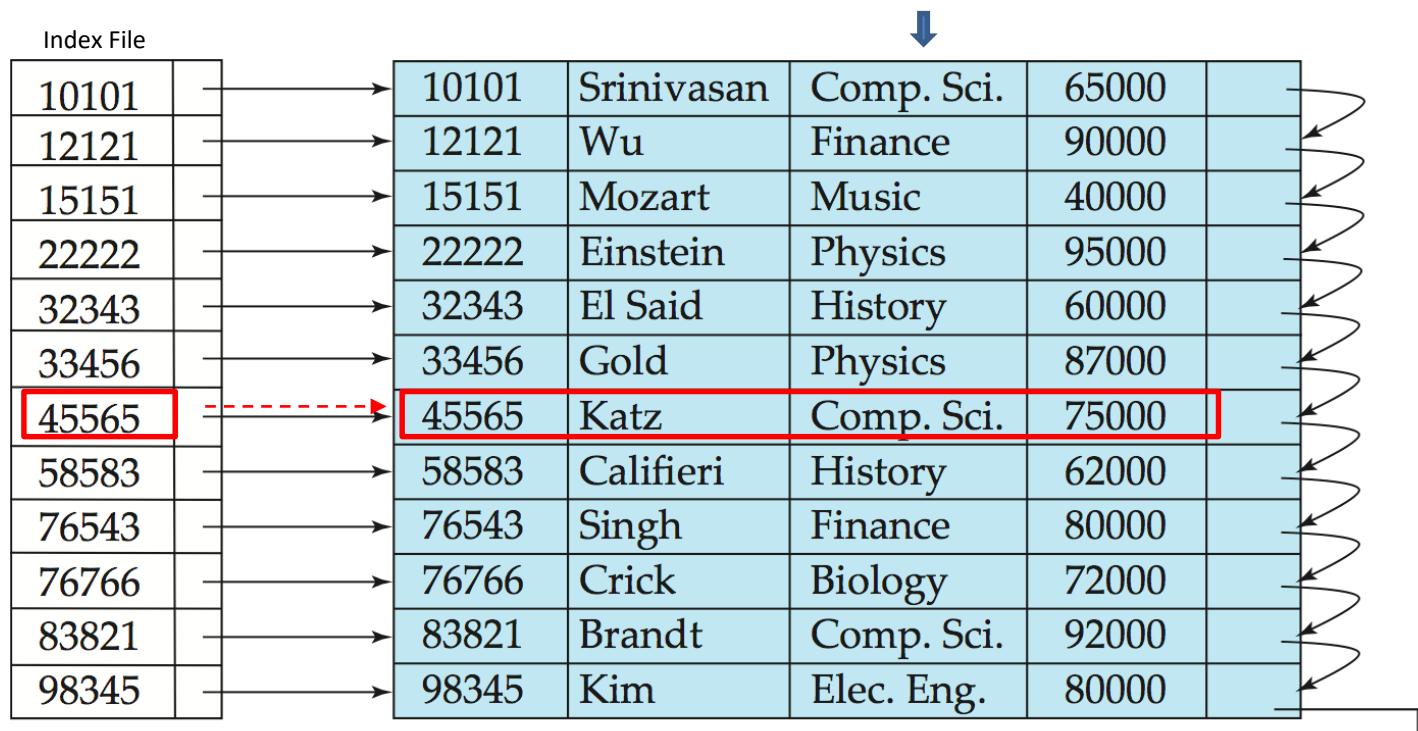
- Dense, Ordered Index on the Primary Key attribute Instructor.InstID



- The Index is Ordered (Index File is sorted on the Search Key).
- The Index is Primary (Data File are sorted on the same Search Key).
- The Index is Dense (All File Search Keys have an Index File Record).

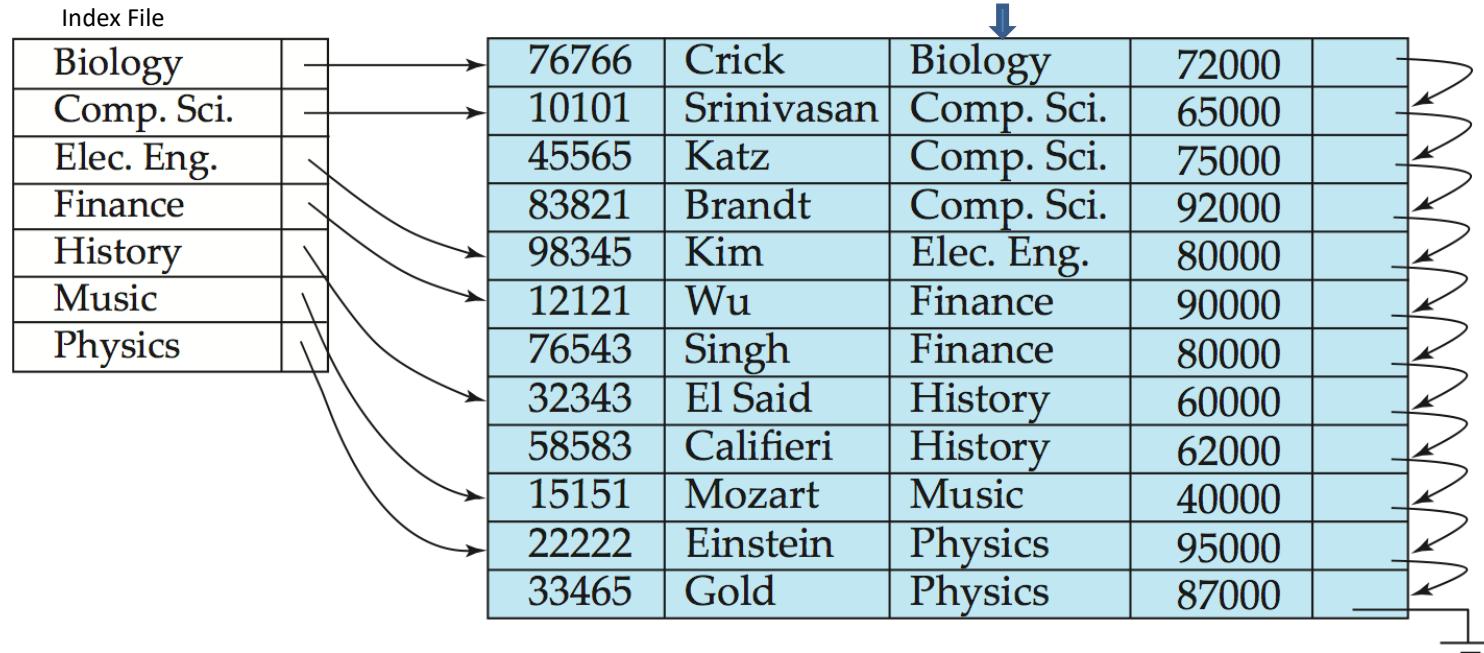
Queries Using Dense Index

- To locate records with Search Key value = K (e.g. Instructor.InstID = 45565)
 - Find the record with Search Key value V = K in the Index File.
In practice: read the index into memory and use binary searching.
 - Follow the associated pointer to the record in the Data File and read the record (block containing it) into main memory.



Primary Index Which Is Dense

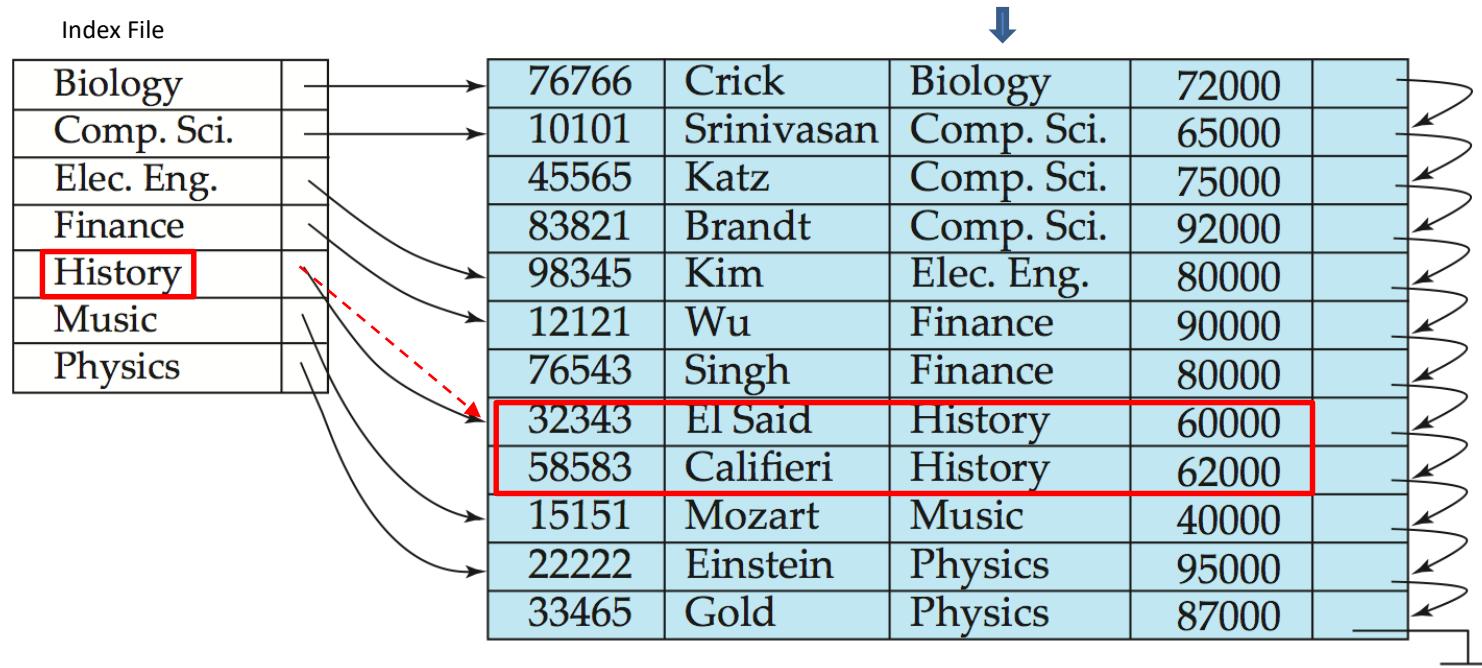
- Dense, Ordered Index on a Non-Key Attribute Instructor.DeptName



- The Index is Ordered (Index File is sorted on the Search Key).
- The Index is Primary (Index File and File are sorted on the same Search Key).
- The Index is Dense (Index File has a record for every Search Key value in the Data File).

Queries Using a Dense Index

- To locate records with Search Key value = K (e.g. Instructor.DeptID = History)
 - Find the record with Search Key value V = K in the Index File.
In practice: read the index into memory and use binary searching.
 - Follow the associated pointer to the Data File and search sequentially from there.
In practice: read the pointed block into main memory and perform searching on that. Read more blocks, if needed.



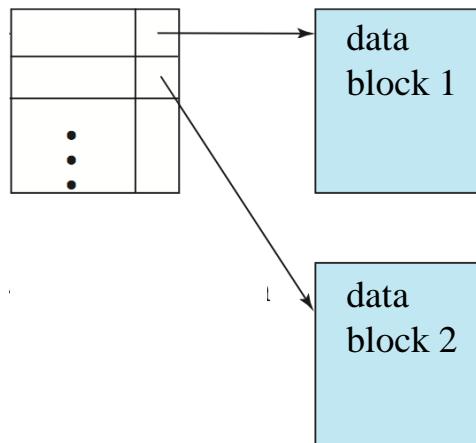
Sparse Index Files

Sparse Index

- By definition: The Index File contains only some of the Search Key values of the Data File.
- Only possible for Primary Indexes (Index File and Data File sorted on the same Search Key), as the same ordering is needed to locate records.

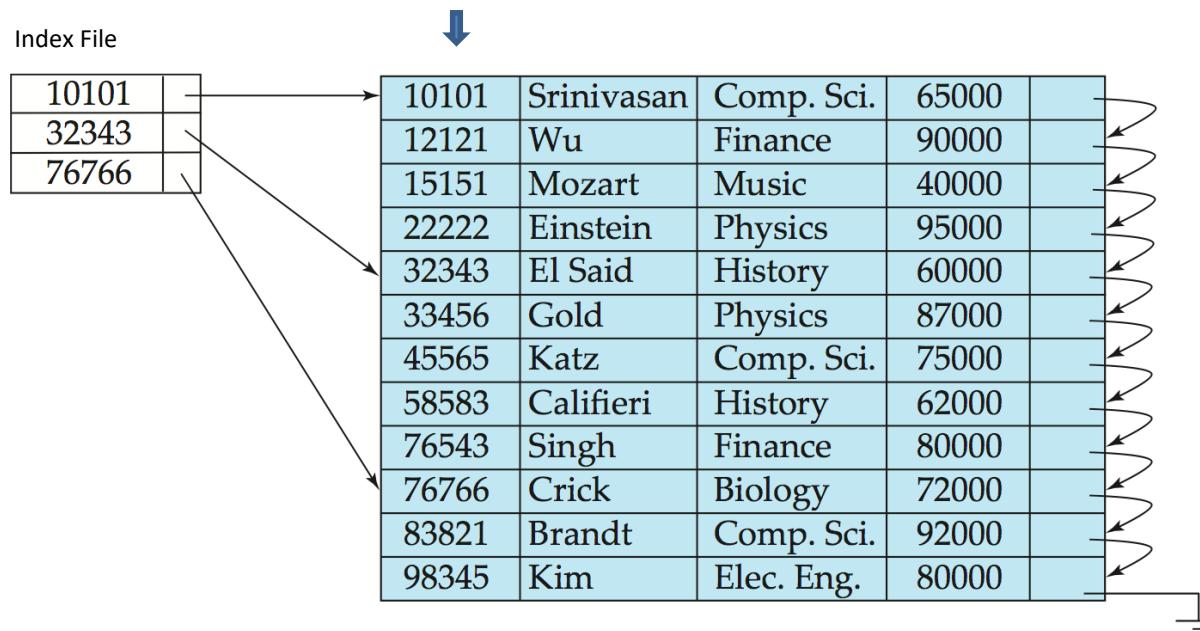
Which Search Keys to include in the index:

- A good choice is to include each key of the 1st record in every block in the file:
 - 1st record in index points to the 1st record in the 1st block of the data file
 - 2nd record in index points to the 1st record in the 2nd block of the data file
 - ...



Primary Index Which Is Sparse

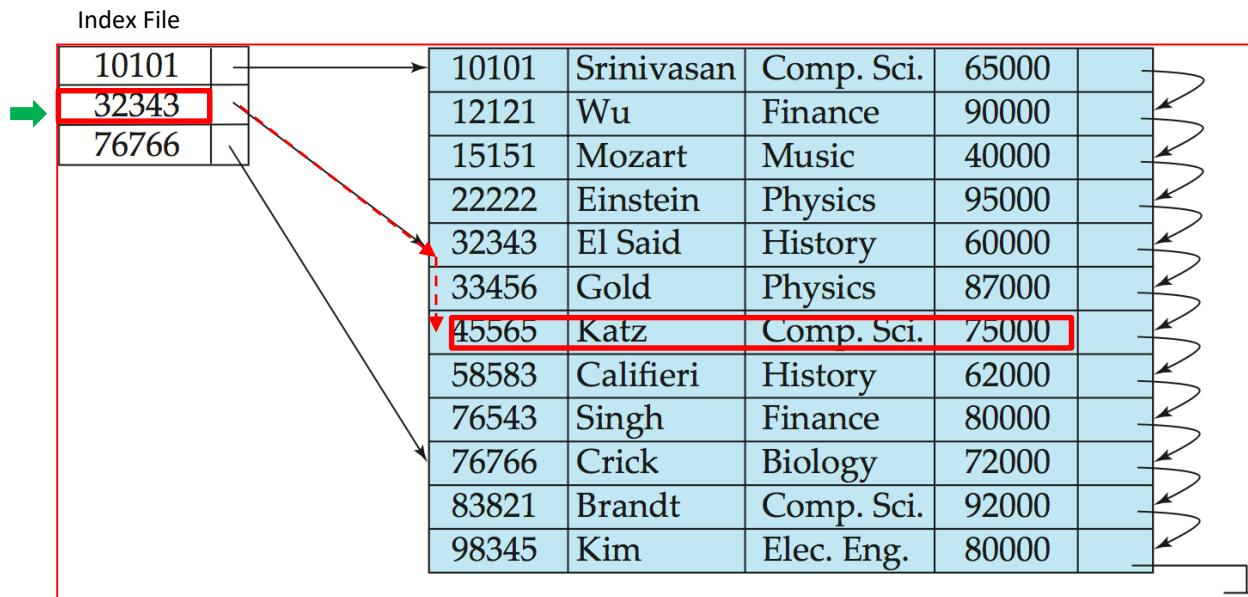
- Sparse, Ordered Index on the Primary Key attribute Instructor.InstID



- The Index is Ordered (Index File is sorted on the Search Key).
- The Index is Primary (Data File are sorted on the same Search Key).
- The Index is Sparse (Not all File Search Keys have an Index File Record).

Queries Using a Sparse Index

- To locate a record with Search Key value K (e.g. Instructor.InstID = 45565)
 - Find the record with largest Search Key value V <= K (V = 32343) in the Index File. In practice: read the index into memory and use binary searching.
 - Follow the pointer to a record in the data File and start sequential searching. In practice: read the pointed block into main memory and perform searching on that. If it is not in that block, read next block and search in that. And so on.



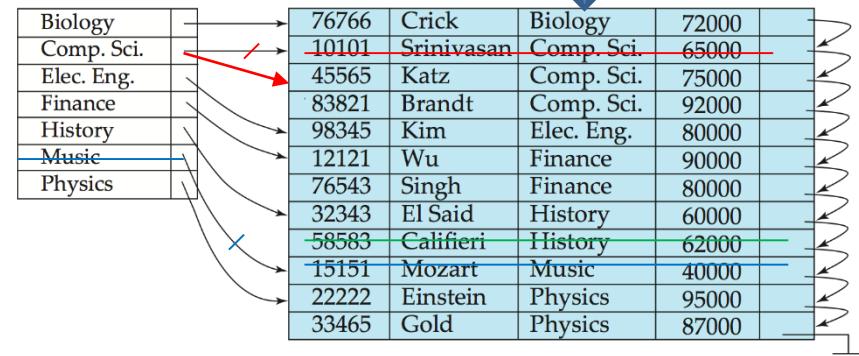
Index File Update After an Insertion in the Data File

- **Single-level Index File record insertion**
 - Search for the Search Key value of the record to be inserted
 - Dense Index:
If the Search Key value does not appear in the index, then insert it
 - Sparse Index:
 - If the Index File stores an entry for each block of the Data File, no change needs to be made to the Index File unless a new block is created.
 - If a new block is created, the first search key value appearing in the new block is inserted into the Index File.
- **Multilevel insertion and deletion**
 - Algorithms are simple extensions of the single-level algorithms

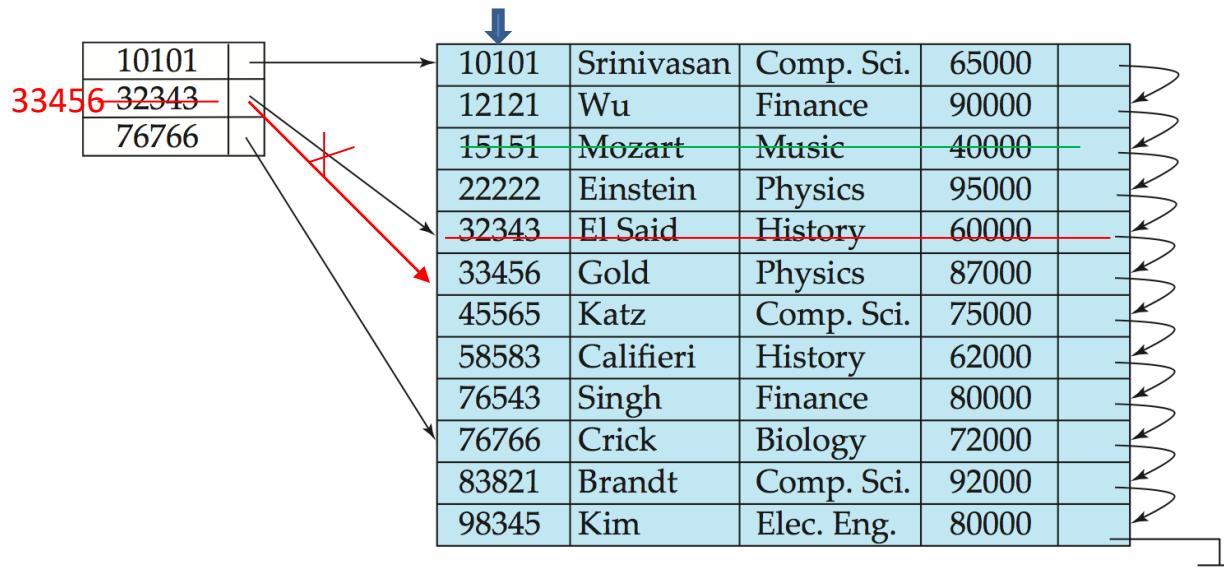
Index File Update After a Deletion in the Data File

▪ Single-level Index

- Dense Index, example:



- Sparse Index, example:



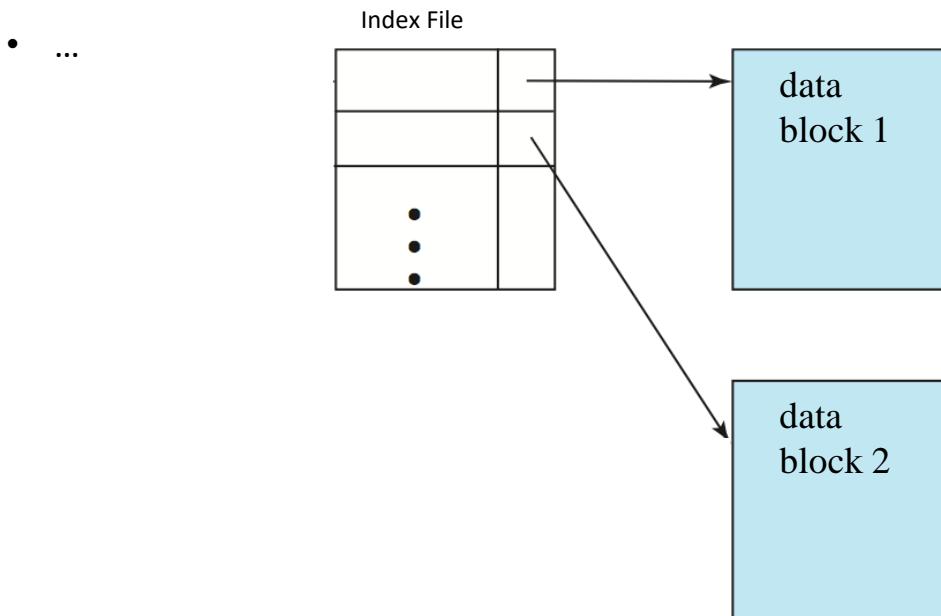
Sparse Index Compared to Dense Index

Sparse Index

- +: Less space and less maintenance overhead for INSERT and DELETE
- -: Generally slower than Dense Index for locating records.

Good tradeoff solution

- Sparse Index File with one index entry per block in the Data File:
 - 1st record in index points to the 1st record in the 1st block of the data file
 - 2nd record in index points to the 1st record in the 2nd block of the data file



Secondary Index Files

■ Secondary Index Files

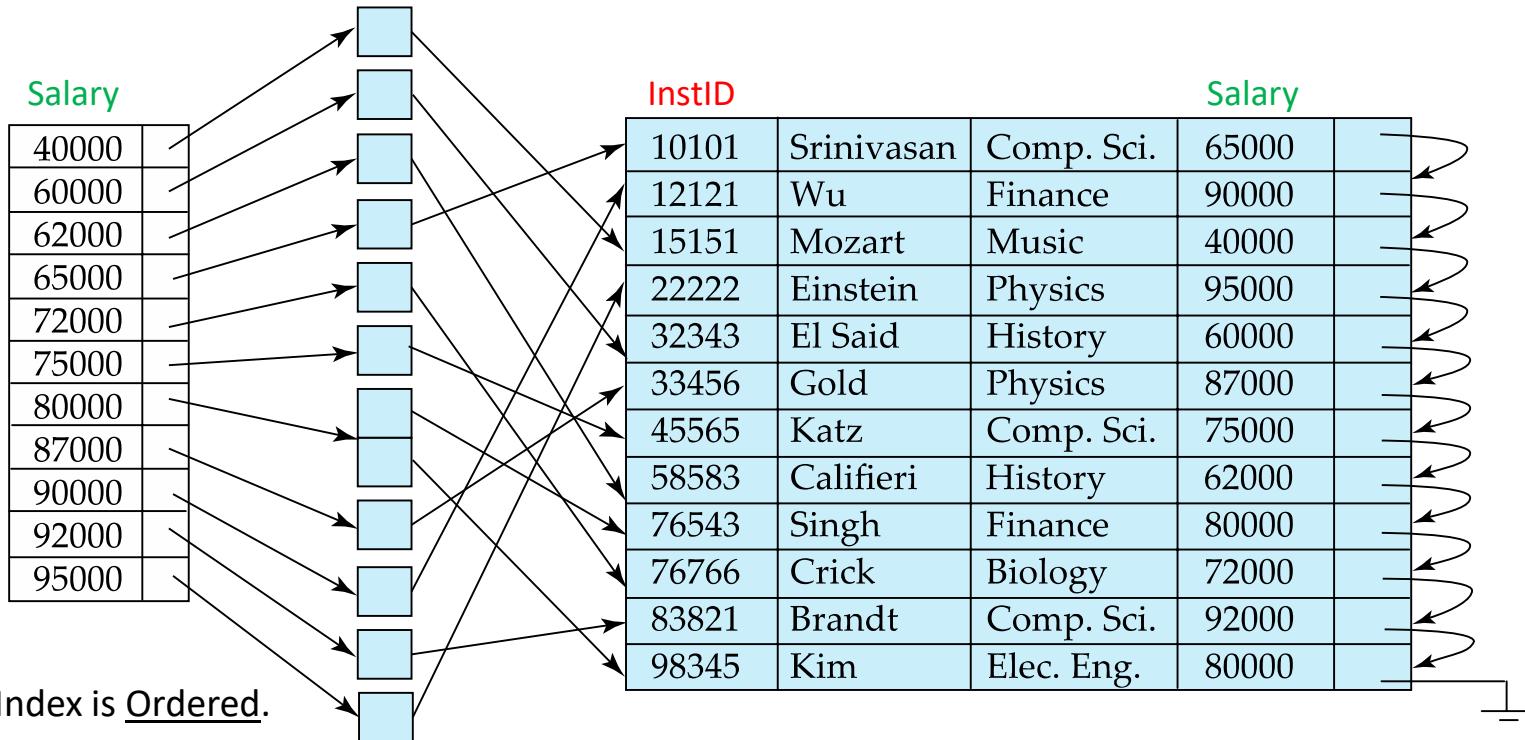
- Help finding all records whose values in a certain field satisfy some condition.
- The Instructor table is stored sequentially by InstID, finding all instructors in a particular department is easy with a Secondary Index on DeptName.
- A Secondary Index on Salary is helpful finding instructors with a specified salary or with a salary in a specified range of values.

■ Secondary Index Files and MySQL/MariaDB

- MySQL/MariaDB will automatically make a Primary Index for the Primary Key and Secondary Indexes for all Foreign Keys.

Secondary Index

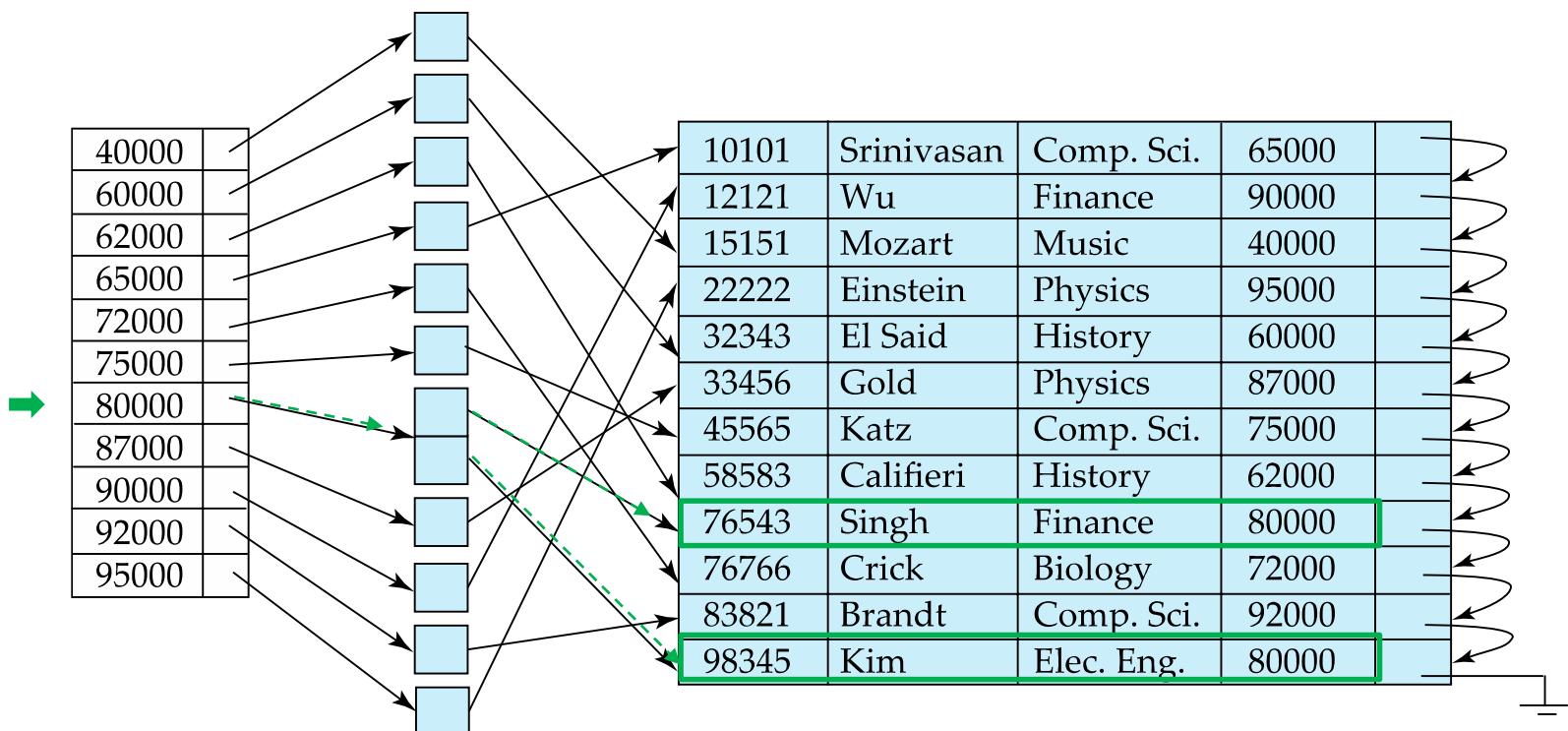
- Secondary Index on **non primary key**: Instructor.Salary (assume primary index on InstID)



- The Index is Ordered.
- The Index is Secondary (Index File and Data File are sorted on different Search Keys: **Salary** and **InstID**).
- Secondary Indexes are always Dense (All File Search Keys have an Index File Record).
- For each Search Key value K in the Index File, the associated pointer points to a bucket that contains pointers to all the data file records with that value for the Search Key. (If the index had been on a candidate key, buckets were not needed.)

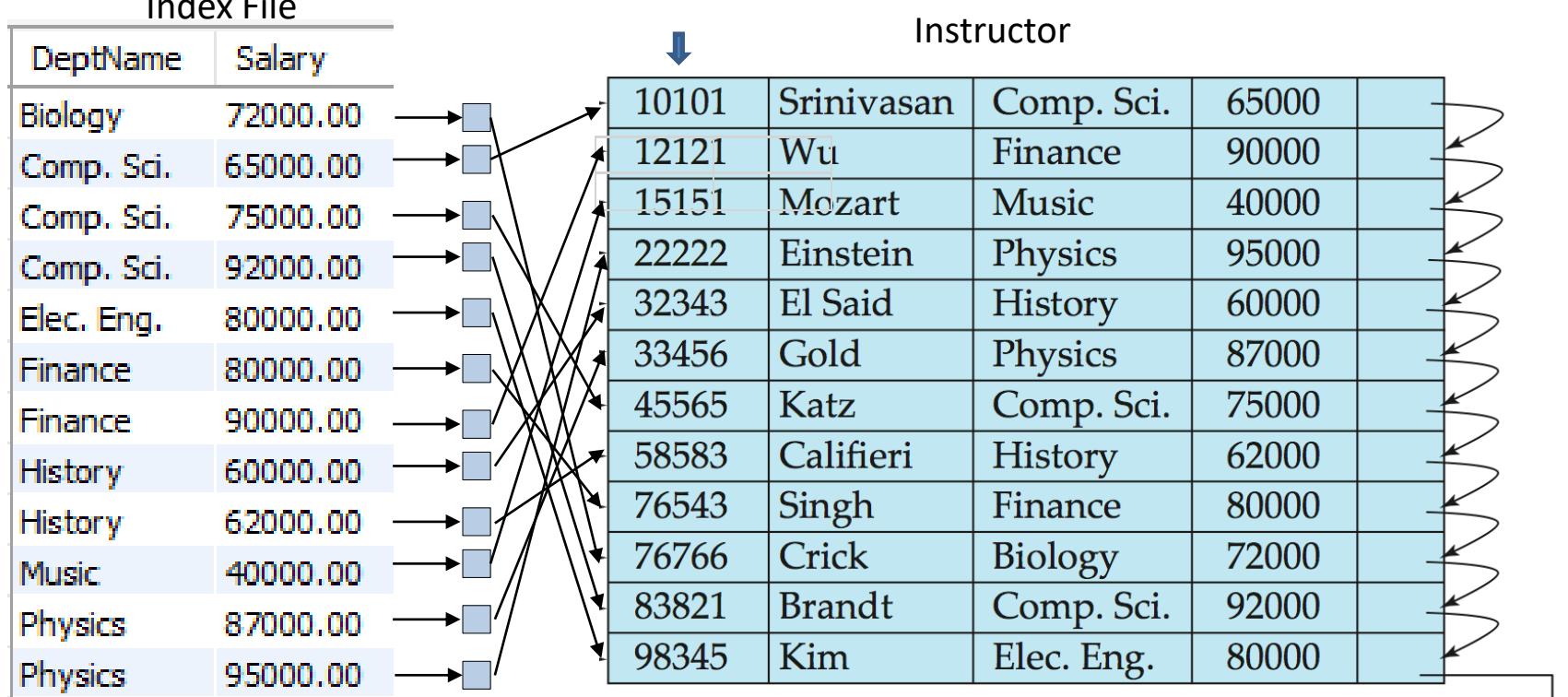
Querying Using a Secondary Index

- To locate a record with Search Key value K (e.g. Salary = 80000)
 - Find the record with Search Key value V = K in the Index File.
In practice: read the index into memory and use binary searching.
 - Follow the associated pointer to the associated bucket and follow each of the bucket pointers to a record in the Data File and read the record (block containing it) into main memory.



Multi Column Index

- has **Composite search keys** containing more than one attribute
 - E.g. a secondary index on instructor(DeptName, Salary) or a primary index on Classroom(Building, Room).
- Index file is lexicographically ordered: $(a_1, a_2) < (b_1, b_2)$ if either
 - $a_1 < b_1$, or $a_1 = b_1$ and $a_2 < b_2$



Searching Using a Multi Column Index

- If the table has a multiple-column index on (A1, A2, ..., An), it provides indexed search capabilities on:
 - (A1), (A1, A2), ..., and (A1, A2, ..., An).
- Example: having a secondary index on (DeptName, Salary) for Instructor facilitates queries like:
 - **SELECT ... FROM Instructor WHERE DeptName = “Finance” AND Salary = 80000**
 - **SELECT ... FROM Instructor WHERE DeptName = “Finance”**
- but not a query like
 - **SELECT ... FROM Instructor WHERE Salary = 80000**

Primary and Secondary, Ordered Index Files

- Substantial time saving benefits in searching for records
 - Searching using Primary, Ordered Index File is efficient.
 - Searching using Secondary, Ordered Index File is less efficient:
 - Each record access may fetch a new Block from disk!
- However
 - When a Data File is modified the Index Files must also be updated.
 - Updating Index Files imposes an overhead on database modification.

Multilevel Index

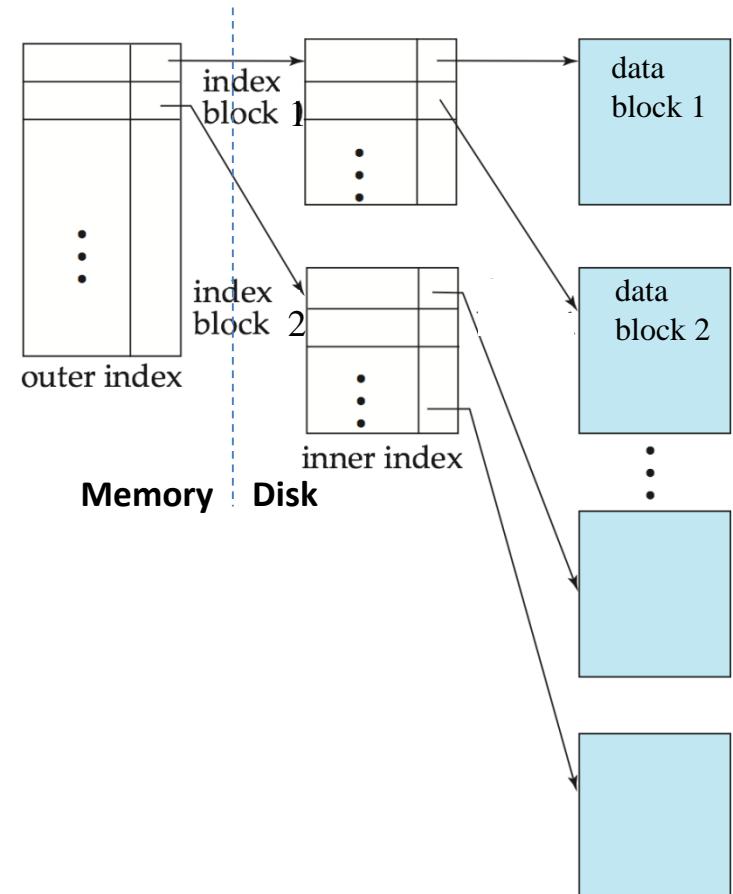
- When the Index File gets too big to fit in memory!

- Create a new index for the Index File!

- Solution: Build an Outer Index

- Inner index:
The original Index File
 - Outer index:
A Sparse Index of the Index File

- If the outer Index File gets too large to fit in memory, yet another level of index can be created.
 - However, indexes at all levels must be managed on record insertion or deletion.

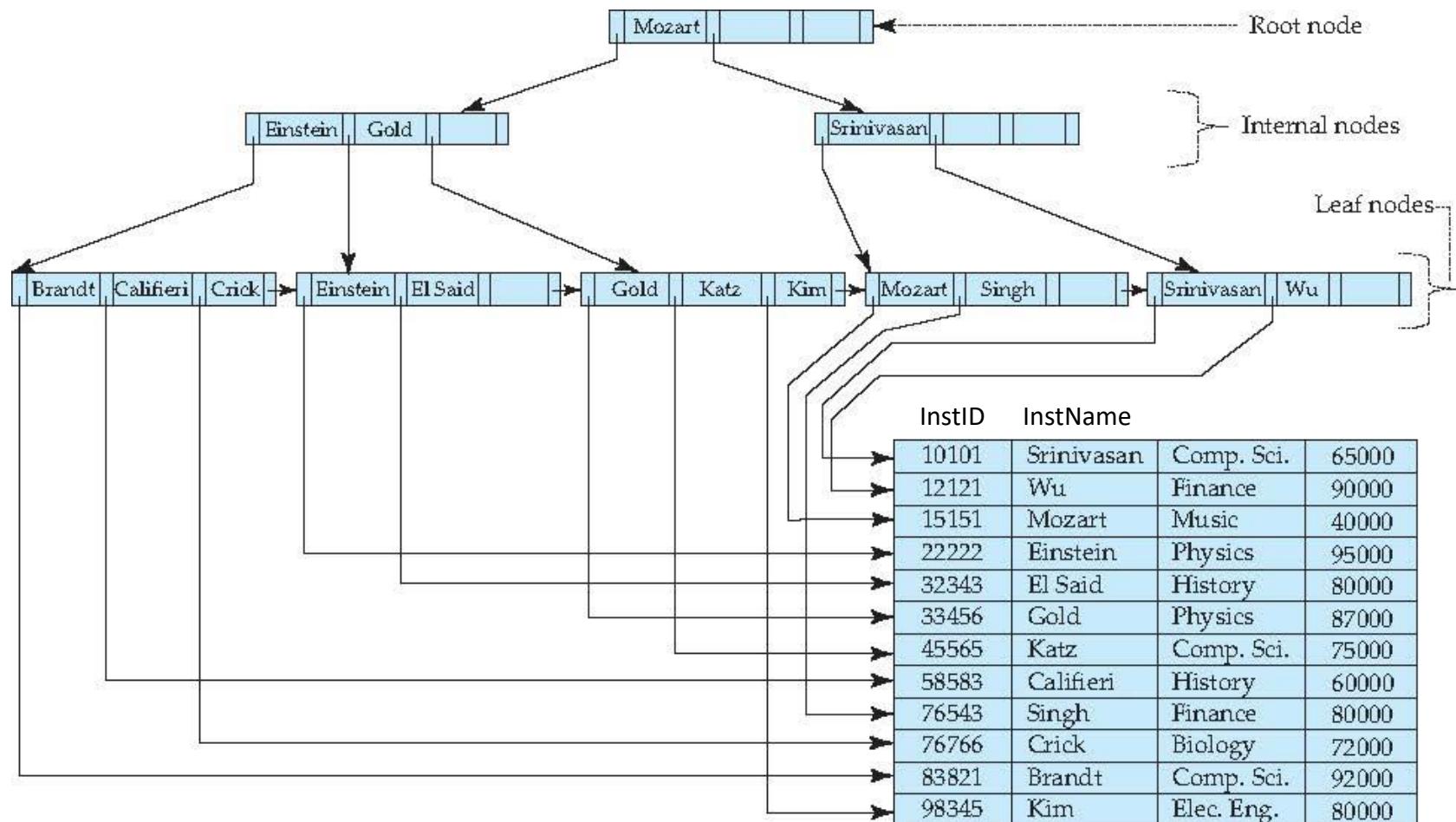


B⁺-Tree Index Files

- A *B+ Tree Index File* is a **dynamic multi-level index** structured as a *B+ tree*.
- *B+ Trees* are special cases of tree data structures.
- **Disadvantage of Index-Sequential Files**
 - Performance degrades as file grows (e.g. due to overflow blocks)
 - Periodic reorganization of the entire file is required
- **Advantage of a B⁺-Tree Index File**
 - Automatically reorganizes itself with small, local, changes after insertions and deletions.
 - Reorganization of the entire file is never required.
- **Minor disadvantage of B⁺-Trees**
 - Extra insertion and deletion overhead, space overhead
- **Advantages of B⁺-Trees outweigh disadvantages**
 - B⁺-Trees are used extensively in most database systems although they call them B-Trees!

Example of B⁺-Tree Index of Order 4 for Instructor with Search Key InstName

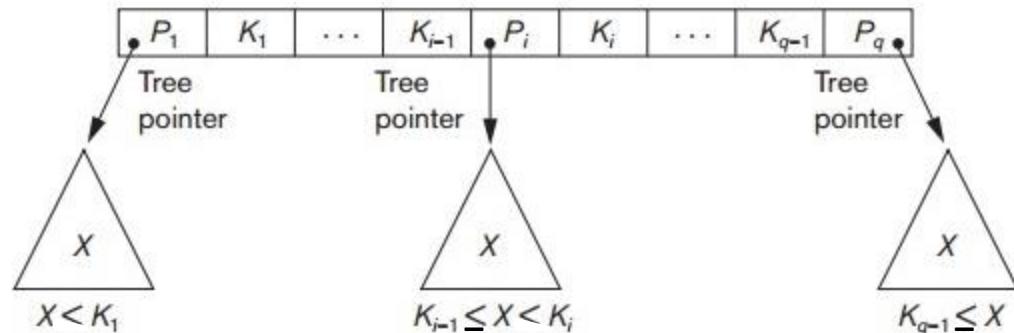
- Each Record has max 4 Pointers and max $4-1 = 3$ Search Keys in sorted order.
 - Leaf nodes form a Dense Secondary Index (ordered by InstName) to the records in the data file.
 - Non-leaf nodes at depth i form a Sparse Index to the nodes at depth $i + 1$.
 - Read record for El Said?



B⁺-Tree of Order n

- The B⁺-Tree constraints and properties

- Each node has one more pointer than search key values, except last leaf node.
- The tree must be sorted:
 - Keys must be in sorted order in each node: $K_1 < \dots < K_{q-1}$. (Assuming no duplicate search keys.)
 - The keys X in subtrees must satisfy:



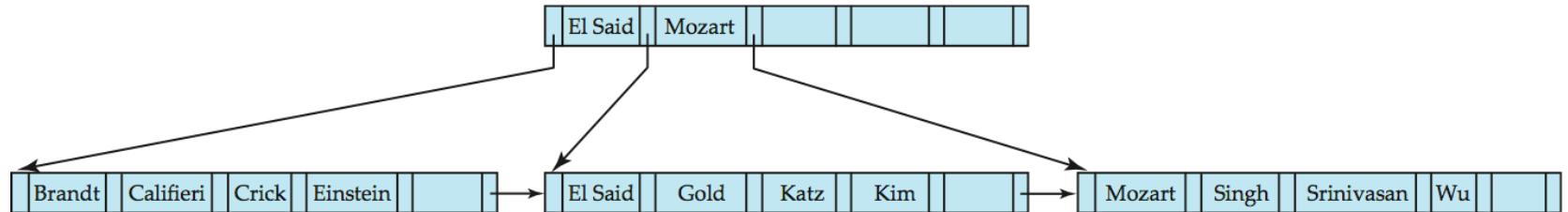
- The leaf nodes must be ordered, so $K_i < K_j$ for $K_i \rightarrow K_j$
- The tree must be balanced ("B" in the B⁺-Tree stands for Balanced), i.e. all paths from the root to a leaf are of the same length.
- Filling of non-root nodes:**
 - Each **internal node** has min $\lceil n/2 \rceil$ and max n pointers to subtrees / (children) and min $\lceil n/2 \rceil - 1$ and max $n - 1$ search key values .
 - A **leaf node** has min $\lceil (n - 1)/2 \rceil$ and max $(n - 1)$ search key values and pointers to records in the datafile.
 - Above $\lceil x \rceil$ denotes the least integer i for which $x \leq i$.

The basic idea is to leave room in the nodes for new search keys.

Example of B⁺-Tree of Order 6

- B⁺-Tree with $n = 6$, that is max 6 pointers in each node.
 - Leaf nodes must have between 3 and 5 search key values (i.e. between $\lceil (n - 1)/2 \rceil$ and $n - 1$).
 - Non-leaf nodes other than root must have 3 to 6 pointers/children (i.e. between $\lceil n/2 \rceil$ and n), and must have 2 to 5 search key values.

B⁺-Tree for *instructor* file ($n = 6$)



Queries on B⁺-Trees

- Find record with search-key value V . (Assuming no duplicate search values.)

$C = \text{root}$

while C is not a leaf node

let i be least value s.t. $V \leq K_i$ (so $K_{i-1} < V$)

if no such exists, set $C = \text{last non-null pointer in } C$

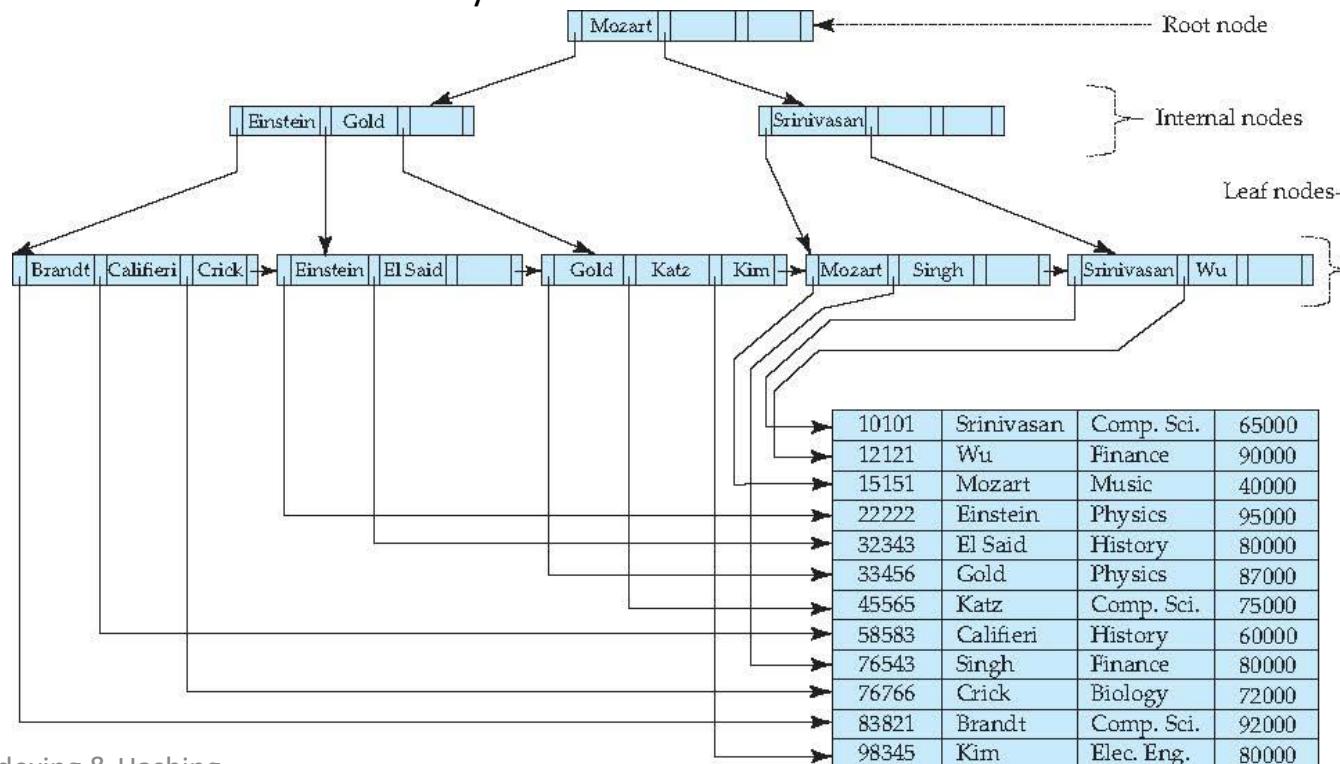
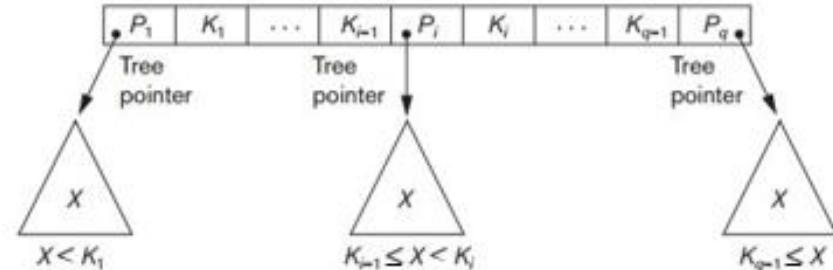
else if ($V = K_i$) set $C = P_{i+1}$

otherwise set $C = P_i$

let i be least value s.t. $K_i = V$

if there is such a value i , follow pointer P_i to the desired record.

else no record with search-key value k exists.



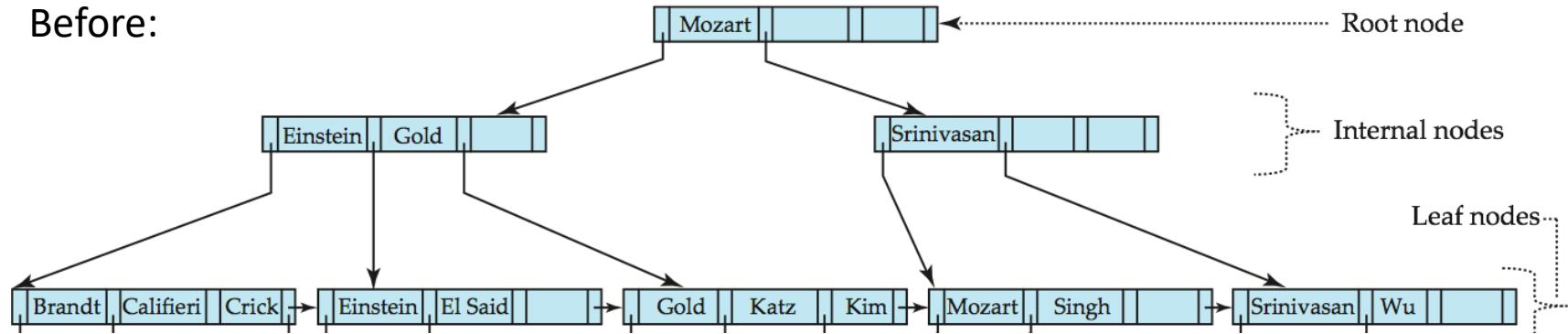
Queries on B⁺-Trees of order n

- If there are K search-key values in the file, the **height** of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size B as a disk block.
- Example:
 - If B is 4 kilobytes and the number bi of bytes per index entry is around 40 then n is typically around $B/bi = 100$.
 - With $K = 1 \text{ million}$ search key values and $n = 100$ at most $\log_{50}(1,000,000) = 4$ nodes are accessed in a lookup.
 - Contrast this with a *balanced binary tree* with 1 million search key values — around $\log_2(1,000,000) \sim 20$ nodes are accessed in a lookup.
 - Above difference is significant since **every node access may need a disk I/O**, costing typically around 10-20 milliseconds.

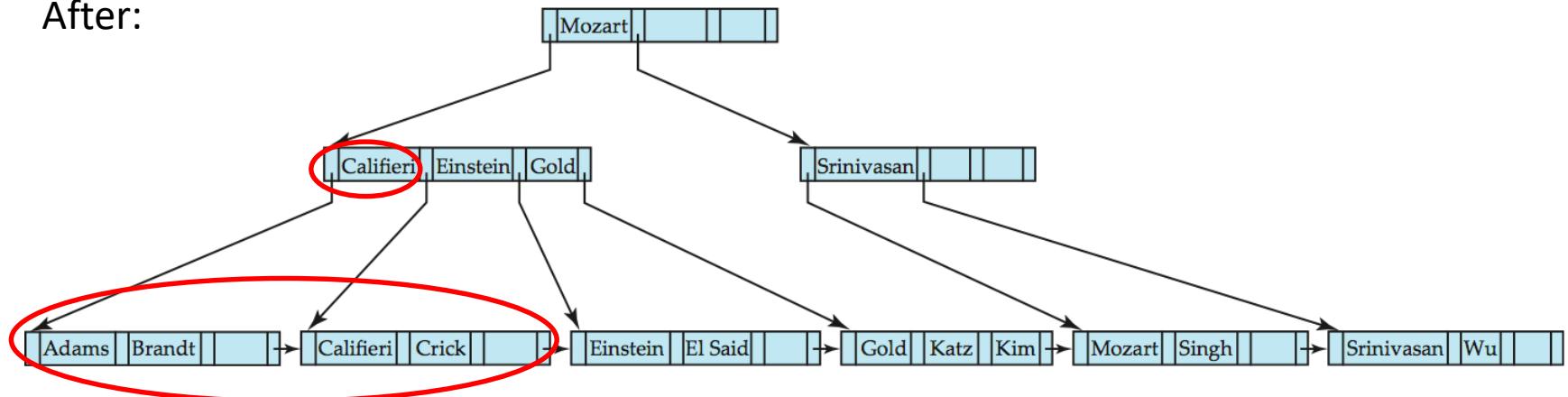
B⁺-Tree Insertion

B⁺-Tree before and after insertion of Adams!

Before:



After:



Observations about B⁺-Trees

▪ Major Observations

- The leaf nodes form a **Dense Index**.
- The non-leaf nodes form a **Sparse Index** to the leaf nodes.
- The B⁺-Tree is effective for queries and modification.

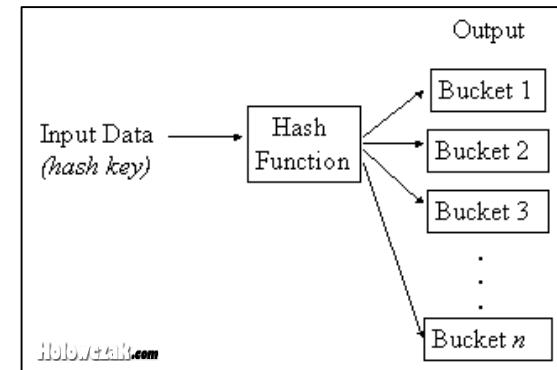
▪ Detailed Observations

- Contains a relatively small number of levels.
- This means that searches are conducted efficiently.
- Insertions and deletions to the data file can be handled efficiently, with a minimum overhead of index manipulation.

Static Hashing

Principles

- A **bucket** is a unit of storage containing one or more records, a bucket is typically a disk block.
- In hashing the bucket of a record is found directly: it is calculated from its search key value using a hash function.
- A **hash function** is a function mapping each search key value (also called a **hash key**) to a bucket number.



- The hash function is used to locate records for access, insertion and deletion:
Record with search key k is placed in bucket with number $\text{hash}(k)$.
- When hashing is used, the data file is said to be a **hash organized file**.
- Records with different search key values may be mapped to the same bucket; Thus the entire bucket has to be searched sequentially to locate a record.

Example of a Hash File Organization for Instructor Table

- Search key: DeptName (a text).
- Number of buckets: 8, numbered 0, 1, ..., 7.
- Hash function: $h(text) = (\text{int}(letter_1(text)) + \dots + \text{int}(letter_n(text))) \% 8$.
where int maps a letter to its alphabetic number.

$x \% 8$ will provide
buckets 0 to 7!

Example:

$$\begin{aligned} h(\text{Music}) &= \\ (13+21+19+9+3) \% 8 &= \\ 65 \% 8 &= 1 \end{aligned}$$

Examples:

- $h(\text{Music}) = 1$
- $h(\text{History}) = 2$
- $h(\text{Physics}) = 3$
- $h(\text{Elec. Eng.}) = 3$

Bucket 0			

Bucket 1			
15151	Mozart	Music	40000

Bucket 2			
32343	El Said	History	80000
58583	Califieri	History	60000

Bucket 3			
22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

Bucket 4			
12121	Wu	Finance	90000
76543	Singh	Finance	80000

Bucket 5			
76766	Crick	Biology	72000

Bucket 6			
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

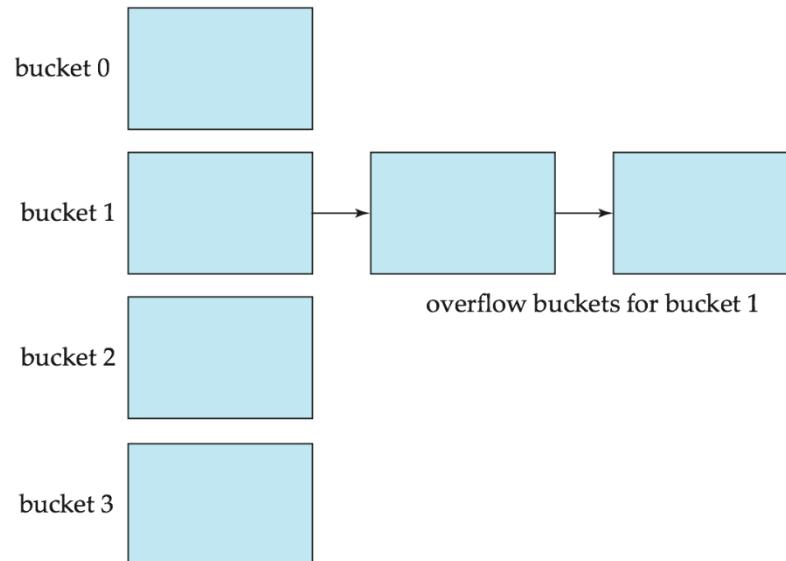
Bucket 7			

Hash Functions

- Good and bad hash functions
 - **Worst case of hashing:** all records are assigned to the same bucket.
 - **Ideal case of hashing:** each bucket will have the same number of records assigned (record distribution to buckets is uniform). Hard to achieve as the distribution of search keys in the records may change over time.
 - An **ideal hash function h**
 - is **uniform**, i.e. each bucket is assigned the same number of search key values from the set of all possible key values. Example: if there are 5 buckets and 20 search key values, the h should map 4 keys to each bucket.
 - is **random**, i.e. independent actual distribution of search key values in the file (so each bucket will have the same number of records assigned to it irrespective of the actual distribution of search key values in the file).
- Typical hash functions
 - Perform computation on the internal binary representation of the search key.
 - For example, for a string search key, the binary representations of all the characters in the string could be added and the remainder of the sum modulo the number of buckets could be returned.

Handling of Bucket Overflows

- Bucket overflow can occur
 - Insufficient number of buckets.
 - Skew in distribution of records due to
 - Multiple records have the same search key value.
 - Hash function produces non-uniform distribution of search key values.
- Overflow buckets
 - Although the probability of bucket overflow can be reduced, it cannot be eliminated; It is handled by using overflow buckets.

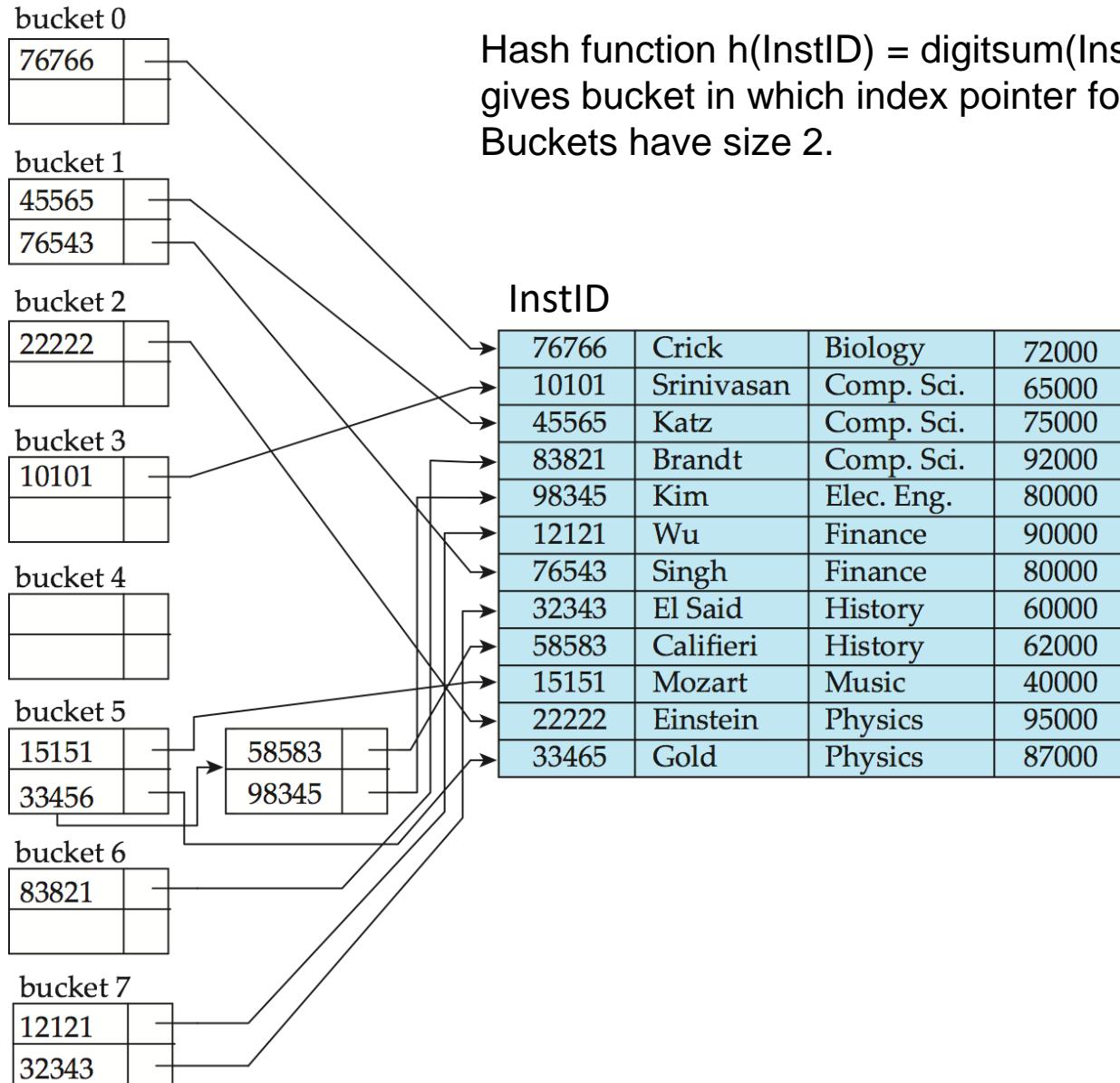


Hash Index Files

■ Hash Index File

- Hashing can also be used to make a stored **hash index file**,
i.e. the index file is organized as a hash file: HashIndexFile(Search Key, Pointer)
- Example: see next page.
- Hash Index Files are always Secondary Index Files.

Example of Hash Index on *instructor*, on attribute *InstID*



Hash function $h(\text{InstID}) = \text{digitsum}(\text{InstID}) \% 8$
gives bucket in which index pointer for InstID is kept.
Buckets have size 2.

Deficiencies of Static Hashing

- **Static hashing**
 - The hash function maps the search key values to a fixed set of bucket addresses. However, databases grow or shrink with time!
 - If the number of buckets is too small, and the file grows, performance will degrade due to too much overflow.
 - If the number of buckets is too large, space will be wasted.
- **One solution**
 - Periodic re-organization of the file with a new hash function.
 - Expensive and disrupts normal operations.
- **Better solution**
 - Use **Dynamic Hashing** allowing the number of buckets to be modified dynamically. (If you wish to know how, read section 11.7.)

Hashing in MySQL/MariaDB

- Can declare a table to be stored in a hash organized file, like:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    job_code INT,
    store_id INT )
PARTITION BY HASH(store_id) PARTITIONS 4;
```

A record with store_id = 2005, will be placed in partition number $2005 \% 4 = 1$

- Can declare a hash index on a table, like:

```
CREATE INDEX NameIndex ON Instructor(InstName) USING HASH;
```

Choosing Indexes/Hashing

1. List Search Key Candidates (to potentially define indexes/hashing on)

- Primary keys (many DBMS automatically choose primary indexes on these)
- Foreign keys (many DBMS automatically choose secondary indexes on these)
- Attributes A_i that are often used for searching and sorting
(appear e.g. in join conditions, WHERE conditions, ORDER BY, ...)

2. Select which of the candidates should have an index/hash access

- Use database statistics about frequencies of queries and data modifications, and table sizes.
- Discuss with domain experts.
- Never define indexes for small tables or for attributes that are often updated.
- Trade off between
 - (1) shorter time for queries (SELECT)
 - (2) longer time for data modifications (INSERT, DELETE, UPDATE)
 - (3) space overhead for index data structures

3. Hashing versus indexing on A_i

- If **SELECT ...**, A_i ... **FROM** table **WHERE** $A_i = c$ is common
hashing on A_i is better.
- If *range queries* like **SELECT ... FROM** table **WHERE** $A_i \leq c_2$ **AND** $A_i \geq c_1$ are common, then **sequentially ordered indexes** and **B+ trees** are better.

It is all also a matter of your DBMS.



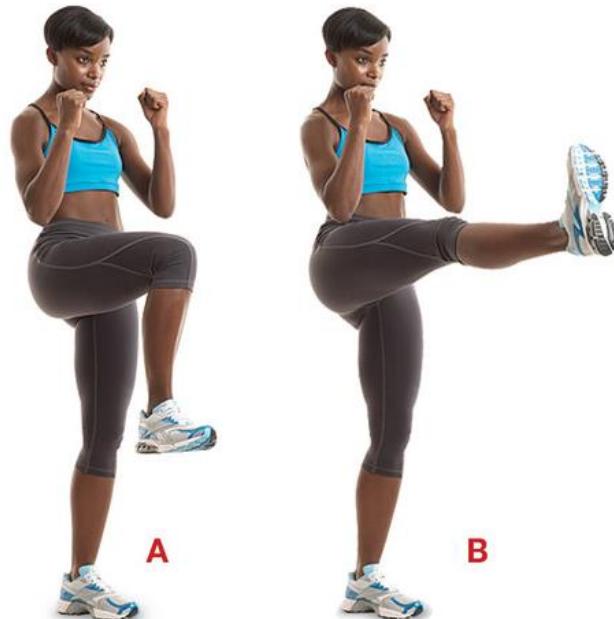
Summary

- How to avoid reading a data file sequentially to find a record during a query?
 1. Use **index files**. An index file is an additional file that contains information about where records are stored in a data file.
Examples of index file organizations: sequential, B⁺-trees, hash.
 2. Use **hashing**. Uses a hash function instead of an index file to locate a record in the data file.
- What to choose is a tradeoff between (1) access time and (2) insertion time, deletion time and space overhead.

Readings

- In Database Systems Concepts please read Chapter 11.1-3 (11.3.3-11.3.4 can be skipped), 11.5-6, 11.8, 11.10-11.
- Pay special attention to the Summary

Demo Exercises



Demo Exercises clarify ideas and concepts from the Lecture to provide you with good Database Skills.

Do the
Demo Exercises.

Ordered Indexes

Student Table for 11.1 and 11.2,

StudID	StudName	Birth	DeptName	TotCredits
00128	Zhang	1992-04-18	Comp. Sci.	102
12345	Shankar	1995-12-06	Comp. Sci.	32
19991	Brandt	1993-05-24	History	80
23121	Chavez	1992-04-18	Finance	110
44553	Peltier	1995-10-18	Physics	56
45678	Levy	1995-08-01	Physics	46
54321	Williams	1995-02-28	Comp. Sci.	54
55739	Sanchez	1995-06-04	Music	38
70557	Snow	1995-11-22	Physics	0
76543	Brown	1994-03-05	Comp. Sci.	58

11.1 Ordered, Primary and Dense Index

Make an Ordered, Primary and Dense Index for the Student table assuming a maximum of 4 records in each block.

(Instead of drawing the pointer, you can give the address it is pointing to, by a block# and the record# in that block.)

11.2 Ordered, Primary and Sparse Index

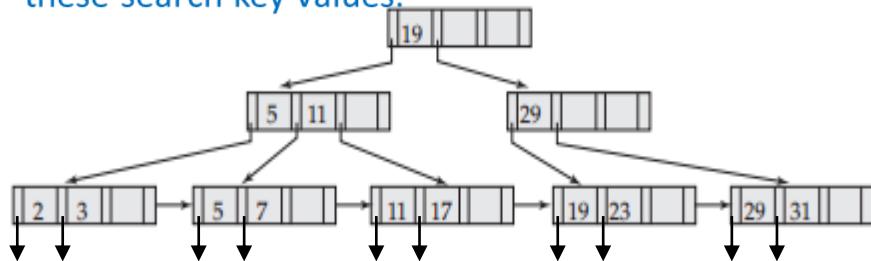
Make an Ordered, Primary and Sparse Index for the Student table assuming a maximum of 3 records in each Block.

(Instead of drawing the pointer, you can give the address it is pointing by a block# and the record# in that block.)

B⁺-Tree Indexes

11.3 B⁺-Tree

A file has the following set of search key values: (2, 3, 5, 7, 11, 17, 19, 23, 29, 31). Consider the following B⁺-Tree used as an index containing these search key values.



Show the steps involved in the two following queries:

- Find records with a search-key value of 11.
- Find records with a search-key value between 7 and 17, inclusive.

Hashing

11.4 Hash File Organization (Hashed Data File)

Suppose that hashing (a hash file organization) with search key A should be used for a data file of a relation R(A,B). There are 8 buckets numbered 0-7, each can hold max 3 records.

The hash function is: $h(x) = x \% 8$.

a) Show the contents of the 8 buckets after insertion of the following records of (A,B) pairs:

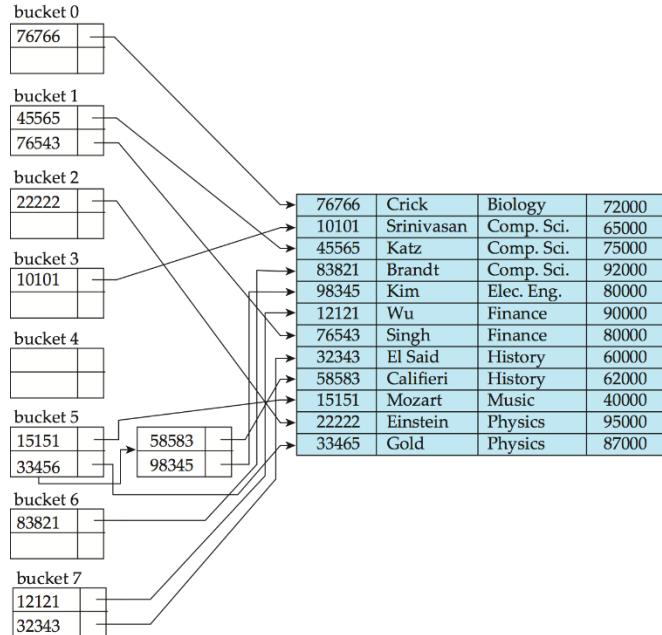
(2, X), (3, H), (5, P), (7, P), (11, Y), (17, C),

(19, D), (23, F), (29, X), (31, J)

b) After that, where should record (15, H) be added

Hashing

11.5 Hash Index (Hashed Index File)



Consider the hash index on InstID for Instructor, shown above. It uses hash function where e.g. $h(58583) = 5$.

Show the steps involved in the query “find the record with InstID = 58583”:

SELECT * FROM Instructor WHERE InstID = 58583;

Solutions to Demo Exercises



Ordered Indexes

Student Table for 11.1 and 11.2,

StudID	StudName	Birth	DeptName	TotCredits
00128	Zhang	1992-04-18	Comp. Sci.	102
12345	Shankar	1995-12-06	Comp. Sci.	32
19991	Brandt	1993-05-24	History	80
23121	Chavez	1992-04-18	Finance	110
44553	Peltier	1995-10-18	Physics	56
45678	Levy	1995-08-01	Physics	46
54321	Williams	1995-02-28	Comp. Sci.	54
55739	Sanchez	1995-06-04	Music	38
70557	Snow	1995-11-22	Physics	0
76543	Brown	1994-03-05	Comp. Sci.	58

11.1 Ordered, Primary and Dense Index

Make an Ordered, Primary and Dense Index for the Student table assuming a maximum of 4 records in each block.

(Instead of drawing the pointer, you can give the address it is pointing to, by a block# and the record# in that block.)

11.2 Ordered, Primary and Sparse Index

Make an Ordered, Primary and Sparse Index for the Student table assuming a maximum of 3 records in each Block.

(Instead of drawing the pointer, you can give the address it is pointing by a block# and the record# in that block.)

11.1 StudentIndex(Search Key, Block#, Record#)

Ordered:

Index is a sorted list

Primary:

Index and file are sorted on the same Search Key

Dense:

All file records have an index record.

Search Key	Block#	Record#
00128	1	1
12345	1	2
19991	1	3
23121	1	4
44553	2	1
45678	2	2
54321	2	3
55739	2	4
70557	3	1
76543	3	2

11.2 StudentIndex(Search Key, Block#, Record#)

Ordered:

Index is a sorted list

Primary:

Index and file are sorted on the same Search Key

Sparse:

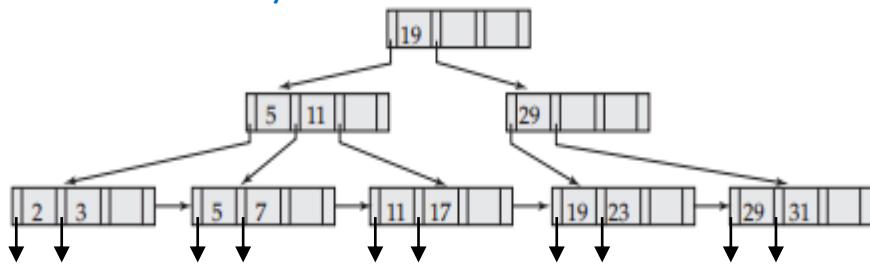
Only first record from each block record.

Search Key	Block#	Record#
00128	1	1
23121	2	1
54321	3	1
76543	4	1

B⁺-Tree Indexes

11.3 B⁺-Tree

A file has the following set of search key values: (2, 3, 5, 7, 11, 17, 19, 23, 29, 31). Consider the following B⁺-Tree used as an index containing these search key values.



Show the steps involved in the two following queries:

- Find records with a search-key value of 11.
- Find records with a search-key value between 7 and 17, inclusive.

- Find records with a value of 11
 - Search the first level index; follow the first pointer.
 - Search next level; follow the third pointer.
 - Search leaf node; follow first pointer to records with key value 11.
- Find records with value between 7 and 17, inclusive.
 - Search top index; follow first pointer.
 - Search next level; follow second pointer.
 - Search third level; follow second pointer to records with key value 7, and after accessing them, return to leaf node.
 - Follow fourth pointer to next leaf block in the chain.
 - Follow first pointer to records with key value 11, then return.
 - Follow second pointer to records with key value 17.

In this way 3 records are found.

Hashing

11.4 Hash File Organization (Hashed Data File)

Suppose that hashing (a hash file organization) with search key A should be used for a data file of a relation R(A,B). There are 8 buckets numbered 0-7, each can hold max 3 records.

The hash function is: $h(x) = x \% 8$.

a) Show the contents of the 8 buckets after insertion of the following records of (A,B) pairs:

(2, X), (3, H), (5, P), (7, P), (11, Y), (17, C),
(19, D), (23, F), (29, X), (31, J)

b) After that, where should record (15, H) be added?

11.4 Hash File Organization (Hashed Data File)

x	2	3	5	7	11	17	19	23	29	31
$h(x)$	2	3	5	7	3	1	3	7	5	7

Bucket 1 contains (17, C)

Bucket 2 contains (2, X)

Bucket 3 contains (3, H), (11, Y), (19, D)

Bucket 5 contains (5, P), (29, X)

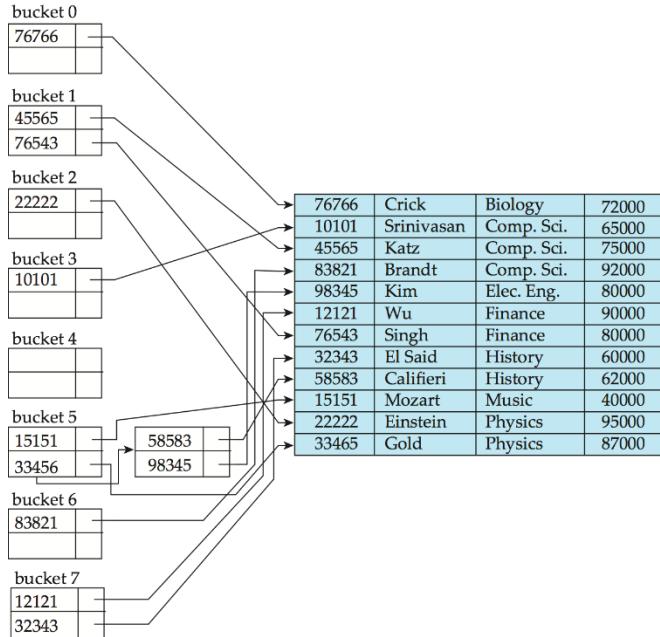
Bucket 7 contains (7, P), (23, F), (31, J)

Buckets 0, 4, and 6 are empty.

b) (15, H) should be added to bucket 7, but there is no space as it already has 3 records. The record must be added in an **overflow bucket** for bucket 7.

Hashing

11.5 Hash Index (Hashed Index File)



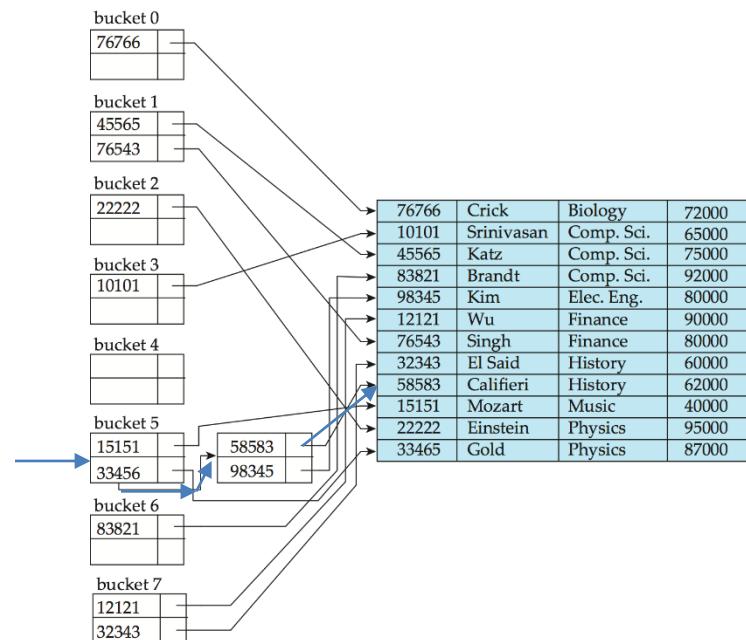
Consider the hash index on InstID for Instructor, shown above. It uses hash function where e.g. $h(58583) = 5$.

Show the steps involved in the query “find the record with InstID = 58583”:

SELECT * FROM Instructor WHERE InstID = 58583;

11.5 Hash Index (Hashed Index File)

$h(58583) = 5$, so the index pointer for 58583 is searched for in bucket 5. It is not in the main part of bucket 5, so the pointer to the overflow bucket of bucket 5 is followed and the searching is continued in that bucket. The record is found as the first. The pointer of this record is followed to the record in the data file having InstID = 58583 (and then all its attributes are selected).



Exercises



Please answer all exercises
to demonstrate your
Database Skills.

Pencil and Paper Exercises
Solutions are available at 11:45

Indexes

Student Table

StudID	StudName	Birth	DeptName	TotCredits
00128	Zhang	1992-04-18	Comp. Sci.	102
12345	Shankar	1995-12-06	Comp. Sci.	32
19991	Brandt	1993-05-24	History	80
23121	Chavez	1992-04-18	Finance	110
44553	Peltier	1995-10-18	Physics	56
45678	Levy	1995-08-01	Physics	46
54321	Williams	1995-02-28	Comp. Sci.	54
55739	Sanchez	1995-06-04	Music	38
70557	Snow	1995-11-22	Physics	0
76543	Brown	1994-03-05	Comp. Sci.	58

11.6 Ordered, Secondary and Dense Index

Make a Sequentially Ordered, Secondary and Dense Index for the Student table with Search Key DeptName.

Assume that the data file has 4 records in each block.

11.7 B⁺-Tree Index

A file has the following set of search key values: (2, 3, 5, 7, 11, 17, 19, 23, 29, 31). Consider the following B⁺-Tree used as an index containing these search key values.



- What is the order n of the B⁺ Tree?
- What is the allowed number of search keys in the root and in a non-root node in a B⁺ tree of that order?
- Show the steps involved in the two following queries:
 - Find records with a search-key value of 11.
 - Find records with a search-key value between 7 and 17, inclusive.

Hashing

11.8 Hashed Data File

Make a hash file organization for the Student table with Search Key StudID, assuming that records are inserted in the order shown in the table. There should be 4 buckets, which each can hold max 5 records.

What would the result be, if there could only be 4 records in a bucket?

11.9 Hash Index (Hashed Index File)

Make a hash index for the Student table with Search Key StudID, assuming that records are inserted in the order shown in the table. There should be 4 buckets, which each can hold max 5 records.