



## CS425 – Fall 2014 Boris Glavic Course Information

Modified from:  
Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



Hi, I am Boris Glavic,  
Assistant Professor in  
CS



CS425 – Fall 2014 – Boris Glavic

0.2

©Silberschatz, Korth and Sudarshan



Hi, I am Boris Glavic,  
Assistant Professor in  
CS

I am a database guy!

CS425 – Fall 2014 – Boris Glavic

0.3

©Silberschatz, Korth and Sudarshan



Hi, I am Boris Glavic,  
Assistant Professor in  
CS

I am a database guy!

I will teach you:  
database stuff

CS425 – Fall 2014 – Boris Glavic

0.4

©Silberschatz, Korth and Sudarshan



## Why are Databases Important?

- What do Databases do?
  1. Provide persistent storage
  2. Efficient declarative access to data -> Querying
  3. Protection from hardware/software failures
  4. Safe concurrent access to data

CS425 – Fall 2014 – Boris Glavic

0.5

©Silberschatz, Korth and Sudarshan



## Who uses Databases?

- Most big software systems involve DBs!
  - Business Intelligence ⇒ e.g., IBM Cognos
  - Web based systems
  - ...
- You! (desktop software)
  - Your music player ⇒ e.g., Amarok
  - Your Web Content Management System
  - Your email client
  - ...
- Every big company
  - Banks
  - Insurance
  - Government
  - Google, ...
  - ...



CS425 – Fall 2014 – Boris Glavic

0.6

©Silberschatz, Korth and Sudarshan



## Who Produces Databases?

- **Traditional relational database systems is big business**
  - IBM ⇒ DB2
  - Oracle ⇒ Oracle ©
  - Microsoft ⇒ SQLServer
  - Open Source ⇒ MySQL, Postgres, ...
- **Emerging distributed systems with DB characteristics and Big Data**
  - Cloud storage and Key-value stores ⇒ Amazon S3, Google Big Table, ...
  - Big Data Analytics ⇒ Hadoop, Google Map & Reduce, ...
  - SQL over Distributed Platforms ⇒ Hive, Tenzing, ...



CS425 – Fall 2014 – Boris Glavic

0.7

©Silberschatz, Korth and Sudarshan



## Why are Database Interesting (for Students)?

- **The pragmatic perspective**
  - Background in databases make you competitive in the job market :-)
- **Systems and theoretical research**
  - Database research has a strong systems aspect
    - ▶ Hacking complex and large systems
    - ▶ Low-level optimization
      - cache-conscious algorithms
      - Exploit modern hardware
  - Databases have a strong theoretical foundation
    - ▶ Complexity of query answering
    - ▶ Expressiveness of query languages
    - ▶ Concurrency theory
    - ▶ ...

CS425 – Fall 2014 – Boris Glavic

0.8

©Silberschatz, Korth and Sudarshan



## Why are Database Interesting (for Students)?

- **Connection to many CS fields**
  - Distributed systems
    - ▶ Getting more and more important
  - Compilers
  - Modeling
  - AI and machine learning
    - ▶ Data mining
  - Operating and file systems
  - Hardware
    - ▶ Hardware-software co-design

CS425 – Fall 2014 – Boris Glavic

0.9

©Silberschatz, Korth and Sudarshan



## Webpage and Faculty

- **Course Info**
  - **Course Webpage:** <http://cs.iit.edu/~cs425>
  - **Google Group:** <https://groups.google.com/d/forum/cs425-2014-fall-group>
    - ▶ Used for announcements
    - ▶ Use it to discuss with me, TA, and fellow students
  - **Syllabus:** <http://cs.iit.edu/~cs425/files/syllabus.pdf>
- **Faculty**
  - **Boris Glavic** (<http://cs.iit.edu/~glavic>)
  - **Email:** [bjglavic@iit.edu](mailto:bjglavic@iit.edu)
  - **Phone:** 312.567.5205
  - **Office:** Stuart Building, room 226C
  - **Office Hours:** Tuesdays, 12pm-1pm (and by appointment)

CS425 – Fall 2014 – Boris Glavic

0.10

©Silberschatz, Korth and Sudarshan



## TAs

- **Tas**
  - **TBA**

CS425 – Fall 2014 – Boris Glavic

0.11

©Silberschatz, Korth and Sudarshan



## Workload and Grading

- **Exams**
  - Midterm (25%)
  - Final (35%)
- **Homework Assignments** (preparation for exams)
  - HW1 (Relational algebra)
  - HW2 (SQL)
  - HW3 (Database modeling)
- **Course Project**
  - In groups of 3 students
  - Given an example application (e.g., ticketing system)
    - ▶ Develop a database model
    - ▶ Derive a database schema from the model
    - ▶ Implement the application accessing the database

CS425 – Fall 2014 – Boris Glavic

0.12

©Silberschatz, Korth and Sudarshan



## Course Objectives

- Understand the underlying ideas of database systems
- Understand the **relational data model**
- Be able to write and understand **SQL** queries and data definition statements
- Understand **relational algebra** and its connection to SQL
- Understand how to **write programs that access a database server**
- Understand the **ER model** used in database design
- Understand **normalization** of database schemata
- Be able to **create a database design** from a requirement analysis for a specific domain
- Know basic **index structures** and understand their importance
- Have a basic understanding of relational database concepts such as **concurrency control, recovery, query processing, and access control**

CS425 – Fall 2014 – Boris Glavic

0.13

©Silberschatz, Korth and Sudarshan



## Course Project

- Forming groups
  - Your responsibility!
  - Inform me + TA
  - Deadline: Sep 8th
- Oracle Server Accounts
- Git repositories
  - Create an account on Bitbucket.org (<https://bitbucket.org>)
  - We will create a repository for each student
  - Use it to exchange code with your fellow group members
  - The project has to be submitted via the group repository
- Timeline:
  - Brainstorming on application (by Sep 11<sup>th</sup>)
  - Design database model (by Nov 12<sup>th</sup>)
  - Derive relational model (by Nov 25<sup>th</sup>)
  - Implement application (by end of the semester)

CS425 – Fall 2014 – Boris Glavic

0.14

©Silberschatz, Korth and Sudarshan



## Fraud and Late Assignments

- All work has to be original!
  - Cheating = 0 points for assignment/exam
  - Possibly E in course and further administrative sanctions
  - Every dishonesty will be reported to office of academic honesty
- Late policy:
  - -20% per day
  - No exceptions!
- Course projects:
  - Every student has to contribute in **every** phase of the project!
  - **Don't let others freeload on you hard work!**
    - ▶ Inform me or TA immediatly

CS425 – Fall 2014 – Boris Glavic

0.15

©Silberschatz, Korth and Sudarshan



## Reading and Prerequisites

- **Textbook:** Silberschatz, Korth and Sudarshan
  - **Database System Concepts, 6<sup>th</sup> edition**
  - McGraw Hill
  - publication date:2006,
  - ISBN 0-13-0-13-142938-8.
- Prerequisites:
  - CS 331 or CS401 or CS403

CS425 – Fall 2014 – Boris Glavic

0.16

©Silberschatz, Korth and Sudarshan



## Outline

- Introduction
- Relational Data Model
- Formal Relational Languages (relational algebra)
- SQL
- Database Design
- Transaction Processing, Recovery, and Concurrency Control
- Storage and File Structures
- Indexing and Hashing
- Query Processing and Optimization

CS425 – Fall 2014 – Boris Glavic

0.17

©Silberschatz, Korth and Sudarshan



# **CS425 – Fall 2016**

## **Boris Glavic**

### **Chapter 1: Introduction**

**Modified from:**

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

©Silberschatz, Korth and Sudarshan

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





# Textbook: Chapter 1



# Database Management System (DBMS)

- DBMS contains information about a particular domain
  - Collection of interrelated data
  - Set of programs to access the data
  - An environment that is both *convenient* and *efficient* to use
- Database Applications:
  - Banking: transactions
  - Airlines: reservations, schedules
  - Universities: registration, grades
  - Sales: customers, products, purchases
  - Online retailers: order tracking, customized recommendations
  - Manufacturing: production, inventory, orders, supply chain
  - Human resources: employee records, salaries, tax deductions
- Databases can be very large.
- Databases touch all aspects of our lives



# University Database Example

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



# Drawbacks of using file systems to store data

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



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

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

**Database systems offer solutions to all the above problems!**



# Levels of Abstraction

- **Physical level:** describes how a record (e.g., customer) is stored.
- **Logical level:** describes data stored in database, and the relationships among the data.

```
type instructor = record
```

```
    ID : string;
```

```
    name : string;
```

```
    dept_name : string;
```

```
    salary : integer;
```

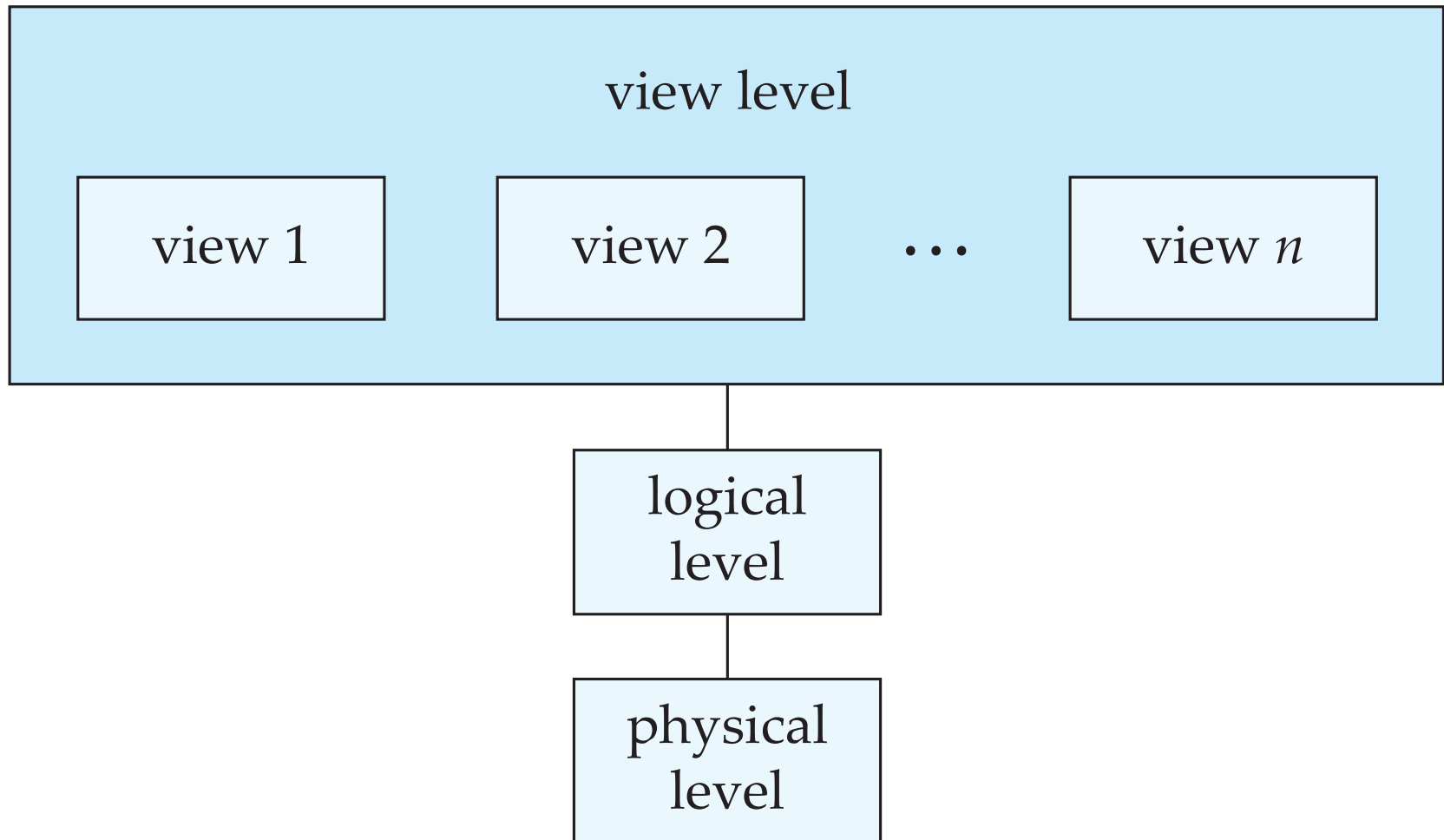
```
end;
```

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



# View of Data

An architecture for a database system





# Instances and Schemas

- Similar to types and variables in programming languages
- **Schema** – the logical structure of the database
  - Example: The database consists of information about a set of customers and accounts and the relationship between them
  - Analogous to type information of a variable in a program
  - **Physical schema**: database design at the physical level
  - **Logical schema**: database design at the logical level
- **Instance** – the actual content of the database at a particular point in time
  - Analogous to the value of a variable
- **Physical Data Independence** – the ability to modify the physical schema without changing the logical schema
  - Applications depend on the logical schema
  - In general, the interfaces between the various levels and components should be well defined so that changes in some parts do not seriously influence others.
- **Logical Data Independence** – the ability to modify the logical schema without changing the applications
  - For example, add new information to each employee





# Data Models

- A collection of tools for describing
  - Data
  - Data relationships
  - Data semantics
  - Data constraints
- Relational model
- Entity-Relationship data model (mainly for database design)
- Object-based data models (Object-oriented and Object-relational)
- Semistructured data model (XML)
- Other older models:
  - Network model
  - Hierarchical model
- Other newer (or revived) models:
  - Key-value



# Relational Model

- Relational model (Chapter 2)
- Example of tabular data in the relational model

Columns (attributes)

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

Rows (tuples)

(a) The *instructor* table



# A Sample Relational Database

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

(a) The *instructor* table

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

(b) The *department* table



# Data Manipulation Language (DML)

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



# Data Definition Language (DDL)

- Specification notation for defining the database schema

Example:       **create table** *instructor* (  
                  *ID*           **char**(5),  
                  *name*       **varchar**(20),  
                  *dept\_name* **varchar**(20),  
                  *salary*     **numeric**(8,2))

- DDL compiler generates a set of table templates stored in a *data dictionary*
- Data dictionary contains **metadata** (i.e., data about data)
  - Database schema
  - **Integrity constraints**
    - ▶ Primary key (ID uniquely identifies instructors)
    - ▶ Referential integrity (**references** constraint in SQL)
      - e.g. *dept\_name* value in any *instructor* tuple must appear in *department* relation
  - Authorization



# SQL

- **SQL**: widely used **declarative** (non-procedural) language
  - Example: Find the name of the instructor with ID 22222

```
select  name
from    instructor
where   instructor.ID = '22222'
```
  - Example: Find the ID and building of instructors in the Physics dept.

```
select instructor.ID, department.building
from   instructor, department
where  instructor.dept_name = department.dept_name and
         department.dept_name = 'Physics'
```
  
- Application programs generally access databases through one of
  - Language extensions to allow embedded SQL
  - Application program interface (e.g., ODBC/JDBC) which allow SQL queries to be sent to a database
  
- Chapters 3, 4 and 5



# Database Design

The process of designing the general structure of a database:

- Logical Design – Deciding on the database schema. Database design requires that we find a “good” representation of the information from an application domain (e.g., banking) as a collection of relation schemas.
  - Business decision – What information should we record in the database?
  - Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?
  
- Physical Design – Deciding on the physical layout of the database



# Database Design?

- Is there any problem with this design?

<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





# Database Design?

- **Example:** Changing the budget of the 'Physics' department
  - Updates to many rows!
  - Easy to break **integrity**
    - ▶ If we forget to update a row, then we have multiple budget values for the physics 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



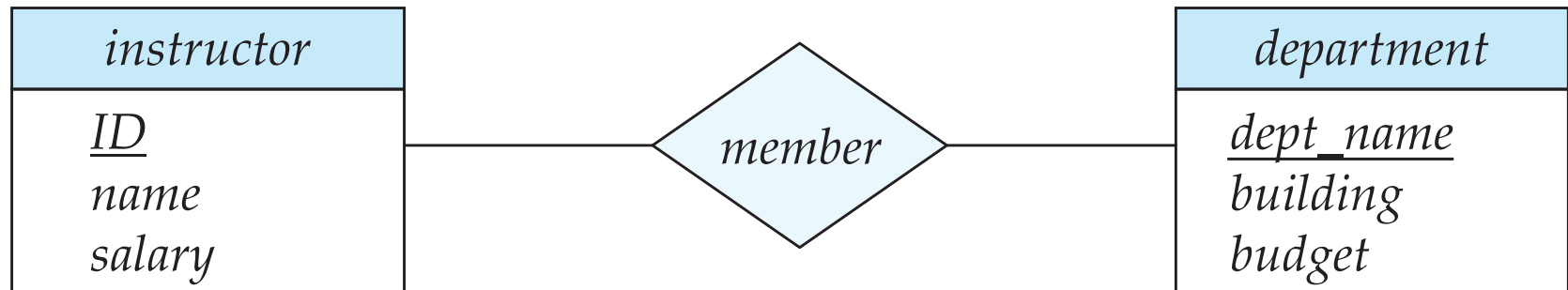
# Design Approaches

- Normalization Theory (Chapter 8)
  - Formalize what designs are “good”, and test for them
  - Translate a “bad” into a “good” design
- Entity Relationship Model (Chapter 7)
  - Models an domain as a collection of *entities* and *relationships*
    - ▶ Entity: a “thing” or “object” in the domain that is distinguishable from other objects
      - Described by a set of *attributes*
    - ▶ Relationship: an association among several entities
  - Represented diagrammatically by an *entity-relationship diagram*



# The Entity-Relationship Model

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



**What happened to dept\_name of instructor and student?**



# Object-Relational Data Models

- Relational model: flat, “atomic” values
  - E.g., integer
- Object Relational Data Models
  - Extend the relational data model by including object orientation and constructs to deal with added data types.
  - Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
  - Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
  - Provide upward compatibility with existing relational languages.



# XML: Extensible Markup Language

- Defined by the WWW Consortium (W3C)
- Originally intended as a document markup language not a database language
- The ability to specify new tags, and to create nested tag structures made XML a great way to exchange **data**, not just documents
- XML has become the basis for all new generation data interchange formats.
- A wide variety of tools is available for parsing, browsing and querying XML documents/data



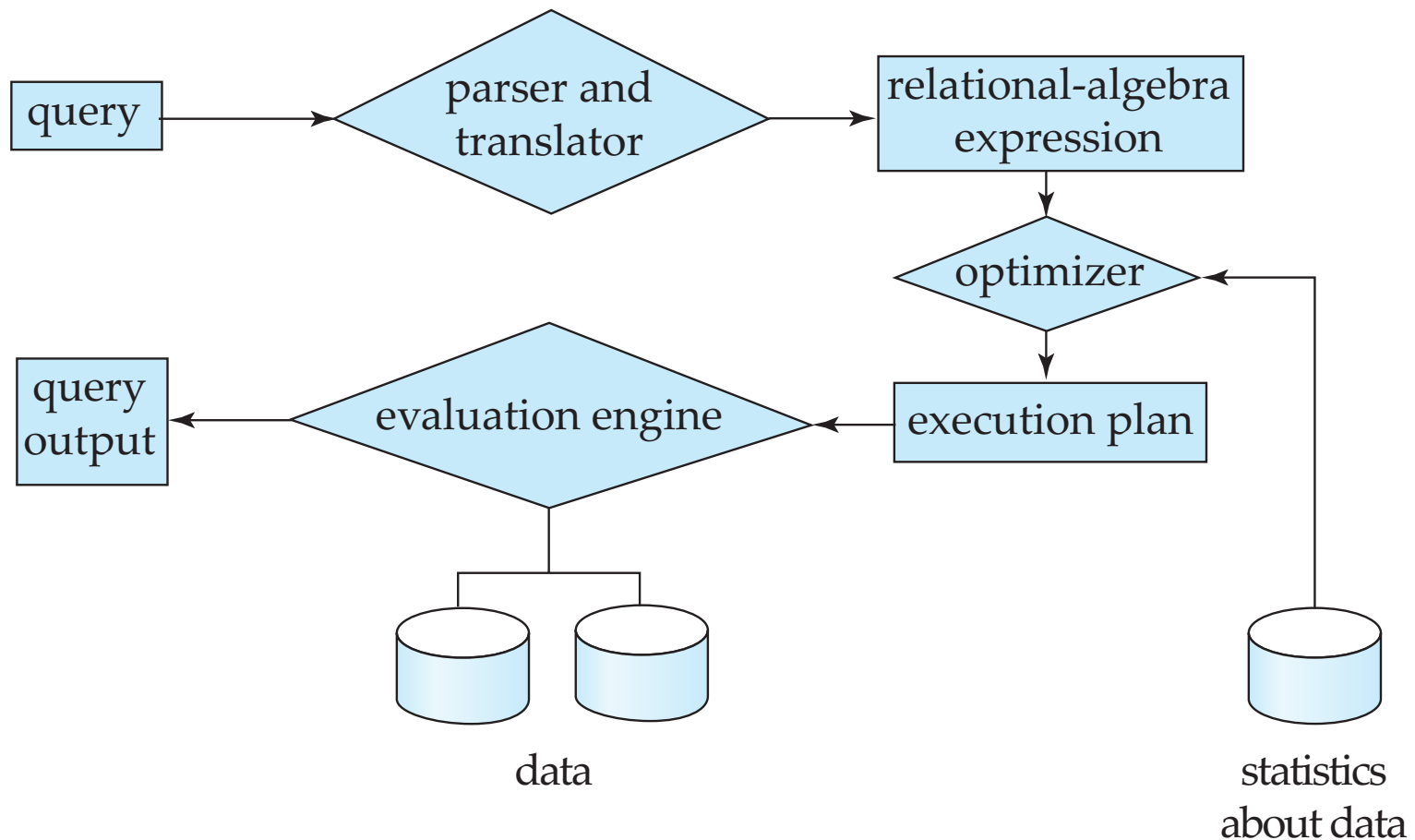
# Storage Management

- **Storage manager** is a program module that provides the interface between the low-level data stored in the database (on disk) and the application programs and queries submitted to the system.
- The storage manager is responsible to the following tasks:
  - Interaction with the file manager
  - Efficient storing, retrieving and updating of data
- Issues:
  - Storage access
  - File organization
  - Indexing and hashing



# Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation





# Query Processing (Cont.)

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation
- Cost difference between a good and a bad way of evaluating a query can be enormous
- Need to estimate the cost of operations
  - Depends critically on statistical information about relations which the database must maintain
  - Need to estimate statistics for intermediate results to compute cost of complex expressions
- Need to search for a good plan (low costs)
  - Traversing the search space of alternative ways (plans) to compute the query result
  - This is called **query optimization**



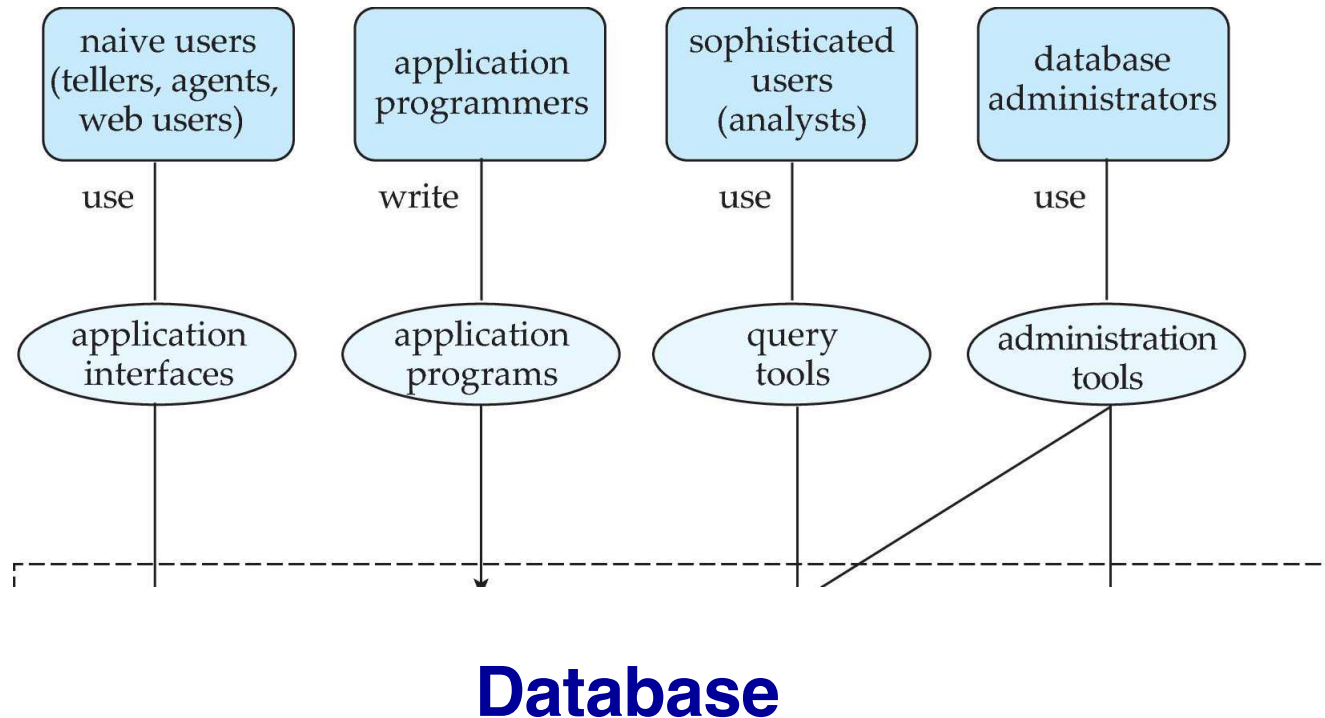


# Transaction Management

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

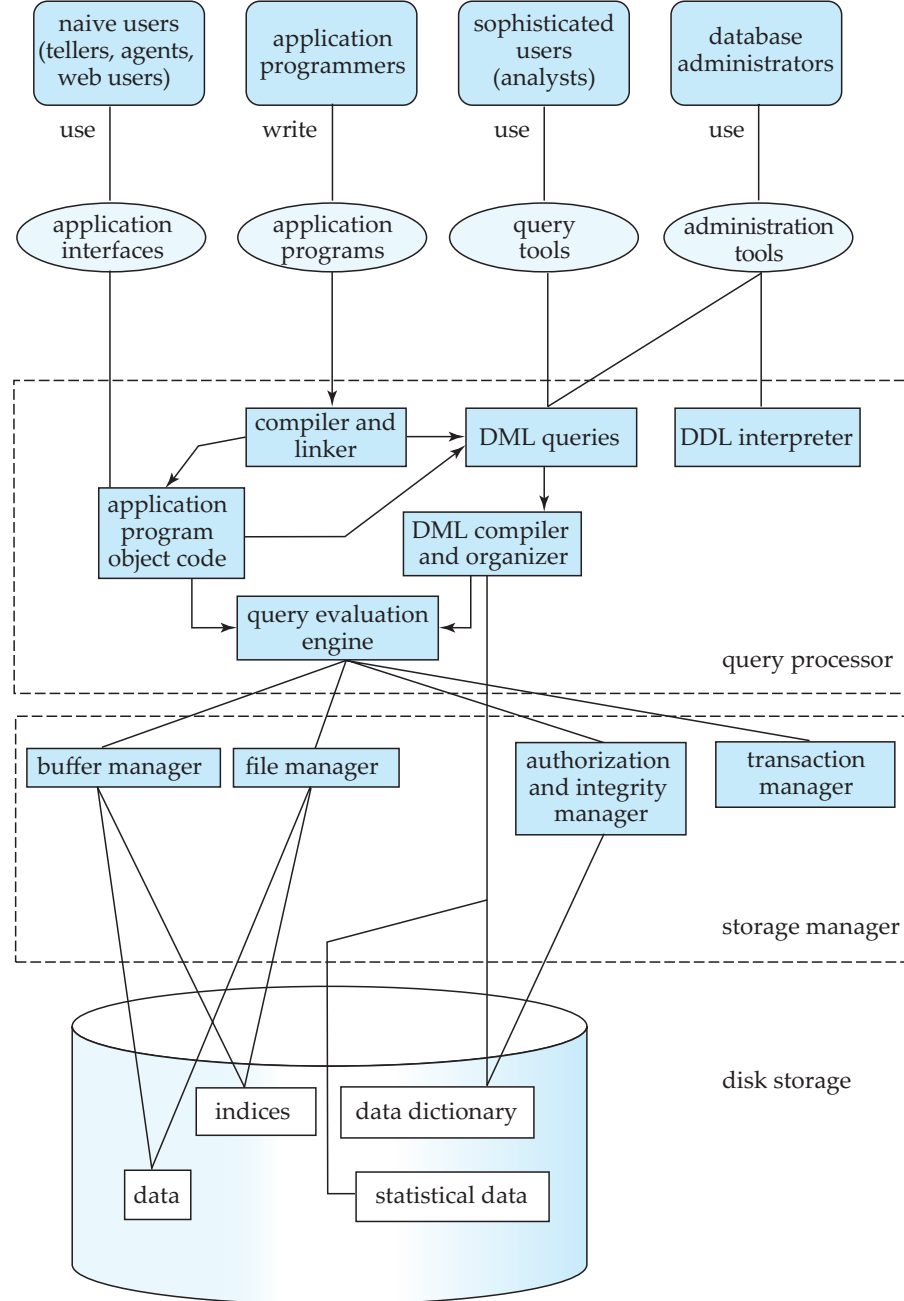


# Database Users and Administrators





# Database System Internals





# Database Architecture

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

- Centralized (embedded, e.g., SQLite)
- vs. Client-server (e.g., Postgres, DB2, Oracle, ...)
- Parallel (multi-processor) (most systems)
- Distributed (e.g., DB2, Hive, ...)



# Build a Complete Database System in your free time?

- How much time do you need?
- To get a rough idea:
  - Postgres (about 800,000 lines of code)
    - ▶ Hundreds of man-years of work
  - Oracle (about 8,000,000 lines of code)
    - ▶ Probably thousands of man-years of work?
- Hmm, ... probably not!
- Maybe a limited research prototype or new feature ;-)



# History of Database Systems

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



# History (cont.)

- 1980s:
  - Research relational prototypes evolve into commercial systems
    - ▶ SQL becomes industrial standard
  - Parallel and distributed database systems
  - Object-oriented database systems
- 1990s:
  - Large decision support and data-mining applications
  - Large multi-terabyte data warehouses
  - Emergence of Web commerce
- Early 2000s:
  - XML and XQuery standards
- Later 2000s:
  - Scalable data storage systems
    - ▶ Google BigTable, Yahoo PNuts, Amazon, ..
  - Scalable distributed query processing
    - ▶ Hive, Spark SQL, Impala, Apache Flink, ...



# Recap

- Why databases?
- What do databases do?
- Data independence
  - Physical and Logical
- Database design
- Data models
  - Relational, object, XML, network, hierarchical
- Query languages
  - DML
  - DDL
- Architecture and systems aspects of database systems
  - Recovery
  - Concurrency control
  - Query processing (optimization)
  - File organization and indexing
- History of databases





# End of Chapter 1



# Outline

- Introduction
- **Relational Data Model**
- Formal Relational Languages (relational algebra)
- SQL
- Database Design
- Transaction Processing, Recovery, and Concurrency Control
- Storage and File Structures
- Indexing and Hashing
- Query Processing and Optimization



# Figure 1.02

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

(a) The *instructor* table

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

(b) The *department* table

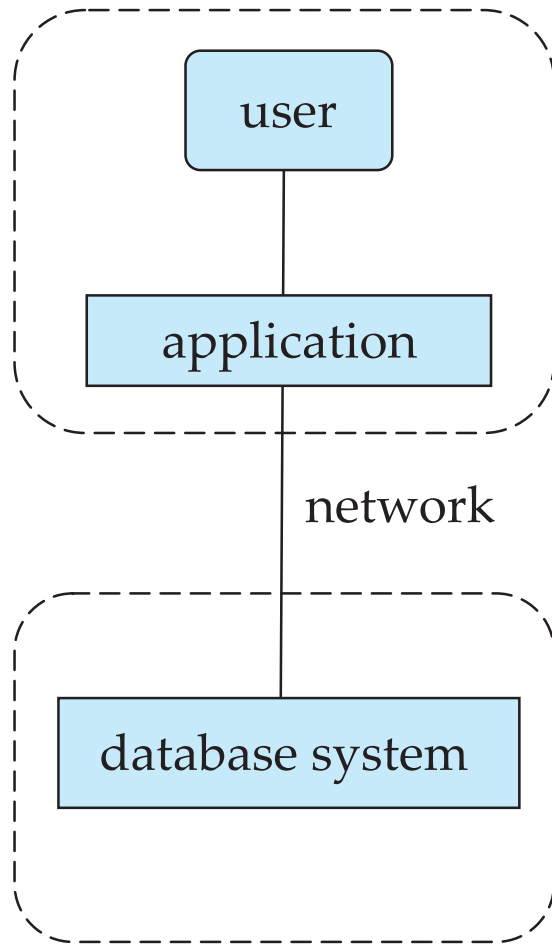


# Figure 1.04

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

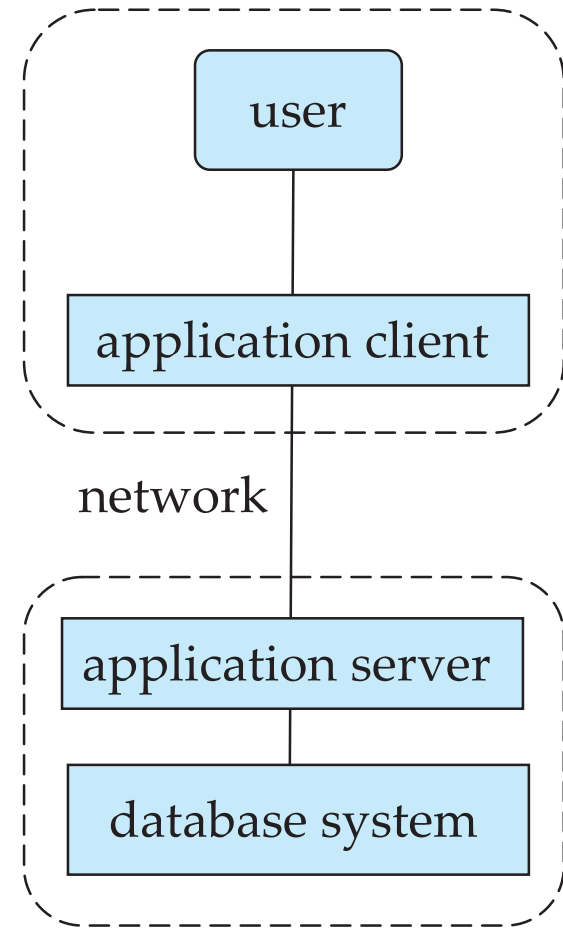


# Figure 1.06



(a) Two-tier architecture

client



(b) Three-tier architecture

server



**CS425 – Fall 2016**  
**Boris Glavic**  
**Chapter 2: Intro to Relational Model**

**Modifies from:**

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

©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Textbook: Chapter 2



# Example of a Relation

attributes  
(or columns)

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

tuples  
(or rows)





# Attribute Types

- The set of allowed values for each attribute is called the **domain** or **data type** of the attribute
- Attribute values are (normally) required to be **atomic**; that is, indivisible
  - E.g., integer values
  - E.g., not address (street, city, zip code, state, country)
- The special value **null** is a member of every domain
  - Means *unknown* or *not applicable*
- The null value causes complications in the definition of many operations
  - Will be detailed later



# Relation Schema and Instance

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

Example:

*instructor = (ID, name, dept\_name, salary)*

- Formally, given sets  $D_1, D_2, \dots, D_n$  of domains a **relation  $r$**  (or **relation instance**) is a subset of

$$D_1 \times D_2 \times \dots \times D_n$$

Thus, a relation is a **set of  $n$ -tuples**  $(a_1, a_2, \dots, a_n)$  where each  $a_i \in D_i$

- The current values (**relation instance**) of a relation are often specified in tabular form
  - **Caveat: being a set, the tuples of the relation do not have any order defined as implied by the tabular representation**
- An element  $t$  of  $r$  is a *tuple*, represented as a *row* in a table





# Alternative Definitions

- A relation schema is often defined as a list of attribute-domain pairs
  - That is the data types of each attribute in the relation are considered as part of the relation schema
- Tuples are sometimes defined as functions from attribute names to values (order of attributes does not matter)
  - E.g.,  $t(\text{name}) = \text{'Bob'}$
- A relation  $r$  can be specified as a function
  - $D_1 \times D_2 \times \dots \times D_n \rightarrow \{true, false\}$
  - $\mathbf{t} = (a_1, a_2, \dots, a_n)$  is mapped to *true* if  $\mathbf{t}$  is in  $r$  and to *false* otherwise
- These alternative definition are useful in database theory
  - We will stick to the simple definition!





# Relations are Unordered

- A relation is a **set** -> the elements of a set are not ordered per se
- From a practical perspective:
  - Order of tuples is irrelevant (tuples may be stored in an arbitrary order)
- Example: *instructor* relation with unordered tuples

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



# Database

- A **database schema**  $S$  consists of multiple relation schema
- A **database instance**  $I$  for a schema  $S$  is a set of relation instances
  - One relation for each relation schema in  $S$
- Information about an enterprise is broken up into parts

*instructor*  
*student*  
*advisor*

- Bad design:  
*univ (instructor -ID, name, dept\_name, salary, student\_Id, ..)*  
results in
  - repetition of information (e.g., two students have the same instructor)
  - the need for many null values (e.g., represent an student with no advisor)
- Normalization theory (Chapter 7) deals with how to design “good” relational schemas avoiding these problems



# Bad Design Example Revisited

- **Example:** Changing the budget of the 'Physics' department
  - Updates to many rows!
    - ▶ Easy to break **integrity**
    - ▶ If we forget to update a row, then we have multiple budget values for the physics department!
- **Example:** Deleting all employees from the 'Physics' department
  - How to avoid deleting the 'Physics' department?
  - Dummy employee's to store departments?
    - ▶ This is bad. E.g., counting the number of employees per department becomes more involved

<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



# Keys

- Let  $K \subseteq R$
- $K$  is a **superkey** of  $R$  if values for  $K$  are sufficient to identify a unique tuple of each possible relation  $r(R)$ 
  - Example:  $\{ID\}$  and  $\{ID, name\}$  are both superkeys of *instructor*.
- Superkey  $K$  is a **candidate key** if  $K$  is minimal (no subset of  $K$  is also a superkey)
  - Example:  $\{ID\}$  is a candidate key for *Instructor*
- One of the candidate keys is selected to be the **primary key**.
  - which one? -> domain specific design choice
- **Foreign key** constraint: Value in one relation must appear in another
  - **Referencing** relation
  - **Referenced** relation



# Keys

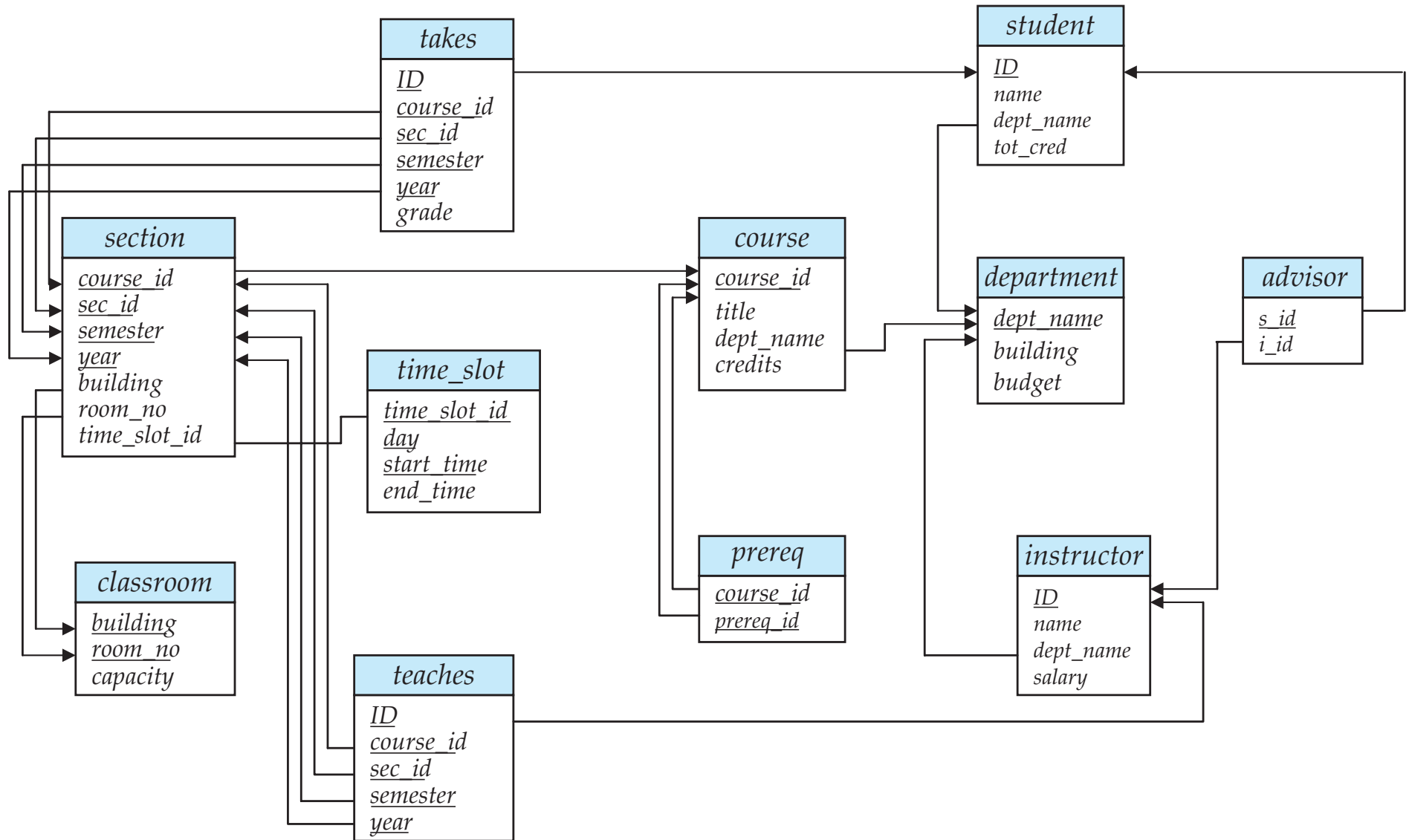
- Formally, a set of attributes  $K \subseteq R$  is a superkey if for every instance  $r$  of  $R$  holds that
  - $\forall t, t' \in r: t.K = t'.K \Rightarrow t = t'$
- A superkey  $K$  is called a candidate key iff
  - $\forall K' \subseteq K: K'$  is not a superkey
- A foreign key constraint  $FK$  is quartuple  $(R, K, R', K')$  where  $R$  and  $R'$  are relation schemata,  $K \subseteq R$ ,  $K'$  is the primary key of  $R'$ , and  $|K| = |K'|$
- A foreign key holds over an instance  $\{r, r'\}$  for  $\{R, R'\}$  iff
  - $\forall t \in R: \exists t' \in R': t.K = t'.K'$







# Schema Diagram for the University Database





# End of Chapter 2

**Modifies from:**

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

©Silberschatz, Korth and Sudarshan

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



# Recap

- **Database Schema** (or short schema)
  - Set of **relation schemata**
    - ▶ List of **attribute names**
- **Database Instance** (or short database)
  - Set of **relations instances**
    - ▶ Set of **tuples**
      - List of **attribute values**
- **Integrity Constraints**
  - **Keys** (Super-, Candidate-, Primary-)
    - ▶ For identifying tuples
  - **Foreign keys**
    - ▶ For referencing tuples in other relations



# Outline

- Introduction
- Relational Data Model
- **Formal Relational Languages (relational algebra)**
- SQL
- Database Design
- Transaction Processing, Recovery, and Concurrency Control
- Storage and File Structures
- Indexing and Hashing
- Query Processing and Optimization



## Figure 2.01

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



## Figure 2.02

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



## Figure 2.03

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



## Figure 2.04

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





## Figure 2.05

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



# Figure 2.06

<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>building</i>	<i>room_number</i>	<i>time_slot_id</i>
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



# Figure 2.07

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



## Figure 2.10

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



## Figure 2.11

<i>ID</i>	<i>salary</i>
10101	65000
12121	90000
15151	40000
22222	95000
32343	60000
33456	87000
45565	75000
58583	62000
76543	80000
76766	72000
83821	92000
98345	80000



## Figure 2.12

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



## Figure 2.13

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



## CS425 – Fall 2016 Boris Glavic Chapter 3: Formal Relational Query Languages

Modified from:  
Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on reuse



## Chapter 3: Formal Relational Query Languages

- Relational Algebra
- Tuple Relational Calculus
- Domain Relational Calculus



Textbook: Chapter 6

CS425 – Fall 2016 – Boris Glavic

3.2

©Silberschatz, Korth and Sudarshan



## Relational Query Languages

- Procedural vs non-procedural (**declarative**)
- “Pure” languages:
  - Relational algebra
  - Tuple relational calculus
  - Domain relational calculus
- Expressive power of a query language
  - What queries can be expressed in this language?
- Relational algebra:
  - Algebra of relations -> set of operators that take relations as input and produce relations as output
  - -> **composable**: the output of evaluating an expression in relational algebra can be used as input to another relational algebra expression
- Now: First introduction to operators of the relational algebra

CS425 – Fall 2016 – Boris Glavic

3.3

©Silberschatz, Korth and Sudarshan



## Relational Algebra

- Procedural language
- Six basic operators
  - select:  $\sigma$
  - project:  $\Pi$
  - union:  $\cup$
  - set difference:  $-$
  - Cartesian product:  $\times$
  - rename:  $\rho$
- The operators take one or two relations as inputs and produce a new relation as a result.
  - **composable**

CS425 – Fall 2016 – Boris Glavic

3.4

©Silberschatz, Korth and Sudarshan



## Select Operation – Example

- Relation r

A	B	C	D
$\alpha$	$\alpha$	1	7
$\alpha$	$\beta$	5	7
$\beta$	$\beta$	12	3
$\beta$	$\beta$	23	10

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

A	B	C	D
$\alpha$	$\alpha$	1	7
$\beta$	$\beta$	23	10

CS425 – Fall 2016 – Boris Glavic

3.5

©Silberschatz, Korth and Sudarshan



## Select Operation

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

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

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

Each **term** is one of:

<attribute>  $op$  <attribute> or <constant>

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

- Example of selection:

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



CS425 – Fall 2016 – Boris Glavic

3.6

©Silberschatz, Korth and Sudarshan



### Project Operation – Example

- Relation  $r$ :
 

A	B	C
$\alpha$	10	1
$\alpha$	20	1
$\beta$	30	1
$\beta$	40	2
- $\Pi_{A,C}(r)$ :
 

A	C
$\alpha$	1
$\alpha$	1
$\beta$	1
$\beta$	2

 =
 

A	C
$\alpha$	1
$\beta$	1
$\beta$	2

CS425 – Fall 2016 – Boris Glavic 3.7 ©Silberschatz, Korth and Sudarshan

### Project Operation

- Notation:
 
$$\pi_{A_1, A_2, \dots, A_k}(r)$$
 where  $A_1, A_2$  are attribute names and  $r$  is a relation name.
- The result is defined as the relation of  $k$  columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are sets
- Let  $A$  be a subset of the attributes of relation  $r$  then:
 
$$\pi_A(r) = \{t.A \mid t \in r\}$$
- Example: To eliminate the *dept\_name* attribute of *instructor*

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

**! theory Warning**

CS425 – Fall 2016 – Boris Glavic 3.8 ©Silberschatz, Korth and Sudarshan

### Union Operation – Example

- Relations  $r, s$ :
 

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

 $r$ 

A	B
$\alpha$	2
$\beta$	3

 $s$
- $r \cup s$ :
 

A	B
$\alpha$	1
$\alpha$	2
$\beta$	1
$\beta$	3

CS425 – Fall 2016 – Boris Glavic 3.9 ©Silberschatz, Korth and Sudarshan

### Union Operation

- Notation:  $r \cup s$
- Defined as:
 
$$r \cup s = \{t \mid t \in r \vee t \in s\}$$
- For  $r \cup s$  to be valid.
  - $r, s$  must have the **same arity** (same number of attributes)
  - The attribute domains must be **compatible** (example: 2<sup>nd</sup> column of  $r$  deals with the same type of values as does the 2<sup>nd</sup> column of  $s$ )
- Example: to find all courses taught in the Fall 2009 semester, or in the Spring 2010 semester, or in both
 
$$\Pi_{course\_id}(\sigma_{semester="Fall" \wedge year=2009}(section)) \cup \Pi_{course\_id}(\sigma_{semester="Spring" \wedge year=2010}(section))$$

CS425 – Fall 2016 – Boris Glavic 3.10 ©Silberschatz, Korth and Sudarshan

### Set difference of two relations

- Relations  $r, s$ :
 

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

 $r$ 

A	B
$\alpha$	2
$\beta$	3

 $s$
- $r - s$ :
 

A	B
$\alpha$	1
$\beta$	1

CS425 – Fall 2016 – Boris Glavic 3.11 ©Silberschatz, Korth and Sudarshan

### Set Difference Operation

- Notation  $r - s$
- Defined as:
 
$$r - s = \{t \mid t \in r \wedge t \notin s\}$$
- Set differences must be taken between **compatible** relations.
  - $r$  and  $s$  must have the **same arity**
  - attribute domains of  $r$  and  $s$  must be compatible
- Example: to find all courses taught in the Fall 2009 semester, but not in the Spring 2010 semester
 
$$\Pi_{course\_id}(\sigma_{semester="Fall" \wedge year=2009}(section)) - \Pi_{course\_id}(\sigma_{semester="Spring" \wedge year=2010}(section))$$

CS425 – Fall 2016 – Boris Glavic 3.12 ©Silberschatz, Korth and Sudarshan

### Cartesian-Product Operation – Example

- Relations  $r, s$ :
 

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

 $r$ 

C	D	E
$\alpha$	10	a
$\beta$	10	a
$\beta$	20	b
$\gamma$	10	b

 $s$
- $r \times s$ :
 

A	B	C	D	E
$\alpha$	1	$\alpha$	10	a
$\alpha$	1	$\beta$	10	a
$\alpha$	1	$\beta$	20	b
$\alpha$	1	$\gamma$	10	b
$\beta$	2	$\alpha$	10	a
$\beta$	2	$\beta$	10	a
$\beta$	2	$\beta$	20	b
$\beta$	2	$\gamma$	10	b

CS425 – Fall 2016 – Boris Glavic 3.13 ©Silberschatz, Korth and Sudarshan

### Cartesian-Product Operation

- Notation  $r \times s$
- Defined as:
 
$$r \times s = \{t, t' \mid t \in r \wedge t' \in s\}$$
- Assume that attributes of  $r(R)$  and  $s(S)$  are disjoint. (That is,  $R \cap S = \emptyset$ ).
- If attributes of  $r(R)$  and  $s(S)$  are not disjoint, then renaming must be used.

CS425 – Fall 2016 – Boris Glavic 3.14 ©Silberschatz, Korth and Sudarshan

### Composition of Operations

- Can build expressions using multiple operations
- Example:  $\sigma_{A=C}(r \times s)$
- $r \times s$ 

A	B	C	D	E
$\alpha$	1	$\alpha$	10	a
$\alpha$	1	$\beta$	10	a
$\alpha$	1	$\beta$	20	b
$\alpha$	1	$\gamma$	10	b
$\beta$	2	$\alpha$	10	a
$\beta$	2	$\beta$	10	a
$\beta$	2	$\beta$	20	b
$\beta$	2	$\gamma$	10	b
- $\sigma_{A=C}(r \times s)$ 

A	B	C	D	E
$\alpha$	1	$\alpha$	10	a
$\beta$	2	$\beta$	10	a
$\beta$	2	$\beta$	20	b

CS425 – Fall 2016 – Boris Glavic 3.15 ©Silberschatz, Korth and Sudarshan

### Rename Operation

- Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
- Allows us to refer to a relation by more than one name.
- Example:
 
$$\rho_X(t)$$
 returns the expression  $E$  under the name  $X$
- If a relational-algebra expression  $E$  has arity  $n$ , then
 
$$\rho_{X(A_1, A_2, \dots, A_n)}(r)$$
 returns the result of expression  $E$  under the name  $X$ , and with the attributes renamed to  $A_1, A_2, \dots, A_n$ .
 
$$\rho_X(r) = \{t(X) \mid t \in r\}$$

$$\rho_{X(A)}(r) = \{t(X).A \mid t \in r\}$$

CS425 – Fall 2016 – Boris Glavic 3.16 ©Silberschatz, Korth and Sudarshan

### Example Query

- Find the largest salary in the university
  - Step 1: find instructor salaries that are less than some other instructor salary (i.e. not maximum)
    - using a copy of *instructor* under a new name  $d$ 

$$\pi_{instructor.salary}(\sigma_{instructor.salary < d.salary}(instructor \times \rho_d(instructor)))$$
  - Step 2: Find the largest salary
    - $\pi_{salary}(instructor) -$ 

$$\pi_{instructor.salary}(\sigma_{instructor.salary < d.salary}(instructor \times \rho_d(instructor)))$$

CS425 – Fall 2016 – Boris Glavic 3.17 ©Silberschatz, Korth and Sudarshan

### Example Queries

- Find the names of all instructors in the Physics department, along with the *course\_id* of all courses they have taught
  - Query 1
 
$$\pi_{instructor.ID, course\_id}(\sigma_{dept\_name='Physics'}(\sigma_{instructor.ID=teaches.ID}(instructor \times teaches)))$$
  - Query 2
 
$$\pi_{instructor.ID, course\_id}(\sigma_{instructor.ID=teaches.ID}(\sigma_{dept\_name='Physics'}(instructor \times teaches)))$$

CS425 – Fall 2016 – Boris Glavic 3.18 ©Silberschatz, Korth and Sudarshan



## Formal Definition (Syntax)

- A basic expression in the relational algebra consists of either one of the following:
  - A relation in the database
  - A constant relation: e.g.,  $\{(1),(2)\}$
- Let  $E_1$  and  $E_2$  be relational-algebra expressions; the following are all relational-algebra expressions:
  - $E_1 \cup E_2$
  - $E_1 - E_2$
  - $E_1 \times E_2$
  - $\sigma_P(E_1)$ ,  $P$  is a predicate on attributes in  $E_1$
  - $\pi_S(E_1)$ ,  $S$  is a list consisting of some of the attributes in  $E_1$
  - $\rho_X(E_1)$ ,  $X$  is the new name for the result of  $E_1$



CS425 - Fall 2016 - Boris Glavic

3.19

©Silberschatz, Korth and Sudarshan



## Formal Definition (Semantics)

- Let  $E$  be an relational algebra expression. We use  $[E](I)$  to denote the evaluation of  $E$  over a database instance  $I$ 
  - For simplicity we will often drop  $I$  and  $[\ ]$
- The result of evaluating a simple relational algebra expression  $E$  over a database is defined as
  - Simple relation:  $[R](I) = R(I)$
  - Constant relation:  $[C](I) = C$



CS425 - Fall 2016 - Boris Glavic

3.20

©Silberschatz, Korth and Sudarshan



## Formal Definition (Semantics)

- Let  $E_1$  and  $E_2$  be relational-algebra expressions.

$$[E_1 \cup E_2] = \{t \mid t \in [E_1] \vee t \in [E_2]\}$$

$$[E_1 - E_2] = \{t \mid t \in [E_1] \wedge t \notin [E_2]\}$$

$$[E_1 \times E_2] = \{t, t' \mid t \in [E_1] \wedge t' \in [E_2]\}$$

$$[\sigma_P(E_1)] = \{t \mid t \in [E_1] \wedge p(t)\}$$

$$[\pi_A(E_1)] = \{t.A \mid t \in [E_1]\}$$

$$[\rho_X(E_1)] = \{t(X) \mid t \in [E_1]\}$$



CS425 - Fall 2016 - Boris Glavic

3.21

©Silberschatz, Korth and Sudarshan



## Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*.
- Aggregate functions simply ignore null values (as in SQL)
- For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same (as in SQL)
- For logical expressions
  - True AND null = null, False AND null = False
  - True OR null = True, False OR null = null
  - Not (null) = null

CS425 - Fall 2016 - Boris Glavic

3.22

©Silberschatz, Korth and Sudarshan



## Null Values

- Comparisons with null values return the special truth value: *unknown*
  - If *false* was used instead of *unknown*, then  $not(A < 5)$  would not be equivalent to  $A \geq 5$
- Three-valued logic using the truth value *unknown*:
  - OR: (*unknown* or *true*) = *true*,  
(*unknown* or *false*) = *unknown*,  
(*unknown* or *unknown*) = *unknown*
  - AND: (*true* and *unknown*) = *unknown*,  
(*false* and *unknown*) = *false*,  
(*unknown* and *unknown*) = *unknown*
  - NOT: (*not unknown*) = *unknown*
  - In SQL "*P* is *unknown*" evaluates to true if predicate  $P$  evaluates to *unknown*
- Result of select predicate is treated as *false* if it evaluates to *unknown*

CS425 - Fall 2016 - Boris Glavic

3.23

©Silberschatz, Korth and Sudarshan



## Additional Operations

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

- Set intersection
- Natural join
- Assignment
- Outer join

CS425 - Fall 2016 - Boris Glavic

3.24

©Silberschatz, Korth and Sudarshan

### Set-Intersection Operation

- Notation:  $r \cap s$
- Defined as:
 
$$r \cap s = \{t \mid t \in r \wedge t \in s\}$$
- Assume:
  - $r, s$  have the same arity
  - attributes of  $r$  and  $s$  are compatible
- Note:  $r \cap s = r - (r - s)$ 
  - That is adding intersection to the language does not make it more expressive

CS425 – Fall 2016 – Boris Glavic 3.25 ©Silberschatz, Korth and Sudarshan

### Set-Intersection Operation – Example

- Relation  $r, s$ :
 

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

A	B
$\alpha$	2
$\beta$	3
- $r \cap s$ 

A	B
$\alpha$	2

CS425 – Fall 2016 – Boris Glavic 3.26 ©Silberschatz, Korth and Sudarshan

### Natural-Join Operation

- Notation:  $r \bowtie s$
- Let  $r$  and  $s$  be relations on schemas  $R$  and  $S$  respectively. Then,  $r \bowtie s$  is a relation on schema  $R \cup S$  obtained as follows:
  - Consider each pair of tuples  $t_r$  from  $r$  and  $t_s$  from  $s$ .
  - If  $t_r$  and  $t_s$  have the same value on each of the attributes in  $R \cap S$ , add a tuple  $t$  to the result, where
    - $t$  has the same value as  $t_r$  on  $r$
    - $t$  has the same value as  $t_s$  on  $s$
- Example:
  - $R = (A, B, C, D)$
  - $S = (E, B, D)$
  - Result schema =  $(A, B, C, D, E)$
  - $r \bowtie s$  is defined as:
 
$$\Pi_{r,A,r.B,r.C,r.D,s.E}(\sigma_{r.B=s.B \wedge r.D=s.D}(r \times s))$$

CS425 – Fall 2016 – Boris Glavic 3.27 ©Silberschatz, Korth and Sudarshan

### Natural-Join Operation (cont.)

- Let  $r$  and  $s$  be relations on schemas  $R$  and  $S$  respectively. Then,  $r \bowtie s$  is defined as:
 
$$X = R \cap S$$

$$S' = S - R$$

$$r \bowtie s = \pi_{R,S'}(\sigma_{r.X=s.X}(r \times s))$$

CS425 – Fall 2016 – Boris Glavic 3.28 ©Silberschatz, Korth and Sudarshan

### Natural Join Example

- Relations  $r, s$ :
 

A	B	C	D
$\alpha$	1	$\alpha$	a
$\beta$	2	$\gamma$	a
$\gamma$	4	$\beta$	b
$\alpha$	1	$\gamma$	a
$\delta$	2	$\beta$	b

B	D	E
1	a	$\alpha$
3	a	$\beta$
1	a	$\gamma$
2	b	$\delta$
3	b	$\epsilon$
- $r \bowtie s$ 

A	B	C	D	E
$\alpha$	1	$\alpha$	a	$\alpha$
$\alpha$	1	$\alpha$	a	$\gamma$
$\alpha$	1	$\gamma$	a	$\alpha$
$\alpha$	1	$\gamma$	a	$\gamma$
$\delta$	2	$\beta$	b	$\delta$

CS425 – Fall 2016 – Boris Glavic 3.29 ©Silberschatz, Korth and Sudarshan

### Natural Join and Theta Join

- Find the names of all instructors in the Comp. Sci. department together with the course titles of all the courses that the instructors teach
  - $\Pi_{name, title}(\sigma_{dept\_name='Comp. Sci.'}(instructor \bowtie teaches_{course}))$
- Natural join is associative
  - $(instructor \bowtie teaches_{course}) \bowtie instructor$  is equivalent to  $instructor \bowtie (teaches_{course})$
- Natural join is commutative (we ignore attribute order)
  - $instructor \bowtie teaches$  is equivalent to  $teaches \bowtie instructor$
- The **theta join** operation  $\bowtie_{\theta}$  is defined as
 
$$r \bowtie_{\theta} s = \sigma_{\theta}(r \times s)$$

CS425 – Fall 2016 – Boris Glavic 3.30 ©Silberschatz, Korth and Sudarshan



## Assignment Operation

- The assignment operation ( $\leftarrow$ ) provides a convenient way to express complex queries.
  - Write query as a sequential program consisting of
    - a series of assignments
    - followed by an expression whose value is displayed as a result of the query.
  - Assignment must always be made to a temporary relation variable.

$$E_1 \leftarrow \sigma_{salary > 40000}(instructor)$$

$$E_2 \leftarrow \sigma_{salary < 10000}(instructor)$$

$$E_3 \leftarrow E_1 \cup E_2$$

CS425 – Fall 2016 – Boris Glavic

3.31

©Silberschatz, Korth and Sudarshan



## Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values:
  - *null* signifies that the value is unknown or does not exist
  - All comparisons involving *null* are (roughly speaking) **false** by definition.
    - We shall study precise meaning of comparisons with nulls later

CS425 – Fall 2016 – Boris Glavic

3.32

©Silberschatz, Korth and Sudarshan



## Outer Join – Example

- Relation *instructor1*

ID	name	dept_name
10101	Srinivasan	Comp. Sci.
12121	Wu	Finance
15151	Mozart	Music

- Relation *teaches1*

ID	course_id
10101	CS-101
12121	FIN-201
76766	BIO-101

CS425 – Fall 2016 – Boris Glavic

3.33

©Silberschatz, Korth and Sudarshan



## Outer Join – Example

- Join

 $instructor \bowtie teaches$ 

ID	name	dept_name	course_id
10101	Srinivasan	Comp. Sci.	CS-101
12121	Wu	Finance	FIN-201

- Left Outer Join

 $instructor \ltimes teaches$ 

ID	name	dept_name	course_id
10101	Srinivasan	Comp. Sci.	CS-101
12121	Wu	Finance	FIN-201
15151	Mozart	Music	null

CS425 – Fall 2016 – Boris Glavic

3.34

©Silberschatz, Korth and Sudarshan



## Outer Join – Example

- Right Outer Join

 $instructor \rhd \bowtie teaches$ 

ID	name	dept_name	course_id
10101	Srinivasan	Comp. Sci.	CS-101
12121	Wu	Finance	FIN-201
76766	null	null	BIO-101

- Full Outer Join

 $instructor \ltimes \bowtie teaches$ 

ID	name	dept_name	course_id
10101	Srinivasan	Comp. Sci.	CS-101
12121	Wu	Finance	FIN-201
15151	Mozart	Music	null
76766	null	null	BIO-101

CS425 – Fall 2016 – Boris Glavic

3.35

©Silberschatz, Korth and Sudarshan



## Outer Join using Joins

- Outer join can be expressed using basic operations

$$r \rhd \bowtie s = (r \bowtie s) \cup ((r - \Pi_R(r \bowtie s)) \times \{(null, \dots, null)\})$$

$$r \ltimes \bowtie s = (r \bowtie s) \cup (\{(null, \dots, null)\} \times (s - \Pi_S(r \bowtie s)))$$

$$r \ltimes \bowtie s = (r \bowtie s) \cup ((r - \Pi_R(r \bowtie s)) \times \{(null, \dots, null)\}) \cup (\{(null, \dots, null)\} \times (s - \Pi_S(r \bowtie s)))$$

CS425 – Fall 2016 – Boris Glavic

3.36

©Silberschatz, Korth and Sudarshan

## Division Operator

- Given relations  $r(R)$  and  $s(S)$ , such that  $S \subset R$ ,  $r \div s$  is the largest relation  $t(R-S)$  such that  $t \times s \subseteq r$ 
  - Alternatively, all tuples from  $r.(R-S)$  such that all their extensions on  $R \cap S$  with tuples from  $s$  exist in  $R$
- E.g. let  $r(ID, course\_id) = \Pi_{ID, course\_id}(takes)$  and  $s(course\_id) = \Pi_{course\_id}(\sigma_{dept\_name='Biology'}(course))$  then  $r \div s$  gives us students who have taken all courses in the Biology department
- Can write  $r \div s$  as
 
$$E_1 \leftarrow \Pi_{R-S}(r)$$

$$E_2 \leftarrow \Pi_{R-S}((E_1 \times s) - \Pi_{R-S,S}(r \bowtie s))$$

$$r \div s = E_1 - E_2$$

CS425 - Fall 2016 - Boris Glavic 3.37 ©Silberschatz, Korth and Sudarshan

## Division Operator Example

- Return the name of all persons that read all newspapers

reads	
name	newspaper
Peter	Times
Bob	Wall Street
Alice	Times
Alice	Wall Street

newspaper
Times
Wall Street

$$E_1 \leftarrow \Pi_{name}(reads)$$

$$E_2 \leftarrow ((E_1 \times newspaper) - \Pi_{name,newspaper}(reads \bowtie newspaper))$$

$$reads \div newspaper = E_1 - E_2$$

$$[reads \div newspaper] = \{Alice\}$$

CS425 - Fall 2016 - Boris Glavic 3.38 ©Silberschatz, Korth and Sudarshan

## Extended Relational-Algebra-Operations

- Generalized Projection
- Aggregate Functions

CS425 - Fall 2016 - Boris Glavic 3.39 ©Silberschatz, Korth and Sudarshan

## Generalized Projection

- Extends the projection operation by allowing arithmetic functions to be used in the projection list.
 
$$\pi_{F_1, \dots, F_n}(E)$$
- $E$  is any relational-algebra expression
- Each of  $F_1, F_2, \dots, F_n$  are arithmetic expressions and function calls involving constants and attributes in the schema of  $E$ .
- Given relation  $instructor(ID, name, dept\_name, salary)$  where salary is annual salary, get the same information but with monthly salary
 
$$\Pi_{ID, name, dept\_name, salary/12}(instructor)$$
- Adding functions increases expressive power!
  - In standard relational algebra there is no way to change attribute values

CS425 - Fall 2016 - Boris Glavic 3.40 ©Silberschatz, Korth and Sudarshan

## Aggregate Functions and Operations

- Aggregation function** takes a set of values and returns a single value as a result.
  - avg**: average value
  - min**: minimum value
  - max**: maximum value
  - sum**: sum of values
  - count**: number of values
- Aggregate operation** in relational algebra
 
$$G_1, G_2, \dots, G_n \mathcal{G}_{F_1(A_1), F_2(A_2), \dots, F_n(A_n)}(E)$$
  - $E$  is any relational-algebra expression
  - $G_1, G_2, \dots, G_n$  is a list of attributes on which to group (can be empty)
  - Each  $F_i$  is an aggregate function
  - Each  $A_i$  is an attribute name
- Note: Some books/articles use  $\gamma$  instead of  $\mathcal{G}$  (Calligraphic G)

CS425 - Fall 2016 - Boris Glavic 3.41 ©Silberschatz, Korth and Sudarshan

## Aggregate Operation - Example

- Relation  $r$ :
 

A	B	C
$\alpha$	$\alpha$	7
$\alpha$	$\beta$	7
$\beta$	$\beta$	3
$\beta$	$\beta$	10
- $\mathcal{G}_{sum(C)}(r)$ 

sum(C)
27

CS425 - Fall 2016 - Boris Glavic 3.42 ©Silberschatz, Korth and Sudarshan



## Aggregate Operation – Example

- Find the average salary in each department

$dept\_name \Join \text{avg}(salary)$  (*instructor*)

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

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



## Aggregate Functions (Cont.)

- What are the names for attributes in aggregation results?
  - Need some convention!
    - E.g., use the expression as a name  $\text{avg}(salary)$
  - For convenience, we permit renaming as part of aggregate operation
    - $dept\_name \Join \text{avg}(salary)$  as  $\text{avg\_sal}$  (*instructor*)



## Modification of the Database

- The content of the database may be modified using the following operations:
  - Deletion
  - Insertion
  - Updating
- All these operations can be expressed using the assignment operator
- Example: Delete instructors with salary over \$1,000,000

$$R \leftarrow R - (\sigma_{salary > 1000000}(R))$$



## Restrictions for Modification

- Consider a modification where  $R=(A,B)$  and  $S=(C)$

$$R \leftarrow \sigma_{C > 5}(S)$$

- This would change the schema of  $R$ !
  - Should not be allowed
- Requirements for modifications
  - The name  $R$  on the left-hand side of the assignment operator refers to an existing relation in the database schema
  - The expression on the right-hand side of the assignment operator should be union-compatible with  $R$




## Tuple Relational Calculus



## Tuple Relational Calculus

- A nonprocedural query language, where each query is of the form  $\{t \mid P(t)\}$
- It is the set of all tuples  $t$  such that predicate  $P$  is true for  $t$
- $t$  is a *tuple variable*,  $t[A]$  denotes the value of tuple  $t$  on attribute  $A$
- $t \in r$  denotes that tuple  $t$  is in relation  $r$
- $P$  is a *formula* similar to that of the predicate calculus




## Predicate Calculus Formula

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

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

CS425 – Fall 2016 – Boris Glavic 3.49 ©Silberschatz, Korth and Sudarshan




## Example Queries

- Find the  $ID$ ,  $name$ ,  $dept\_name$ ,  $salary$  for instructors whose salary is greater than \$80,000
 
$$\{t \mid t \in instructor \wedge t[salary] > 80000\}$$
- As in the previous query, but output only the  $ID$  attribute value
 
$$\{t \mid \exists s \in instructor (t[ID] = s[ID] \wedge s[salary] > 80000)\}$$

Notice that a relation on schema ( $ID$ ) is implicitly defined by the query, because

- 1)  $t$  is not bound to any relation by the predicate
- 2) we implicitly state that  $t$  has an  $ID$  attribute ( $t[ID] = s[ID]$ )


CS425 – Fall 2016 – Boris Glavic 3.50 ©Silberschatz, Korth and Sudarshan



## Example Queries

- Find the names of all instructors whose department is in the Watson building
 
$$\{t \mid \exists s \in instructor (t[name] = s[name] \wedge \exists u \in department (u[dept\_name] = s[dept\_name] \wedge u[building] = "Watson"))\}$$
- Find the set of all courses taught in the Fall 2009 semester, or in the Spring 2010 semester, or both
 
$$\{t \mid \exists s \in section (t[course\_id] = s[course\_id] \wedge s[semester] = "Fall" \wedge s[year] = 2009) \vee \exists u \in section (t[course\_id] = u[course\_id] \wedge u[semester] = "Spring" \wedge u[year] = 2010)\}$$


CS425 – Fall 2016 – Boris Glavic 3.51 ©Silberschatz, Korth and Sudarshan



## Example Queries

- Find the set of all courses taught in the Fall 2009 semester, and in the Spring 2010 semester
 
$$\{t \mid \exists s \in section (t[course\_id] = s[course\_id] \wedge s[semester] = "Fall" \wedge s[year] = 2009) \wedge \exists u \in section (t[course\_id] = u[course\_id] \wedge u[semester] = "Spring" \wedge u[year] = 2010)\}$$
- Find the set of all courses taught in the Fall 2009 semester, but not in the Spring 2010 semester
 
$$\{t \mid \exists s \in section (t[course\_id] = s[course\_id] \wedge s[semester] = "Fall" \wedge s[year] = 2009) \wedge \neg \exists u \in section (t[course\_id] = u[course\_id] \wedge u[semester] = "Spring" \wedge u[year] = 2010)\}$$


CS425 – Fall 2016 – Boris Glavic 3.52 ©Silberschatz, Korth and Sudarshan



## Safety of Expressions

- It is possible to write tuple calculus expressions that generate infinite relations.
- For example,  $\{t \mid \neg t \in r\}$  results in an infinite relation if the domain of any attribute of relation  $r$  is infinite
- To guard against the problem, we restrict the set of allowable expressions to safe expressions.
- An expression  $\{t \mid P(t)\}$  in the tuple relational calculus is *safe* if every component of  $t$  appears in one of the relations, tuples, or constants that appear in  $P$ 
  - NOTE: this is more than just a syntax condition.
    - ▶ E.g.  $\{t \mid t[A] = 5 \vee \text{true}\}$  is not safe --- it defines an infinite set with attribute values that do not appear in any relation or tuples or constants in  $P$ .

CS425 – Fall 2016 – Boris Glavic 3.53 ©Silberschatz, Korth and Sudarshan




## Universal Quantification

- Find all students who have taken all courses offered in the Biology department
  - $\{t \mid \exists r \in student (t[ID] = r[ID]) \wedge (\forall u \in course (u[dept\_name] = "Biology" \Rightarrow \exists s \in takes (t[ID] = s[ID] \wedge s[course\_id] = u[course\_id]))\}$
  - Note that without the existential quantification on student, the above query would be unsafe if the Biology department has not offered any courses.


CS425 – Fall 2016 – Boris Glavic 3.54 ©Silberschatz, Korth and Sudarshan





## Domain Relational Calculus


CS425 – Fall 2016 – Boris Glavic 3.55  
©Silberschatz, Korth and Sudarshan



## Domain Relational Calculus

- A nonprocedural query language equivalent in power to the tuple relational calculus
- Each query is an expression 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
    - ▶ Variables that range of attribute values
  - $P$  represents a formula similar to that of the predicate calculus
  - Tuples can be formed using  $\langle \rangle$ 
    - ▶ E.g.,  $\langle \text{'Einstein'}, \text{'Physics'} \rangle$


CS425 – Fall 2016 – Boris Glavic 3.56  
©Silberschatz, Korth and Sudarshan



## Example Queries

- Find the *ID*, *name*, *dept\_name*, *salary* for instructors whose salary is greater than \$80,000
  - $\{ \langle i, n, d, s \rangle \mid \langle i, n, d, s \rangle \in \text{instructor} \wedge s > 80000 \}$
- As in the previous query, but output only the *ID* attribute value
  - $\{ \langle i \rangle \mid \langle i, n, d, s \rangle \in \text{instructor} \wedge s > 80000 \}$
- Find the names of all instructors whose department is in the Watson building
 
$$\{ \langle n \rangle \mid \exists i, d, s (\langle i, n, d, s \rangle \in \text{instructor} \wedge \exists b, a (\langle d, b, a \rangle \in \text{department} \wedge b = \text{"Watson"})) \}$$

CS425 – Fall 2016 – Boris Glavic 3.57  
©Silberschatz, Korth and Sudarshan




## Example Queries

- Find the set of all courses taught in the Fall 2009 semester, or in the Spring 2010 semester, or both
 
$$\{ \langle c \rangle \mid \exists a, s, y, b, r, t (\langle c, a, s, y, b, t \rangle \in \text{section} \wedge s = \text{"Fall"} \wedge y = 2009) \vee \exists a, s, y, b, r, t (\langle c, a, s, y, b, t \rangle \in \text{section} \wedge s = \text{"Spring"} \wedge y = 2010) \}$$

This case can also be written as

$$\{ \langle c \rangle \mid \exists a, s, y, b, r, t (\langle c, a, s, y, b, t \rangle \in \text{section} \wedge ((s = \text{"Fall"} \wedge y = 2009) \vee (s = \text{"Spring"} \wedge y = 2010))) \}$$
- Find the set of all courses taught in the Fall 2009 semester, and in the Spring 2010 semester
 
$$\{ \langle c \rangle \mid \exists a, s, y, b, r, t (\langle c, a, s, y, b, t \rangle \in \text{section} \wedge s = \text{"Fall"} \wedge y = 2009) \wedge \exists a, s, y, b, r, t (\langle c, a, s, y, b, t \rangle \in \text{section} \wedge s = \text{"Spring"} \wedge y = 2010) \}$$

CS425 – Fall 2016 – Boris Glavic 3.58  
©Silberschatz, Korth and Sudarshan



## Safety of Expressions


The expression:

$$\{ \langle x_1, x_2, \dots, x_n \rangle \mid P(x_1, x_2, \dots, x_n) \}$$

is safe if all of the following hold:

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

CS425 – Fall 2016 – Boris Glavic 3.59  
©Silberschatz, Korth and Sudarshan




## Universal Quantification

- Find all students who have taken all courses offered in the Biology department
  - $\{ \langle i \rangle \mid \exists n, d, tc (\langle i, n, d, tc \rangle \in \text{student} \wedge (\forall ci, ti, dn, cr (\langle ci, ti, dn, cr \rangle \in \text{course} \wedge dn = \text{"Biology"} \Rightarrow \exists si, se, y, g (\langle i, ci, si, se, y, g \rangle \in \text{takes}))) \}$
  - Note that without the existential quantification on student, the above query would be unsafe if the Biology department has not offered any courses.

\* Above query fixes bug in page 246, last query


CS425 – Fall 2016 – Boris Glavic 3.60  
©Silberschatz, Korth and Sudarshan



## Relationship between Relational Algebra and Tuple (Domain) Calculus


- **Codd's theorem**
  - Relational algebra and tuple calculus are equivalent in terms of expressiveness
- That means that every query expressible in relational algebra can also be expressed in tuple calculus and vice versa
- Since domain calculus is as expressive as tuple calculus the same holds for the domain calculus
- Note: Here relational algebra refers to the standard version (no aggregation and projection with functions)

CS425 – Fall 2016 – Boris Glavic 3.61 ©Silberschatz, Korth and Sudarshan



## End of Chapter 3


Modified from:  
Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## Recap

- **Query language**
  - **Declarative**
  - Retrieve, combine, and analyze data from a database instance
- **Relational algebra**
  - Standard relational algebra:
    - Selection, projection, renaming, cross product, union, set difference
  - Null values
  - Semantic sugar operators:
    - Intersection, joins, division,
  - Extensions:
    - Aggregation, extended projection
- **Tuple Calculus**
  - safety
- **Domain Calculus**


CS425 – Fall 2016 – Boris Glavic 3.63 ©Silberschatz, Korth and Sudarshan



## Outline

- Introduction
- Relational Data Model
- Formal Relational Languages (relational algebra)
- **SQL - Introduction**
- Database Design
- Transaction Processing, Recovery, and Concurrency Control
- Storage and File Structures
- Indexing and Hashing
- Query Processing and Optimization


CS425 – Fall 2016 – Boris Glavic 3.64 ©Silberschatz, Korth and Sudarshan



## Figure 6.01

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

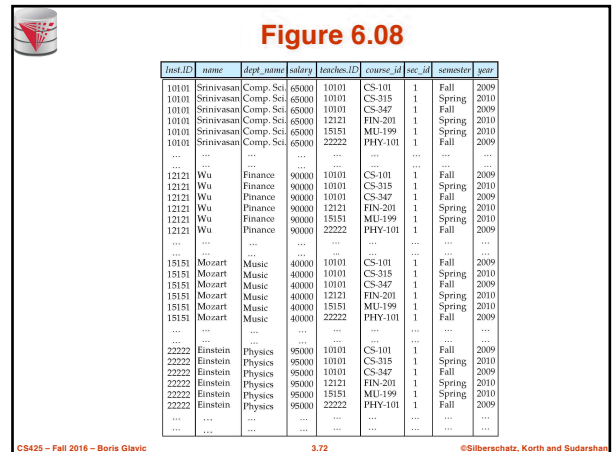
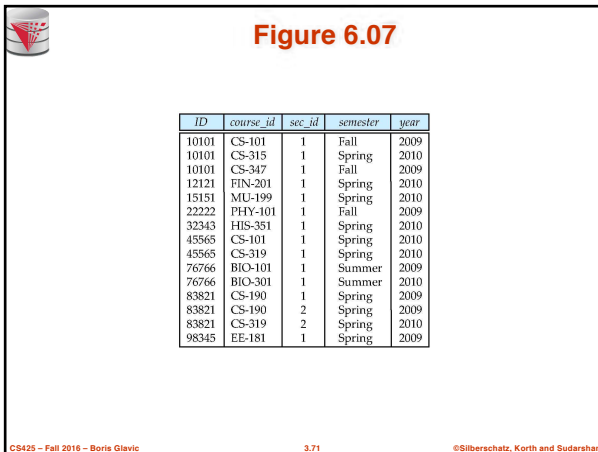
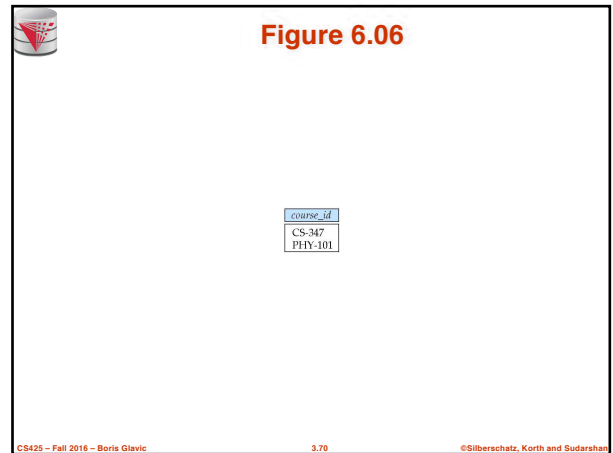
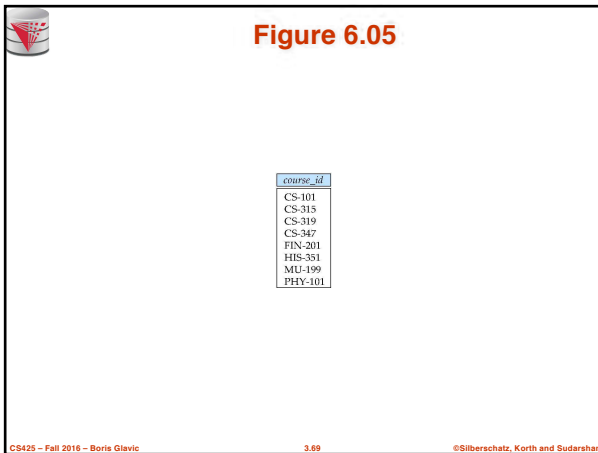
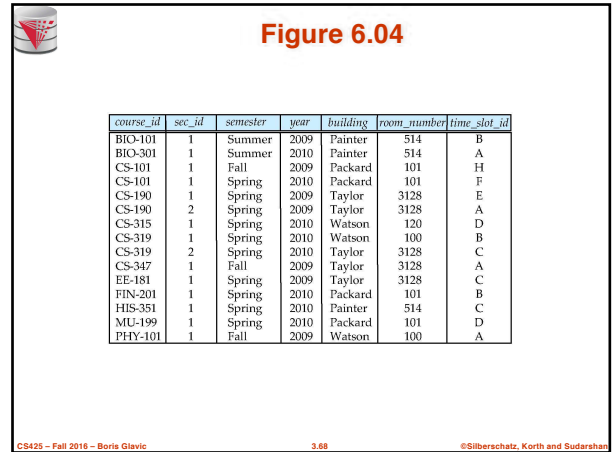
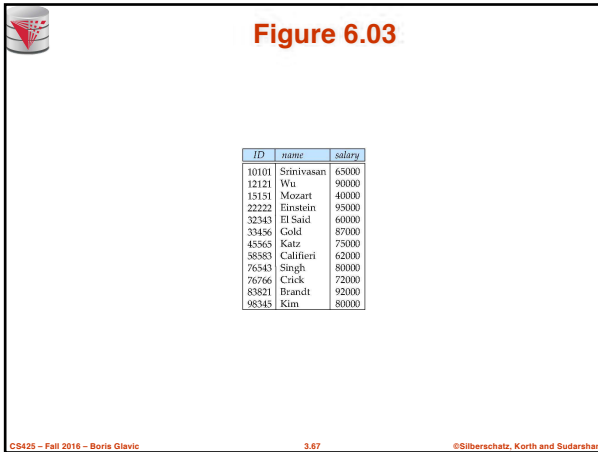
CS425 – Fall 2016 – Boris Glavic 3.65 ©Silberschatz, Korth and Sudarshan

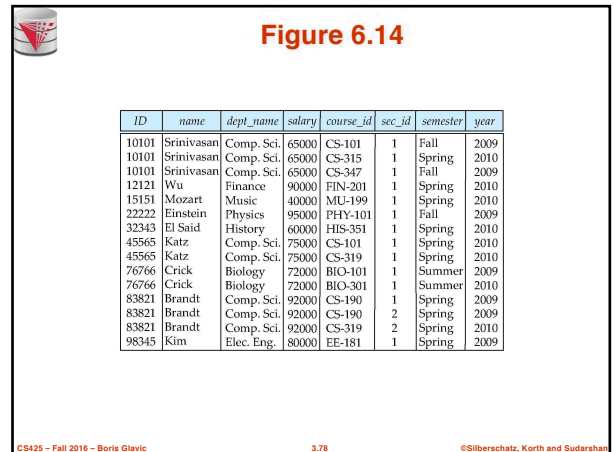
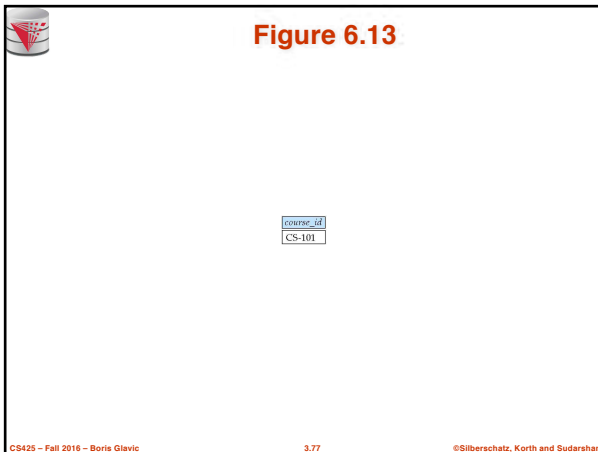
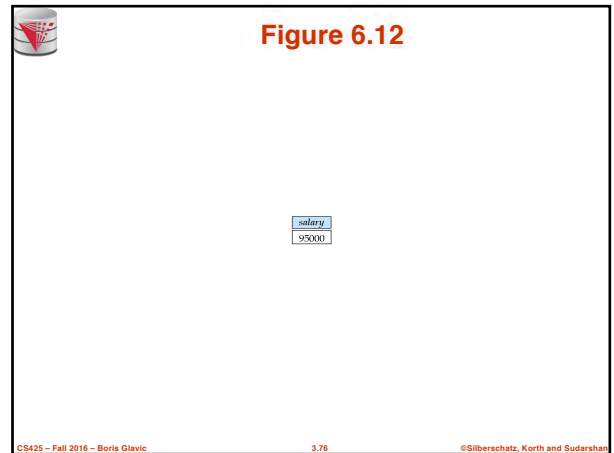
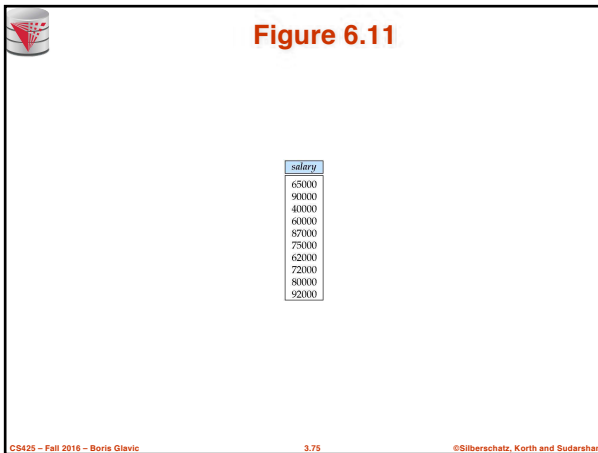
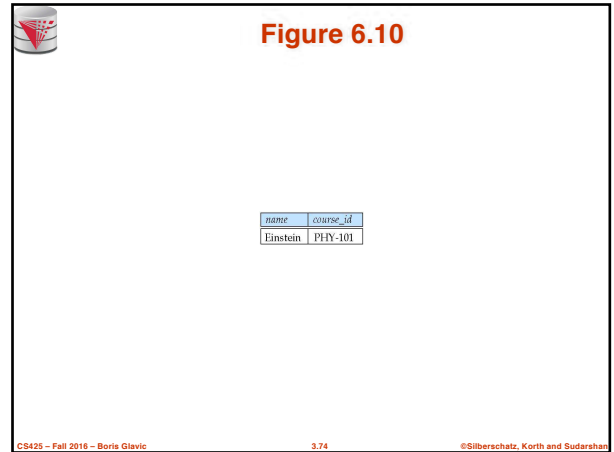
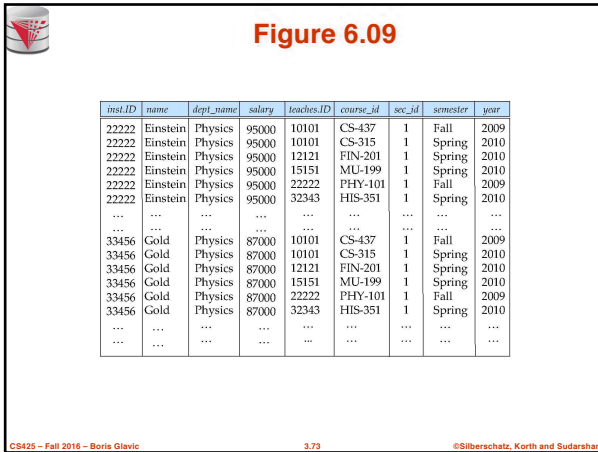


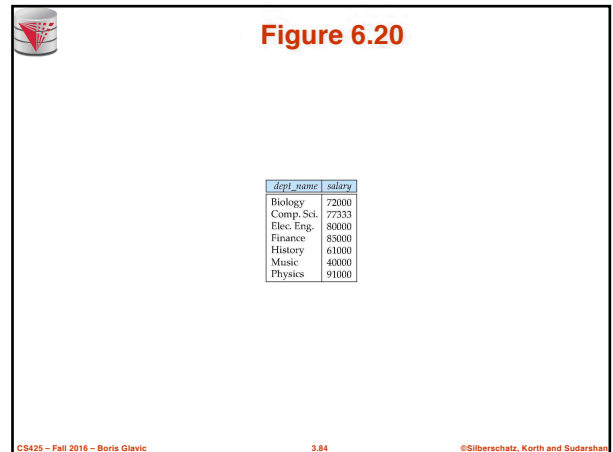
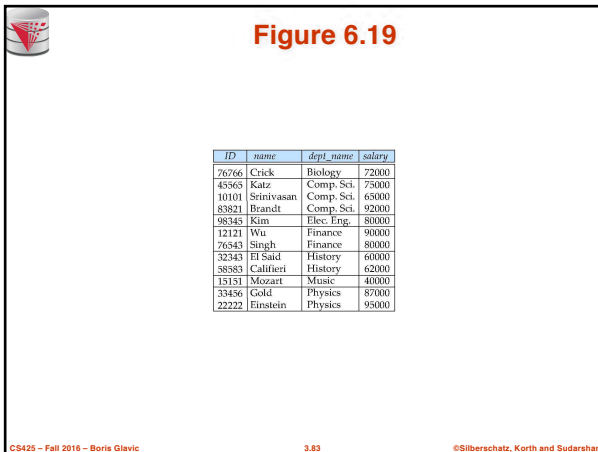
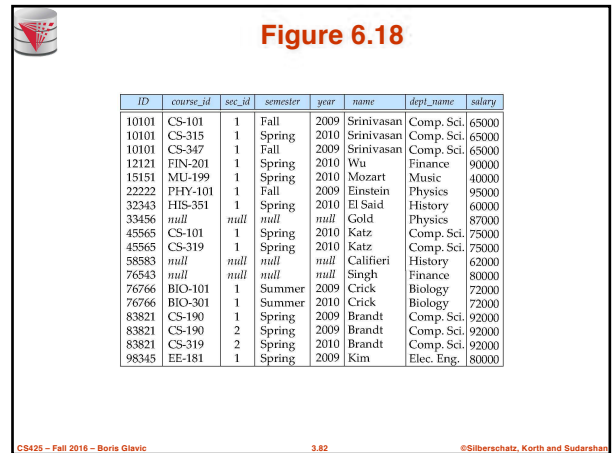
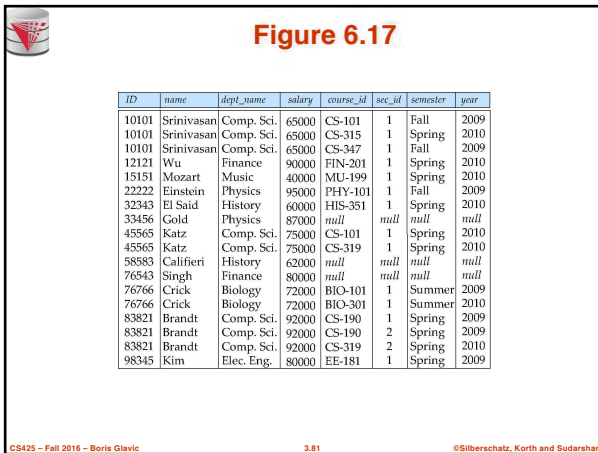
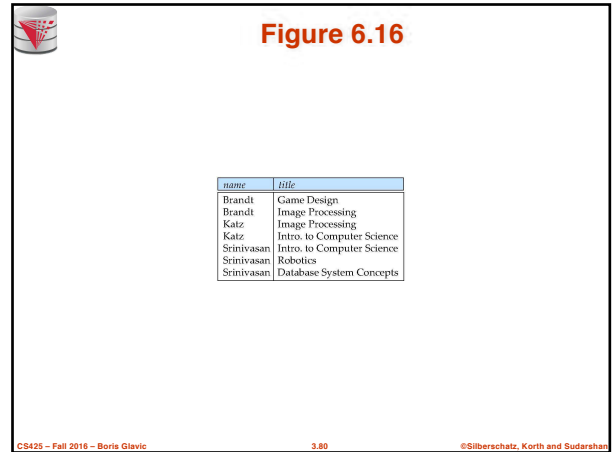
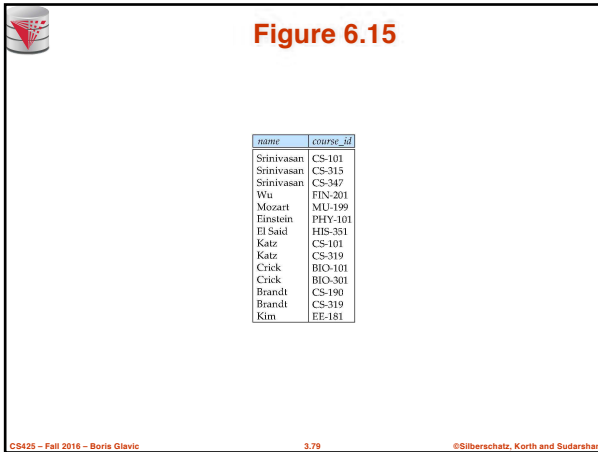
## Figure 6.02

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

CS425 – Fall 2016 – Boris Glavic 3.66 ©Silberschatz, Korth and Sudarshan







### Figure 6.21

name
Einstein
Crick
Gold

CS425 – Fall 2016 – Boris Glavic 3.85 ©Silberschatz, Korth and Sudarshan

### Deletion

- A delete request is expressed similarly to a query, except instead of displaying tuples to the user, the selected tuples are removed from the database.
- Can delete only whole tuples; cannot delete values on only particular attributes
- A deletion is expressed in relational algebra by:
 
$$r \leftarrow r - E$$
 where  $r$  is a relation and  $E$  is a relational algebra query.

CS425 – Fall 2016 – Boris Glavic 3.86 ©Silberschatz, Korth and Sudarshan

### Deletion Examples

- Delete all account records in the Perryridge branch.
 
$$account \leftarrow account - \sigma_{branch\_name = "Perryridge"}(account)$$
- Delete all loan records with amount in the range of 0 to 50
 
$$loan \leftarrow loan - \sigma_{amount \geq 0 \text{ and } amount \leq 50}(loan)$$
- Delete all accounts at branches located in Needham.
 
$$r_1 \leftarrow \sigma_{branch\_city = "Needham"}(account \bowtie branch)$$

$$r_2 \leftarrow \pi_{account\_number, branch\_name, balance}(r_1)$$

$$r_3 \leftarrow \pi_{customer\_name, account\_number}(r_1 \bowtie depositor)$$

$$account \leftarrow account - r_2$$

$$depositor \leftarrow depositor - r_3$$

CS425 – Fall 2016 – Boris Glavic 3.87 ©Silberschatz, Korth and Sudarshan

### Insertion

- To insert data into a relation, we either:
  - specify a tuple to be inserted
  - write a query whose result is a set of tuples to be inserted
- in relational algebra, an insertion is expressed by:
 
$$r \leftarrow r \cup E$$
 where  $r$  is a relation and  $E$  is a relational algebra expression.
- The insertion of a single tuple is expressed by letting  $E$  be a constant relation containing one tuple.

CS425 – Fall 2016 – Boris Glavic 3.88 ©Silberschatz, Korth and Sudarshan

### Insertion Examples

- Insert information in the database specifying that Smith has \$1200 in account A-973 at the Perryridge branch.
 
$$account \leftarrow account \cup \{("A-973", "Perryridge", 1200)\}$$

$$depositor \leftarrow depositor \cup \{("Smith", "A-973")\}$$
- Provide as a gift for all loan customers in the Perryridge branch, a \$200 savings account. Let the loan number serve as the account number for the new savings account.
 
$$r_1 \leftarrow (\sigma_{branch\_name = "Perryridge"}(borrower \bowtie loan))$$

$$account \leftarrow account \cup \pi_{loan\_number, branch\_name, 200}(r_1)$$

$$depositor \leftarrow depositor \cup \pi_{customer\_name, loan\_number}(r_1)$$

CS425 – Fall 2016 – Boris Glavic 3.89 ©Silberschatz, Korth and Sudarshan

### Updating

- A mechanism to change a value in a tuple without changing *all* values in the tuple
- Use the generalized projection operator to do this task
 
$$r \leftarrow \pi_{F_1, F_2, \dots, F_i, \dots, F_n}(r)$$
- Each  $F_i$  is either
  - the  $i^{\text{th}}$  attribute of  $r$ , if the  $i^{\text{th}}$  attribute is not updated, or,
  - if the attribute is to be updated  $F_i$  is an expression, involving only constants and the attributes of  $r$ , which gives the new value for the attribute

CS425 – Fall 2016 – Boris Glavic 3.90 ©Silberschatz, Korth and Sudarshan



## Update Examples

- Make interest payments by increasing all balances by 5 percent.

$$account \leftarrow \Pi_{account\_number, branch\_name, balance * 1.05}(account)$$

- Pay all accounts with balances over \$10,000 6 percent interest and pay all others 5 percent

$$account \leftarrow \Pi_{account\_number, branch\_name, balance * 1.06}(\sigma_{BAL > 10000}(account)) \cup \Pi_{account\_number, branch\_name, balance * 1.05}(\sigma_{BAL \leq 10000}(account))$$


## Example Queries

- Find the names of all customers who have a loan and an account at bank.

$$\Pi_{customer\_name}(borrower) \cap \Pi_{customer\_name}(depositor)$$

- Find the name of all customers who have a loan at the bank and the loan amount

$$\Pi_{customer\_name, loan\_number, amount}(borrower \bowtie loan)$$


## Example Queries

- Find all customers who have an account from at least the "Downtown" and the Uptown" branches.

- Query 1

$$\Pi_{customer\_name}(\sigma_{branch\_name = \text{"Downtown"}}(depositor \bowtie account)) \cap$$

$$\Pi_{customer\_name}(\sigma_{branch\_name = \text{"Uptown"}}(depositor \bowtie account))$$

- Query 2

$$\Pi_{customer\_name, branch\_name}(depositor \bowtie account) \div \rho_{temp}(branch\_name)({{"Downtown"}, {"Uptown"}})$$

Note that Query 2 uses a constant relation.



## Bank Example Queries

- Find all customers who have an account at all branches located in Brooklyn city.

$$\Pi_{customer\_name, branch\_name}(depositor \bowtie account)$$

$$\div \Pi_{branch\_name}(\sigma_{branch\_city = \text{"Brooklyn"}}(branch))$$



## CS425 – Fall 2016

### Boris Glavic

## Chapter 4: Introduction to SQL

Modified from:  
Database System Concepts, 6th Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## Chapter 4: Introduction to SQL

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



**Textbook: Chapter 3**

CS425 – Fall 2016 – Boris Glavic

4.2

©Silberschatz, Korth and Sudarshan



## History

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

CS425 – Fall 2016 – Boris Glavic

4.3

©Silberschatz, Korth and Sudarshan



## Data Definition Language

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

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

CS425 – Fall 2016 – Boris Glavic

4.4

©Silberschatz, Korth and Sudarshan



## Domain Types in SQL

- **char(n)**. Fixed length character string, with user-specified length  $n$ .
- **varchar(n)**. Variable length character strings, with user-specified maximum length  $n$ .
- **int**. Integer (a finite subset of the integers that is machine-dependent).
- **smallint**. Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d)**. Fixed point number, with user-specified precision of  $p$  digits, with  $n$  digits to the right of decimal point.
- **real, double precision**. Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n)**. Floating point number, with user-specified precision of at least  $n$  digits.
- More are covered in Chapter 4.

CS425 – Fall 2016 – Boris Glavic

4.5

©Silberschatz, Korth and Sudarshan



## Create Table Construct

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

```
create table r (A1 D1, A2 D2, ..., An Dn,
              (integrity-constraint1),
              ...,
              (integrity-constraintk))
```

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

- Example:

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

- **insert into instructor values** ('10211', 'Smith', 'Biology', 66000);
- **insert into instructor values** ('10211', null, 'Biology', 66000);

CS425 – Fall 2016 – Boris Glavic

4.6

©Silberschatz, Korth and Sudarshan





## Integrity Constraints in Create Table

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

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

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

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



## And a Few More Relation Definitions

- **create table** *student* (
  - $ID$  varchar(5),
  - $name$  varchar(20) not null,
  - $dept\_name$  varchar(20),
  - $tot\_cred$  numeric(3,0),
  - primary key** ( $ID$ ),
  - foreign key** ( $dept\_name$ ) **references** *department*);
- **create table** *takes* (
  - $ID$  varchar(5),
  - $course\_id$  varchar(8),
  - $sec\_id$  varchar(8),
  - $semester$  varchar(6),
  - $year$  numeric(4,0),
  - $grade$  varchar(2),
  - primary key** ( $ID, course\_id, sec\_id, semester, year$ ),
  - foreign key** ( $ID$ ) **references** *student*,
  - foreign key** ( $course\_id, sec\_id, semester, year$ ) **references** *section*);
- Note:  $sec\_id$  can be dropped from primary key above, to ensure a student cannot be registered for two sections of the same course in the same semester



## Even more

- **create table** *course* (
  - $course\_id$  varchar(8) **primary key**,
  - $title$  varchar(50),
  - $dept\_name$  varchar(20),
  - $credits$  numeric(2,0),
  - foreign key** ( $dept\_name$ ) **references** *department*);
- Primary key declaration can be combined with attribute declaration as shown above



## Drop and Alter Table Constructs

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



## Basic Query Structure

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

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

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



## The select Clause

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

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

## The select Clause (Cont.)

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

```
select distinct dept_name
from instructor
```
- The (redundant) keyword **all** specifies that duplicates not be removed.
 

```
select all dept_name
from instructor
```

CS425 – Fall 2016 – Boris Glavic 4.13 ©Silberschatz, Korth and Sudarshan

## The select Clause (Cont.)

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

```
select *
from instructor
```
- The **select** clause can contain arithmetic expressions involving the operation, +, -, \*, and /, and operating on constants or attributes of tuples.
  - Most systems also support additional functions
    - E.g., substring
  - Most systems allow user defined functions (UDFs)
- The query:
 

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

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

CS425 – Fall 2016 – Boris Glavic 4.14 ©Silberschatz, Korth and Sudarshan

## The from Clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *instructor X teaches*

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

  - generates every possible instructor – teaches pair, with all attributes from both relations
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra)

CS425 – Fall 2016 – Boris Glavic 4.15 ©Silberschatz, Korth and Sudarshan

## The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra.
- To find all instructors in Comp. Sci. dept with salary > 80000
 

```
select name
from instructor
where dept_name = 'Comp. Sci.' and salary > 80000
```
- Comparison results can be combined using the logical connectives **and**, **or**, and **not**.
- Comparisons can be applied to results of arithmetic expressions.
- SQL standard: any valid expression that returns a boolean result
  - Vendor specific restrictions may apply!

CS425 – Fall 2016 – Boris Glavic 4.16 ©Silberschatz, Korth and Sudarshan

## Cartesian Product: *instructor X teaches*

instructor				teaches				
ID	name	dept_name	salary	ID	course_id	sec_id	semester	year
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2009
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2010
15151	Mozart	Music	40000	10101	CS-347	1	Fall	2009
22222	Einstein	Physics	95000	12121	FIN-201	1	Spring	2010
32343	El Said	History	60000	15151	MU-199	1	Spring	2010
...	...	...	...	22222	PHY-101	1	Fall	2009

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

CS425 – Fall 2016 – Boris Glavic 4.17 ©Silberschatz, Korth and Sudarshan

## Joins

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

```
select name, course_id
from instructor, teaches
where instructor.ID = teaches.ID
```
- Find the course ID, semester, year and title of each course offered by the Comp. Sci. department
 

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

section
course_id
sec_id
semester
year
building
room_no
time_slot_id

→

course
course_id
title
dept_name
credits

CS425 – Fall 2016 – Boris Glavic 4.18 ©Silberschatz, Korth and Sudarshan

## Try Writing Some Queries in SQL

- Suggest queries to be written.....

CS425 – Fall 2016 – Boris Glavic 4.19 ©Silberschatz, Korth and Sudarshan

## Joined Relations

- **Join operations** take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the **from** clause

CS425 – Fall 2016 – Boris Glavic 4.20 ©Silberschatz, Korth and Sudarshan

## Join operations – Example

- Relation *course*

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

- Relation *prereq*

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

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

CS425 – Fall 2016 – Boris Glavic 4.21 ©Silberschatz, Korth and Sudarshan

## Natural Join

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

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

CS425 – Fall 2016 – Boris Glavic 4.22 ©Silberschatz, Korth and Sudarshan

## Natural Join Example

- List the names of instructors along with the course ID of the courses that they taught.
- **select name, course\_id**  
**from instructor, teaches**  
**where instructor.ID = teaches.ID;**
- **select name, course\_id**  
**from instructor natural join teaches;**

CS425 – Fall 2016 – Boris Glavic 4.23 ©Silberschatz, Korth and Sudarshan

## Natural Join (Cont.)

- Danger in natural join: beware of unrelated attributes with same name which get equated incorrectly
- List the names of instructors along with the titles of courses that they teach
  - Incorrect version (makes course.dept\_name = instructor.dept\_name)
    - **select name, title**  
**from instructor natural join teaches natural join course;**
  - Correct version
    - **select name, title**  
**from instructor natural join teaches, course**  
**where teaches.course\_id = course.course\_id;**
  - Another correct version
    - **select name, title**  
**from (instructor natural join teaches)**  
**join course using(course\_id);**

CS425 – Fall 2016 – Boris Glavic 4.24 ©Silberschatz, Korth and Sudarshan



## Outer Join

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

CS425 – Fall 2016 – Boris Glavic

4.25

©Silberschatz, Korth and Sudarshan



## Left Outer Join

- *course natural left outer join prereq*

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

CS425 – Fall 2016 – Boris Glavic

4.26

©Silberschatz, Korth and Sudarshan



## Right Outer Join

- *course natural right outer join prereq*

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

CS425 – Fall 2016 – Boris Glavic

4.27

©Silberschatz, Korth and Sudarshan



## Joined Relations

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

### Join types

inner join  
left outer join  
right outer join  
full outer join

### Join Conditions

natural  
on <predicate>  
using (A<sub>1</sub>, A<sub>1</sub>, ..., A<sub>n</sub>)

CS425 – Fall 2016 – Boris Glavic

4.28

©Silberschatz, Korth and Sudarshan



## Full Outer Join

- *course natural full outer join prereq*

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

CS425 – Fall 2016 – Boris Glavic

4.29

©Silberschatz, Korth and Sudarshan



## Joined Relations – Examples

- *course inner join prereq on course.course\_id = prereq.course\_id*

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

- What is the difference between the above, and a natural join?
- *course left outer join prereq on course.course\_id = prereq.course\_id*

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

CS425 – Fall 2016 – Boris Glavic

4.30

©Silberschatz, Korth and Sudarshan



## Joined Relations – Examples

- **course natural right outer join prereq**

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

- **course full outer join prereq using (course\_id)**

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

CS425 – Fall 2016 – Boris Glavic

4.31

©Silberschatz, Korth and Sudarshan



## The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause:  
*old-name as new-name*

- E.g.

- **select** ID, name, salary/12 **as** monthly\_salary  
**from** instructor

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

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

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

- Keyword **as** must be omitted in Oracle

CS425 – Fall 2016 – Boris Glavic

4.32

©Silberschatz, Korth and Sudarshan



## String Operations

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

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

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

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

- Match the string "100 %"

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

CS425 – Fall 2016 – Boris Glavic

4.33

©Silberschatz, Korth and Sudarshan



## String Operations (Cont.)

- Patterns are case sensitive.

- Pattern matching examples:

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

- SQL supports a variety of string operations such as

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

CS425 – Fall 2016 – Boris Glavic

4.34

©Silberschatz, Korth and Sudarshan



## Case Construct

- Like case, if, and ? Operators in programming languages

**case**

```
when c1 then e1
when c2 then e2
```

```
...
```

```
[else en]
```

**end**

- Each c<sub>i</sub> is a condition
- Each e<sub>i</sub> is an expression
- Returns the first e<sub>i</sub> for which c<sub>i</sub> evaluates to true
  - If none of the c<sub>i</sub> is true, then return e<sub>n</sub> (**else**)
    - ▶ If there is no else return null

CS425 – Fall 2016 – Boris Glavic

4.35

©Silberschatz, Korth and Sudarshan



## Case Construct Example

- Like case, if, and ? Operators in programming languages

**select**

```
name,
```

**case**

```
when salary > 1000000 then 'premium'
else 'standard'
```

```
end as customer_group
```

**from** customer

CS425 – Fall 2016 – Boris Glavic

4.36

©Silberschatz, Korth and Sudarshan



## Ordering the Display of Tuples

- List in alphabetic order the names of all instructors
 

```
select distinct name
from instructor
order by name
```
- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - Example: **order by name desc**
- Can sort on multiple attributes
  - Example: **order by dept\_name, name**
- Order is not expressible in the relational model!**

CS425 - Fall 2016 - Boris Glavic

4.37

©Silberschatz, Korth and Sudarshan



## Where Clause Predicates

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between \$90,000 and \$100,000 (that is,  $\geq 90,000$  and  $\leq 100,000$ )
  - select name**
  - from instructor**
  - where salary between 90000 and 100000**
- Tuple comparison
  - select name, course\_id**
  - from instructor, teaches**
  - where (instructor.ID, dept\_name) = (teaches.ID, 'Biology');**

CS425 - Fall 2016 - Boris Glavic

4.38

©Silberschatz, Korth and Sudarshan



## Set Operations

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

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

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

```
(select course_id from section where sem = 'Fall' and year = 2009)
except
(select course_id from section where sem = 'Spring' and year = 2010)
```

CS425 - Fall 2016 - Boris Glavic

4.39

©Silberschatz, Korth and Sudarshan



## Set Operations

- Set operations **union**, **intersect**, and **except**
    - Each of the above operations automatically eliminates duplicates
  - To retain all duplicates use the corresponding multiset versions **union all**, **intersect all** and **except all**.
- Suppose a tuple occurs  $m$  times in  $r$  and  $n$  times in  $s$ , then, it occurs:
- $m + n$  times in  **$r$  union all  $s$**
  - $\min(m, n)$  times in  **$r$  intersect all  $s$**
  - $\max(0, m - n)$  times in  **$r$  except all  $s$**

CS425 - Fall 2016 - Boris Glavic

4.40

©Silberschatz, Korth and Sudarshan



## Null Values

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

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

CS425 - Fall 2016 - Boris Glavic

4.41

©Silberschatz, Korth and Sudarshan



## Null Values and Three Valued Logic

- Any comparison with *null* returns *null*
  - Example:  $5 < null$  or  $null <= null$  or  $null = null$
- Three-valued logic using the truth value *null*:
  - OR:  $(null \text{ or } true) = true$ ,  
 $(null \text{ or } false) = null$   
 $(null \text{ or } null) = null$
  - AND:  $(true \text{ and } null) = null$ ,  
 $(false \text{ and } null) = false$ ,  
 $(null \text{ and } null) = null$
  - NOT:  $(\text{not } null) = null$
  - " $P$  is null" evaluates to true if predicate  $P$  evaluates to *null*
- Result of **where** clause predicate is treated as *false* if it evaluates to *null*

CS425 - Fall 2016 - Boris Glavic

4.42

©Silberschatz, Korth and Sudarshan



## Aggregate Functions

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

CS425 – Fall 2016 – Boris Glavic

4.43

©Silberschatz, Korth and Sudarshan



## Aggregate Functions (Cont.)

- Find the average salary of instructors in the Computer Science department
  - select avg (salary)**  
**from instructor**  
**where dept\_name= 'Comp. Sci.' ;**
- Find the total number of instructors who teach a course in the Spring 2010 semester
  - select count (distinct ID)**  
**from teaches**  
**where semester = 'Spring' and year = 2010**
- Find the number of tuples in the *course* relation
  - select count (\*)**  
**from course;**

CS425 – Fall 2016 – Boris Glavic

4.44

©Silberschatz, Korth and Sudarshan



## Aggregate Functions – Group By

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

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

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

CS425 – Fall 2016 – Boris Glavic

4.45

©Silberschatz, Korth and Sudarshan



## Aggregation (Cont.)

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list
  - /\* erroneous query \*/*  
**select dept\_name, ID, avg (salary)**  
**from instructor**  
**group by dept\_name;**

CS425 – Fall 2016 – Boris Glavic

4.46

©Silberschatz, Korth and Sudarshan



## Aggregate Functions – Having Clause

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

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

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

CS425 – Fall 2016 – Boris Glavic

4.47

©Silberschatz, Korth and Sudarshan



## Null Values and Aggregates

- Total all salaries
  - select sum (salary)**  
**from instructor**
  - Above statement ignores null amounts
  - Result is *null* if there is no non-null amount
- All aggregate operations except **count(\*)** ignore tuples with null values on the aggregated attributes
- What if collection has only null values?
  - count returns 0
  - all other aggregates return null

CS425 – Fall 2016 – Boris Glavic

4.48

©Silberschatz, Korth and Sudarshan

## Empty Relations and Aggregates

- What if the input relation is empty
- Conventions:
  - **sum**: returns *null*
  - **avg**: returns *null*
  - **min**: returns *null*
  - **max**: returns *null*
  - **count**: returns 0

CS425 – Fall 2016 – Boris Glavic 4.49 ©Silberschatz, Korth and Sudarshan

## Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- **Multiset (bag semantics)** versions of some of the relational algebra operators – given multiset relations  $r_1$  and  $r_2$ :
  1.  $\sigma_{\theta}(r_1)$ : If there are  $c_1$  copies of tuple  $t_1$  in  $r_1$ , and  $t_1$  satisfies selections  $\sigma_{\theta}$ , then there are  $c_1$  copies of  $t_1$  in  $\sigma_{\theta}(r_1)$ .
  2.  $\Pi_A(r)$ : For each copy of tuple  $t_i$  in  $r_1$ , there is a copy of tuple  $\Pi_A(t_i)$  in  $\Pi_A(r_1)$  where  $\Pi_A(t_i)$  denotes the projection of the single tuple  $t_i$ .
  3.  $r_1 \times r_2$ : If there are  $c_1$  copies of tuple  $t_1$  in  $r_1$  and  $c_2$  copies of tuple  $t_2$  in  $r_2$ , there are  $c_1 \times c_2$  copies of the tuple  $t_1 \times t_2$  in  $r_1 \times r_2$ .

CS425 – Fall 2016 – Boris Glavic 4.50 ©Silberschatz, Korth and Sudarshan

## Multiset Relational Algebra

- Pure relational algebra operates on **set-semantics** (no duplicates allowed)
  - e.g. after projection
- Multiset (**bag-semantics**) relational algebra retains duplicates, to match SQL semantics
  - SQL duplicate retention was initially for efficiency, but is now a feature
- Multiset relational algebra defined as follows
  - **selection**: has as many duplicates of a tuple as in the input, if the tuple satisfies the selection
  - **projection**: one tuple per input tuple, even if it is a duplicate
  - **cross product**: If there are  $m$  copies of  $t_1$  in  $r$ , and  $n$  copies of  $t_2$  in  $s$ , there are  $m \times n$  copies of  $t_1.t_2$  in  $r \times s$
  - Other operators similarly defined
    - ↳ E.g. **union**:  $m + n$  copies, **intersection**:  $\min(m, n)$  copies
    - difference**:  $\max(0, m - n)$  copies

CS425 – Fall 2016 – Boris Glavic 4.51 ©Silberschatz, Korth and Sudarshan

## Duplicates (Cont.)

- Example: Suppose multiset relations  $r_1 (A, B)$  and  $r_2 (C)$  are as follows:
 
$$r_1 = \{(1, a) (2, a)\} \quad r_2 = \{(2), (3), (3)\}$$
- Then  $\Pi_B(r_1)$  would be  $\{(a), (a)\}$ , while  $\Pi_B(r_1) \times r_2$  would be  $\{(a,2), (a,2), (a,3), (a,3), (a,3), (a,3)\}$
- SQL duplicate semantics:
 

```

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

 is equivalent to the *multiset* version of the expression:
 
$$\Pi_{A_1, A_2, \dots, A_n}(\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$

CS425 – Fall 2016 – Boris Glavic 4.52 ©Silberschatz, Korth and Sudarshan

## SQL and Relational Algebra

- ```

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

 is equivalent to the following expression in multiset relational algebra
 
$$\Pi_{A_1, \dots, A_n}(\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$
- ```

select A1, A2, sum(A3)
from r1, r2, ..., rm
where P
group by A1, A2
      
```

 is equivalent to the following expression in multiset relational algebra
 
$$A_1, A_2 \int \text{sum}(A_3) (\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$

CS425 – Fall 2016 – Boris Glavic 4.53 ©Silberschatz, Korth and Sudarshan

## SQL and Relational Algebra

- More generally, the non-aggregated attributes in the **select** clause may be a subset of the **group by** attributes, in which case the equivalence is as follows:
 

```

select A1, sum(A3) AS sumA3
from r1, r2, ..., rm
where P
group by A1, A2
      
```

 is equivalent to the following expression in multiset relational algebra
 
$$\Pi_{A_1, \text{sum}A_3}(A_1, A_2 \int \text{sum}(A_3) \text{ as } \text{sum}A_3 (\sigma_P(r_1 \times r_2 \times \dots \times r_m)))$$

CS425 – Fall 2016 – Boris Glavic 4.54 ©Silberschatz, Korth and Sudarshan





## Subqueries in the From Clause

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

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

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

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

CS425 - Fall 2016 - Boris Glavic 4.55 ©Silberschatz, Korth and Sudarshan



## Nested Subqueries

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

CS425 - Fall 2016 - Boris Glavic 4.56 ©Silberschatz, Korth and Sudarshan



## Example Query

- Find courses offered in Fall 2009 and in Spring 2010

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

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

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

CS425 - Fall 2016 - Boris Glavic 4.57 ©Silberschatz, Korth and Sudarshan



## Example Query

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

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

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

CS425 - Fall 2016 - Boris Glavic 4.58 ©Silberschatz, Korth and Sudarshan



## Quantification

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

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

- Same query using **> some** clause

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

CS425 - Fall 2016 - Boris Glavic 4.59 ©Silberschatz, Korth and Sudarshan



## Definition of Some Clause

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

$(5 < \text{some } \begin{matrix} 0 \\ 5 \\ 6 \end{matrix}) = \text{true}$  (read: 5 < some tuple in the relation)

$(5 < \text{some } \begin{matrix} 0 \\ 5 \end{matrix}) = \text{false}$

$(5 = \text{some } \begin{matrix} 0 \\ 5 \end{matrix}) = \text{true}$

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

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

CS425 - Fall 2016 - Boris Glavic 4.60 ©Silberschatz, Korth and Sudarshan



## Example Query

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

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

CS425 – Fall 2016 – Boris Glavic

4.61

©Silberschatz, Korth and Sudarshan



## Definition of all Clause

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

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

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

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

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

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

CS425 – Fall 2016 – Boris Glavic

4.62

©Silberschatz, Korth and Sudarshan



## Test for Empty Relations

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

CS425 – Fall 2016 – Boris Glavic

4.63

©Silberschatz, Korth and Sudarshan



## Correlation Variables

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

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

- Correlated subquery**
- Correlation name** or **correlation variable**

CS425 – Fall 2016 – Boris Glavic

4.64

©Silberschatz, Korth and Sudarshan



## Not Exists

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

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

- Note that  $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note: Cannot write this query using = all and its variants

CS425 – Fall 2016 – Boris Glavic

4.65

©Silberschatz, Korth and Sudarshan



## Test for Absence of Duplicate Tuples

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

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

CS425 – Fall 2016 – Boris Glavic

4.66

©Silberschatz, Korth and Sudarshan



## Correlated Subqueries in the From Clause

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

```
select name, salary, avg_salary
from instructor I1,
     lateral (select avg(salary) as avg_salary
             from instructor I2
             where I2.dept_name= I1.dept_name);
```
- Lateral clause permits later part of the **from** clause (after the lateral keyword) to access correlation variables from the earlier part.
- Note: lateral is part of the SQL standard, but is not supported on many database systems; some databases such as SQL Server offer alternative syntax

CS425 – Fall 2016 – Boris Glavic

4.67

©Silberschatz, Korth and Sudarshan



## With Clause

- The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the maximum budget

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

CS425 – Fall 2016 – Boris Glavic

4.68

©Silberschatz, Korth and Sudarshan



## Complex Queries using With Clause

- With clause is very useful for writing complex queries
- Supported by most database systems, with minor syntax variations
- Find all departments where the total salary is greater than the average of the total salary at all departments

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

CS425 – Fall 2016 – Boris Glavic

4.69

©Silberschatz, Korth and Sudarshan



## Scalar Subquery

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

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

```
select name
 from instructor
 where salary * 10 >
       (select budget from department
        where department.dept_name = instructor.dept_name)
```
- Runtime error if subquery returns more than one result tuple

CS425 – Fall 2016 – Boris Glavic

4.70

©Silberschatz, Korth and Sudarshan



## Query Features Recap - Syntax

- An SQL query is either a Select-from-where block or a set operation
- An SQL query block is structured like this:
 

```
SELECT [DISTINCT] select_list
[FROM from_list]
[WHERE where_condition]
[GROUP BY group_by_list]
[HAVING having_condition]
[ORDER BY order_by_list]
```
- Set operations
 

```
[Query Block] set_op [Query Block]
set_op: [ALL] UNION | INTERSECT | EXCEPT
```

CS425 – Fall 2016 – Boris Glavic

4.71

©Silberschatz, Korth and Sudarshan




## Query Features Recap - Syntax

- Almost all clauses are optional
- Examples:
  - SELECT \* FROM r;**
  - SELECT 1;**
    - Convention: returns single tuple
  - SELECT 'ok' FROM accounts HAVING sum(balance) = 0;**
  - SELECT 1 GROUP BY 1;**
  - SELECT 1 HAVING true;**
  - Let *r* be a relation with two attributes *a* and *b*
    - SELECT a,b FROM r**
      - WHERE a IN (SELECT a FROM r) AND b IN (SELECT b FROM r)**
      - GROUP BY a,b HAVING count(\*) > 0;**
- Note:
  - Not all systems support all of this "non-sense"

CS425 – Fall 2016 – Boris Glavic

4.72


©Silberschatz, Korth and Sudarshan



## Syntax - SELECT

- **SELECT** [DISTINCT [ON (distinct\_list)]] select\_list
- select\_list
  - List of projection expressions
    - [expr] [AS name]
  - expr
    - Expression over attributes, constants, arithmetic operators, functions, **CASE**-construct, aggregation functions
- distinct\_list
  - List of expressions
- Examples:
  - **SELECT DISTINCT ON** (a % 2) a **FROM** r;
  - **SELECT** substring(a, 1,2) **AS** x **FROM** r;
  - **SELECT CASE WHEN** a = 2 **THEN** a **ELSE null END AS** b **FROM** r;
  - **SELECT** a = b **AS** is\_a\_equal\_to\_b **FROM** r;


CS425 – Fall 2016 – Boris Glavic 4.73 ©Silberschatz, Korth and Sudarshan



## Syntax - FROM

- **FROM** from\_list
- from\_list
  - List of from clause expressions
    - subquery | relation | constant\_relation | join\_expr [alias]
  - subquery
    - Any valid SQL query – alias is not optional
  - relation
    - A relation in the database
  - constant\_relation
    - (**VALUES** tuples) – alias is not optional
  - join\_expr
    - joins between from\_clause entries
  - alias
    - [AS] b [(attribute\_name\_list)]


CS425 – Fall 2016 – Boris Glavic 4.74 ©Silberschatz, Korth and Sudarshan



## Syntax – FROM (cont.)

- Examples (relation r with attributes a and b):
  - **SELECT \* FROM** r;
  - **SELECT \* FROM** r **AS** g(v,w);
  - **SELECT \* FROM** r x;
  - **SELECT \* FROM (VALUES (1,2), (3,1)) AS** s(u,v);
  - **SELECT \* FROM** r **NATURAL JOIN** s, t;
  - **SELECT \* FROM ((r JOIN s ON (r.a = s.c)) NATURAL JOIN (SELECT \* FROM t) AS new);**
  - **SELECT \* FROM (SELECT \* FROM r) AS r;**
  - **SELECT \* FROM (SELECT \* FROM (SELECT \* FROM r) AS r) AS r;**


CS425 – Fall 2016 – Boris Glavic 4.75 ©Silberschatz, Korth and Sudarshan



## Syntax - WHERE

- **WHERE** where\_condition
- where\_condition: A boolean expression over
  - Attributes
  - Constants: e.g., true, 1, 0.5, 'hello'
  - Comparison operators: =, <, >, **IS DISTINCT FROM**, **IS NULL**, ...
  - Arithmetic operators: +, -, /, %
  - Function calls
  - Nested subquery expressions
- Examples
  - **SELECT \* FROM** r **WHERE** a = 2;
  - **SELECT \* FROM** r **WHERE** true **OR** false;
  - **SELECT \* FROM** r **WHERE NOT**(a = 2 **OR** a = 3);
  - **SELECT \* FROM** r **WHERE** a **IS DISTINCT FROM** b;
  - **SELECT \* FROM** r **WHERE** a < **ANY** (SELECT c **FROM** s);
  - **SELECT \* FROM** r **WHERE** a = (SELECT count(\*) **FROM** s);


CS425 – Fall 2016 – Boris Glavic 4.76 ©Silberschatz, Korth and Sudarshan



## Syntax – GROUP BY

- **GROUP BY** group\_by\_list
- group\_by\_list
  - List of expressions
    - Expression over attributes, constants, arithmetic operators, functions, **CASE**-construct, aggregation functions
- Examples:
  - **SELECT** sum(a), b **FROM** r **GROUP BY** b;
  - **SELECT** sum(a), b, c **FROM** r **GROUP BY** b, c;
  - **SELECT** sum(a), b/2 **FROM** r **GROUP BY** b/2;
  - **SELECT** sum(a), b **FROM** r **GROUP BY** b > 5;
  - Incorrect, cannot select b, because it is not an expression in the group by clause
  - **SELECT** sum(a), b **FROM** r **GROUP BY** b **IN** (SELECT c **FROM** s);


CS425 – Fall 2016 – Boris Glavic 4.77 ©Silberschatz, Korth and Sudarshan



## Syntax – HAVING

- **HAVING** having\_condition
- having\_condition
  - Like where\_condition except that expressions over attributes have either to be in the **GROUP BY** clause or are aggregated
- Examples:
  - **SELECT** sum(a), b **FROM** r **GROUP BY** b **HAVING** sum(a) > 10;
  - **SELECT** sum(a), b **FROM** r **GROUP BY** b **HAVING** sum(a) + 5 > 10;
  - **SELECT** sum(a), b **FROM** r **GROUP BY** b **HAVING** true;
  - **SELECT** sum(a), b **FROM** r **GROUP BY** b **HAVING** count(\*) = 50;
  - **SELECT** b **FROM** r **GROUP BY** b **HAVING** sum(a) > 10;


CS425 – Fall 2016 – Boris Glavic 4.78 ©Silberschatz, Korth and Sudarshan



## Syntax – ORDER BY

- **ORDER BY** order\_by\_list
- order\_by\_list
  - Like select\_list minus renaming
  - Optional [**ASC** | **DESC**] for each item
- Examples:
  - **SELECT \* FROM r ORDER BY a;**
  - **SELECT \* FROM r ORDER BY b, a;**
  - **SELECT \* FROM r ORDER BY a \* 2;**
  - **SELECT \* FROM r ORDER BY a \* 2, a;**
  - **SELECT \* FROM r ORDER BY a + (SELECT count(\*) FROM s);**


CS425 – Fall 2016 – Boris Glavic 4.79 ©Silberschatz, Korth and Sudarshan



## Query Semantics

- Evaluation Algorithm (you can do it manually – sort of)
- 1. Compute **FROM** clause
  1. Compute cross product of all items in the **FROM** clause
    - ▶ Relations: nothing to do
    - ▶ Subqueries: use this algorithm to recursively compute the result of subqueries first
    - ▶ Join expressions: compute the join
- 2. Compute **WHERE** clause
  1. For each tuple in the result of 1. evaluate the **WHERE** clause condition
- 3. Compute **GROUP BY** clause
  1. Group the results of step 2. on the **GROUP BY** expressions
- 4. Compute **HAVING** clause
  1. For each group (if any) evaluate the **HAVING** condition

CS425 – Fall 2016 – Boris Glavic 4.80 ©Silberschatz, Korth and Sudarshan




## Query Semantics (Cont.)

5. Compute **ORDER BY** clause
  5. Order the result of step 4 on the **ORDER BY** expressions
6. Compute **SELECT** clause
  5. Project each result tuple from step 5 on the **SELECT** expressions

- If the **WHERE**, **SELECT**, **GROUP BY**, **HAVING**, **ORDER BY** clauses have any nested subqueries
  - For each tuple t in the result of the **FROM** clause
    - ▶ Substitute the correlated attributes with values from t
    - ▶ Evaluate the resulting query
    - ▶ Use the result to evaluate the expression in the clause the subquery occurs in


CS425 – Fall 2016 – Boris Glavic 4.81 ©Silberschatz, Korth and Sudarshan



## Query Semantics (Cont.)

- Equivalent relational algebra expression
  - **ORDER BY** has no equivalent, because relations are unordered
  - Nested subqueries: need to extend algebra (not covered here)
- Each query block is equivalent to
 
$$\pi(\sigma(\mathcal{G}(\pi(\sigma(F_1 \times \dots \times F_n))))))$$
  - Where  $F_i$  is the translation of the  $i^{\text{th}}$  **FROM** clause item
  - Note: we leave out the arguments


CS425 – Fall 2016 – Boris Glavic 4.82 ©Silberschatz, Korth and Sudarshan



## Modification of the Database

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

CS425 – Fall 2016 – Boris Glavic 4.83 ©Silberschatz, Korth and Sudarshan



## Modification of the Database – Deletion

- Delete all instructors
 

```
delete from instructor
```
- Delete all instructors from the Finance department
 

```
delete from instructor
where dept_name = 'Finance';
```
- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building.
 

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

CS425 – Fall 2016 – Boris Glavic 4.84 ©Silberschatz, Korth and Sudarshan



## Deletion (Cont.)

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

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

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

CS425 – Fall 2016 – Boris Glavic

4.85

©Silberschatz, Korth and Sudarshan



## Modification of the Database – Insertion

- Add a new tuple to *course*

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

```
insert into course (course_id, title, dept_name, credits)
values (' CS-437', ' Database Systems', ' Comp. Sci.', 4);
```
- Add a new tuple to *student* with *tot\_creds* set to null
 

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

CS425 – Fall 2016 – Boris Glavic

4.86

©Silberschatz, Korth and Sudarshan



## Insertion (Cont.)

- Add all instructors to the *student* relation with *tot\_creds* set to 0
 

```
insert into student
select ID, name, dept_name, 0
from instructor
```
- The **select from where** statement is evaluated fully before any of its results are inserted into the relation (otherwise queries like **insert into table1 select \* from table1** would cause problems, if *table1* did not have any primary key defined.

CS425 – Fall 2016 – Boris Glavic

4.87

©Silberschatz, Korth and Sudarshan



## Modification of the Database – Updates

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

```
update instructor
set salary = salary * 1.03
where salary > 100000;
update instructor
set salary = salary * 1.05
where salary <= 100000;
```
  - The order is important
  - Can be done better using the **case** statement (next slide)

CS425 – Fall 2016 – Boris Glavic

4.88

©Silberschatz, Korth and Sudarshan



## Case Statement for Conditional Updates

- Same query as before but with case statement
 

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

CS425 – Fall 2016 – Boris Glavic

4.89

©Silberschatz, Korth and Sudarshan



## Updates with Scalar Subqueries

- Recompute and update *tot\_creds* value for all students
 


```
update student S
set tot_cred = ( select sum(credits)
from takes natural join course
where S.ID= takes.ID and
takes.grade <> ' F' and
takes.grade is not null);
```
- Sets *tot\_creds* to null for students who have not taken any course
- Instead of **sum(credits)**, use:
 

```
case
when sum(credits) is not null then sum(credits)
else 0
end
```
- Or **COALESCE(sum(credits),0)**
  - **COALESCE** returns first non-null arguments

CS425 – Fall 2016 – Boris Glavic

4.90


©Silberschatz, Korth and Sudarshan



## Recap


- SQL queries
  - Clauses: **SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY**
  - Nested subqueries
  - Equivalence with relational algebra
- SQL update, inserts, deletes
  - Semantics of referencing updated relation in **WHERE**
- SQL DDL
  - Table definition: **CREATE TABLE**

CS425 – Fall 2016 – Boris Glavic 4.91 ©Silberschatz, Korth and Sudarshan



## End of Chapter 4


Modified from:  
Database System Concepts, 6th Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## Outline

- Introduction
- Relational Data Model
- Formal Relational Languages (relational algebra)
- **SQL - Intermediate**
- Database Design
- Transaction Processing, Recovery, and Concurrency Control
- Storage and File Structures
- Indexing and Hashing
- Query Processing and Optimization


CS425 – Fall 2016 – Boris Glavic 4.93 ©Silberschatz, Korth and Sudarshan



## Advanced SQL Features\*\*

- Create a table with the same schema as an existing table:  
**create table temp\_account like account**


CS425 – Fall 2016 – Boris Glavic 4.94 ©Silberschatz, Korth and Sudarshan



## Figure 3.02

name
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califiori
Singh
Crick
Brandt
Kim

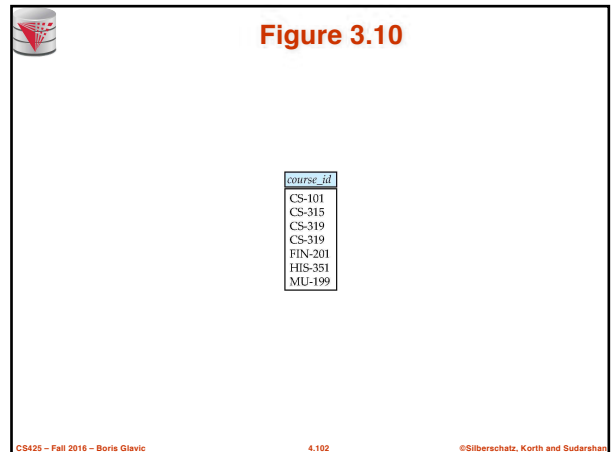
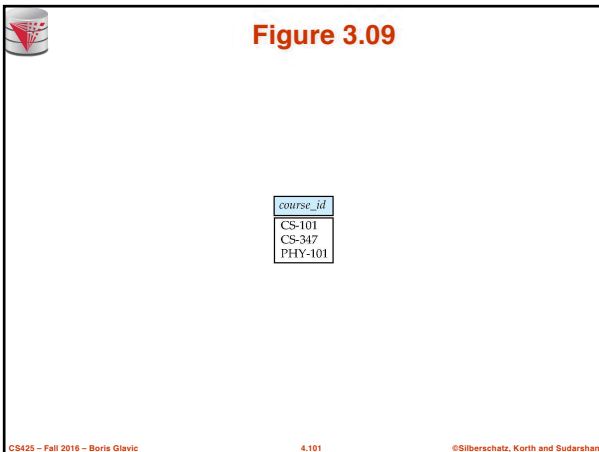
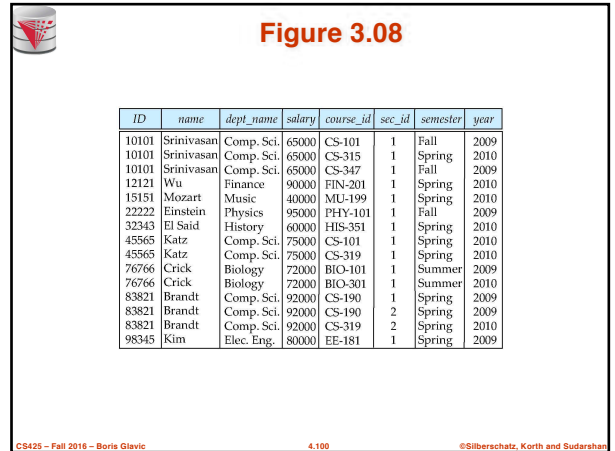
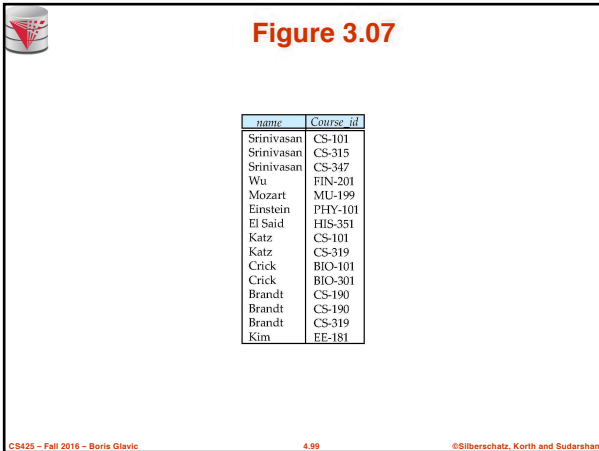
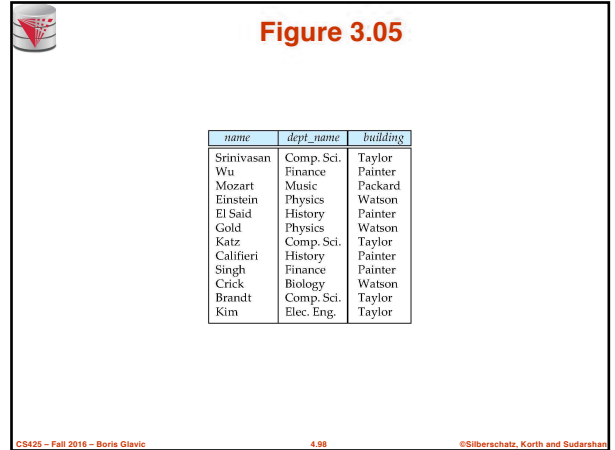
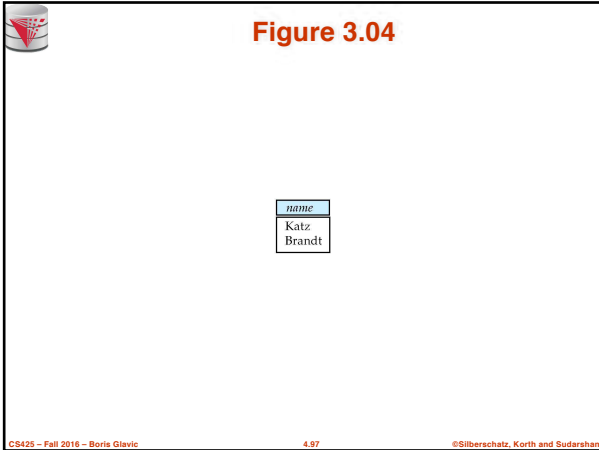
CS425 – Fall 2016 – Boris Glavic 4.95 ©Silberschatz, Korth and Sudarshan



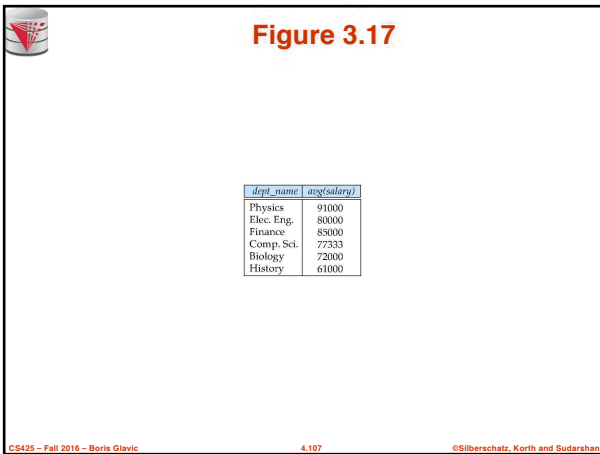
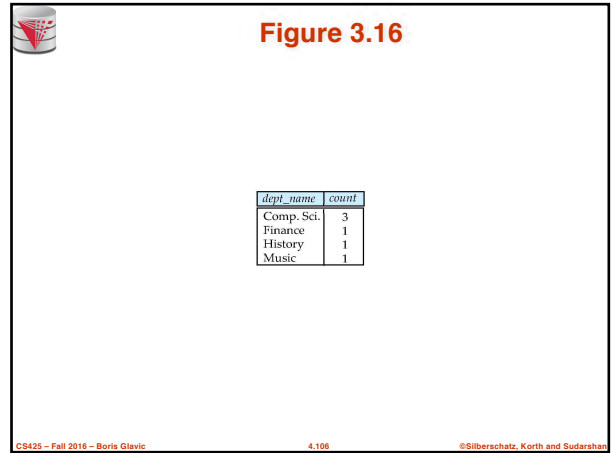
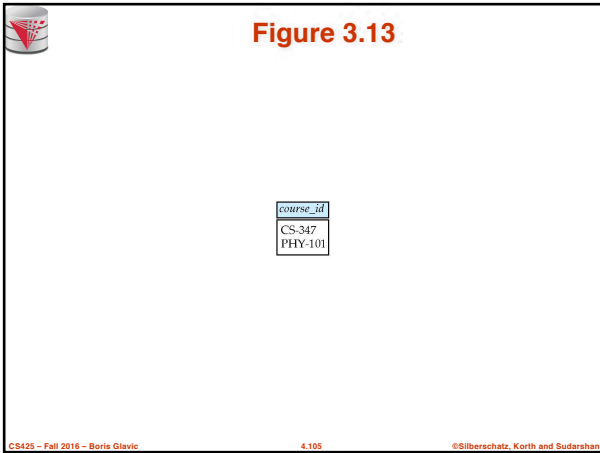
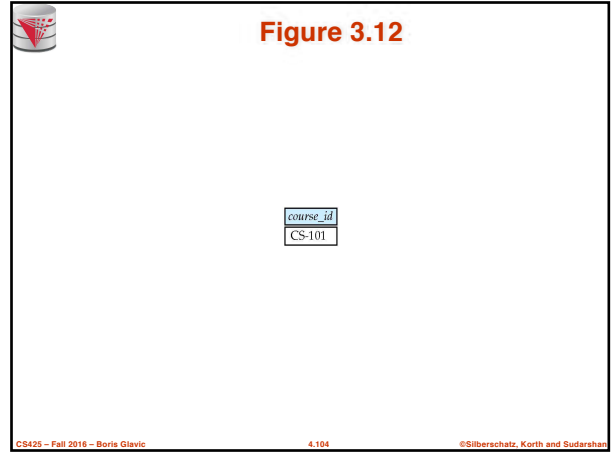
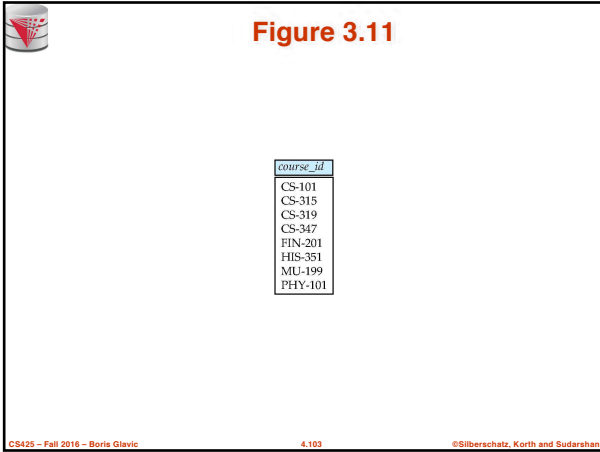
## Figure 3.03

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

CS425 – Fall 2016 – Boris Glavic 4.96 ©Silberschatz, Korth and Sudarshan









## CS425 – Fall 2013 Boris Glavic Chapter 5: Intermediate SQL

modified from:  
Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on reuse



## Chapter 5: Intermediate SQL

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



## Textbook: Chapter 4

CS425 – Fall 2013 – Boris Glavic

5.2

©Silberschatz, Korth and Sudarshan



## Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

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

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.

CS425 – Fall 2013 – Boris Glavic

5.3

©Silberschatz, Korth and Sudarshan



## View Definition

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

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

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

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

CS425 – Fall 2013 – Boris Glavic

5.4

©Silberschatz, Korth and Sudarshan



## Example Views

- A view of instructors without their salary  
**create view faculty as**  

```
select ID, name, dept_name
from instructor
```
- Find all instructors in the Biology department  

```
select name
from faculty
where dept_name = 'Biology'
```
- Create a view of department salary totals  
**create view departments\_total\_salary(dept\_name, total\_salary) as**  

```
select dept_name, sum (salary)
from instructor
group by dept_name;
```

CS425 – Fall 2013 – Boris Glavic

5.5

©Silberschatz, Korth and Sudarshan



## Views Defined Using Other Views

- **create view physics\_fall\_2009 as**  

```
select course.course_id, sec_id, building, room_number
from course, section
where course.course_id = section.course_id
and course.dept_name = 'Physics'
and section.semester = 'Fall'
and section.year = '2009';
```
- **create view physics\_fall\_2009\_watson as**  

```
select course_id, room_number
from physics_fall_2009
where building = 'Watson';
```

CS425 – Fall 2013 – Boris Glavic

5.6

©Silberschatz, Korth and Sudarshan



## View Expansion

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

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

CS425 – Fall 2016 – Boris Glavic

5.7

©Silberschatz, Korth and Sudarshan



## Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation  $v_1$  is said to *depend directly* on a view relation  $v_2$  if  $v_2$  is used in the expression defining  $v_1$
- A view relation  $v_1$  is said to *depend on* view relation  $v_2$  if either  $v_1$  depends directly to  $v_2$  or there is a path of dependencies from  $v_1$  to  $v_2$
- A view relation  $v$  is said to be *recursive* if it depends on itself.

CS425 – Fall 2016 – Boris Glavic

5.8

©Silberschatz, Korth and Sudarshan



## View Expansion

- A way to define the meaning of views defined in terms of other views.
- Let view  $v_1$  be defined by an expression  $e_1$  that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:
  - repeat**
  - Find any view relation  $v_i$  in  $e_1$
  - Replace the view relation  $v_i$  by the expression defining  $v_i$
  - until** no more view relations are present in  $e_1$
- As long as the view definitions are not recursive, this loop will terminate

CS425 – Fall 2016 – Boris Glavic

5.9

©Silberschatz, Korth and Sudarshan



## Update of a View

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

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

 This insertion must be represented by the insertion of the tuple ('30765', 'Green', 'Music', null) into the *instructor* relation

CS425 – Fall 2016 – Boris Glavic

5.10

©Silberschatz, Korth and Sudarshan



## Some Updates cannot be Translated Uniquely

- create view instructor\_info as**

```
select ID, name, building
from instructor, department
where instructor.dept_name = department.dept_name;
```
- insert into instructor\_info values** ('69987', 'White', 'Taylor');
  - which department, if multiple departments in Taylor?
  - what if no department is in Taylor?
- Most SQL implementations allow updates only on simple views
  - The **from** clause has only one database relation.
  - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
  - Any attribute not listed in the **select** clause can be set to null
  - The query does not have a **group** by or **having** clause.

CS425 – Fall 2016 – Boris Glavic

5.11

©Silberschatz, Korth and Sudarshan



## And Some Not at All

- create view history\_instructors as**

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

CS425 – Fall 2016 – Boris Glavic

5.12

©Silberschatz, Korth and Sudarshan



## Materialized Views

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



## Transactions

modified from:

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

©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## Transactions

- Unit of work
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions
- Transactions begin implicitly
  - Ended by **commit work** or **rollback work**
- But default on most databases: each SQL statement commits automatically
  - Can turn off auto commit for a session (e.g. using API)
  - In SQL:1999, can use: **begin atomic** .... **end**
    - Not supported on most databases



## Transactions Example

- Example Atomicity (all-or-nothing)
    - Recall example from the introduction
    - Relation **accounts(accID, cust, type, balance)**
    - A user want to transfer \$100 from his savings (accID = 100) to his checking account (accID= 101)
- ```

UPDATE accounts SET balance = balance - 100 WHERE accID = 100;
UPDATE accounts SET balance = balance + 100 WHERE accID = 101;

```
- This can cause inconsistencies if the system crashes after the first update (user would loose money)
  - Using a transaction either both or none of the statements are executed
- ```

BEGIN
UPDATE accounts SET balance = balance - 100 WHERE accID = 100;
UPDATE accounts SET balance = balance + 100 WHERE accID = 101;
COMMIT

```



## Transactions and Concurrency

- Transactions are also used to isolate concurrent actions of different users
- Recall from the introduction that if several users are modifying the database at the same time that can lead to inconsistencies
- More on that later once we talk about concurrency control



## Integrity Constraints

modified from:

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

©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than \$10,000.00
  - A salary of a bank employee must be at least \$4.00 an hour
  - A customer must have a (non-null) phone number

CS425 – Fall 2016 – Boris Glavic

5.19

©Silberschatz, Korth and Sudarshan



## Integrity Constraints on a Single Relation

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

CS425 – Fall 2016 – Boris Glavic

5.20

©Silberschatz, Korth and Sudarshan



## Not Null and Unique Constraints

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

```
name varchar(20) not null
budget numeric(12,2) not null
```
- **unique ( A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>m</sub> )**
  - The unique specification states that the attributes A<sub>1</sub>, A<sub>2</sub>, ... A<sub>m</sub> form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

CS425 – Fall 2016 – Boris Glavic

5.21

©Silberschatz, Korth and Sudarshan



## The check clause

- **check (P)**

where P is a predicate

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

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

CS425 – Fall 2016 – Boris Glavic

5.22

©Silberschatz, Korth and Sudarshan



## Referential Integrity

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

CS425 – Fall 2016 – Boris Glavic

5.23

©Silberschatz, Korth and Sudarshan



## Cascading Actions in Referential Integrity

- **create table course (**

```
  course_id char(5) primary key,
  title varchar(20),
  dept_name varchar(20) references department
)
```
- **create table course (**

```
  ...
  dept_name varchar(20),
  foreign key (dept_name) references department
  on delete cascade
  on update cascade,
  ...
)
```
- alternative actions to cascade: **set null, set default**

CS425 – Fall 2016 – Boris Glavic

5.24

©Silberschatz, Korth and Sudarshan



## Integrity Constraint Violation During Transactions

- E.g.
 

```
create table person (
  ID char(10),
  name char(40),
  mother char(10),
  father char(10),
  primary key ID,
  foreign key father references person,
  foreign key mother references person)
```
- How to insert a tuple without causing constraint violation ?
  - insert father and mother of a person before inserting person
  - OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
  - OR defer constraint checking (next slide)

CS425 – Fall 2016 – Boris Glavic

5.25

©Silberschatz, Korth and Sudarshan



## Complex Check Clauses

- **check** (*time\_slot\_id* in (select *time\_slot\_id* from *time\_slot*))
  - why not use a foreign key here?
- Every section has at least one instructor teaching the section.
  - how to write this?
- Unfortunately: subquery in check clause not supported by pretty much any database
  - Alternative: triggers (later)
- **create assertion** <assertion-name> **check** <predicate>;
  - Also not supported by anyone

CS425 – Fall 2016 – Boris Glavic

5.26

©Silberschatz, Korth and Sudarshan



## Indexes and User-Defined Types (UDTs)

modified from:

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

©Silberschatz, Korth and Sudarshan

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



## Built-in Data Types in SQL

- **date**: Dates, containing a (4 digit) year, month and date
  - Example: **date** '2005-7-27'
- **time**: Time of day, in hours, minutes and seconds.
  - Example: **time** '09:00:30'     **time** '09:00:30.75'
- **timestamp**: date plus time of day
  - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval**: period of time
  - Example: interval '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

CS425 – Fall 2016 – Boris Glavic

5.28

©Silberschatz, Korth and Sudarshan



## Index Creation

- **create table** *student* (*ID* **varchar** (5), *name* **varchar** (20) **not null**, *dept\_name* **varchar** (20), *tot\_cred* **numeric** (3,0) **default** 0, **primary key** (*ID*))
- **create index** *studentID\_index* on *student*(*ID*)
- Indices are data structures used to speed up access to records with specified values for index attributes
  - e.g. **select** \*  
      **from** *student*  
      **where** *ID* = '12345'

can be executed by using the index to find the required record, without looking at all records of *student*

More on indices later

CS425 – Fall 2016 – Boris Glavic

5.29

©Silberschatz, Korth and Sudarshan



## User-Defined Types

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

```
create type Dollars as numeric (12,2) final
```

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

CS425 – Fall 2016 – Boris Glavic

5.30

©Silberschatz, Korth and Sudarshan



## Domains

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

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

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



## Large-Object Types

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



## Access Control

modified from:

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

©Silberschatz, Korth and Sudarshan

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



## Access Control

Forms of authorization on parts of the database:

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

Forms of authorization to modify the database schema

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



## Authorization Specification in SQL

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

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



## Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
  - Example: grant users  $U_1$ ,  $U_2$ , and  $U_3$  **select** authorization on the *instructor* relation:
 

```
grant select on instructor to U1, U2, U3
```
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

## Revoking Authorization in SQL

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

```
revoke <privilege list>
on <relation name or view name> from <user list>
```
- Example:
 

```
revoke select on branch from U1, U2, U3
```
- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public**, all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

CS425 – Fall 2016 – Boris Glavic 5.37 ©Silberschatz, Korth and Sudarshan

## Roles

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

CS425 – Fall 2016 – Boris Glavic 5.38 ©Silberschatz, Korth and Sudarshan

## Authorization on Views

- create view geo\_instructor as**

```
(select *
from instructor
where dept_name = 'Geology');
```
- grant select on geo\_instructor to geo\_staff**
- Suppose that a *geo\_staff* member issues
  - select \***

```
from geo_instructor;
```
- What if
  - geo\_staff* does not have permissions on *instructor*?
  - creator of view did not have some permissions on *instructor*?

CS425 – Fall 2016 – Boris Glavic 5.39 ©Silberschatz, Korth and Sudarshan

## Other Authorization Features

- references** privilege to create foreign key
  - grant reference (dept\_name) on department to Mariano**;
  - why is this required?
- transfer of privileges
  - grant select on department to Amit with grant option**;
  - revoke select on department from Amit, Satoshi cascade**;
  - revoke select on department from Amit, Satoshi restrict**;
- Etc. read text book Section 4.6 for more details we have omitted here.

CS425 – Fall 2016 – Boris Glavic 5.40 ©Silberschatz, Korth and Sudarshan

## Understanding RESTRICT/CASCADE

- Bob grants right X on Y to Alice with grant option
- Alice grants right X on Y to Peter

```

graph TD
  Bob((Bob)) -- " " --> Alice((Alice))
  Alice -- " " --> Peter((Peter))
  style Bob stroke:#f00,stroke-width:2px
  style Alice stroke:#f00,stroke-width:2px
  style Peter stroke:#f00,stroke-width:2px
  linkStyle 0 stroke:#f00,stroke-width:2px
  
```

- Abandoned right**
  - A right for which there is no justification anymore
- revoke X on Y from Alice restrict**
  - With restrict fails if it would result in abandoned rights
- revoke X on Y from Alice cascade**
  - Also revokes rights that would otherwise be abandoned

CS425 – Fall 2016 – Boris Glavic 5.41 ©Silberschatz, Korth and Sudarshan

## Understanding RESTRICT/CASCADE

- Bob grants right X on Y to Alice with grant option
- Alice grants right X on Y to Peter
- Bob grants right X on Y to Peter

```

graph TD
  Bob((Bob)) -- " " --> Alice((Alice))
  Alice -- " " --> Peter((Peter))
  Bob -- " " --> Peter
  style Bob stroke:#f00,stroke-width:2px
  style Alice stroke:#f00,stroke-width:2px
  style Peter stroke:#f00,stroke-width:2px
  linkStyle 0 stroke:#f00,stroke-width:2px
  
```

- Abandoned privilege**
  - A privilege for which there is no justification anymore
  - Indirect justifications count
- revoke X on Y from Alice restrict**
  - Fails: even though there exists additional justification for the privilege.
- revoke X on Y from Alice cascade**
  - Revokes that right from Peter.
  - Peter still has the right to do X on Y

CS425 – Fall 2016 – Boris Glavic 5.42 ©Silberschatz, Korth and Sudarshan





## Recap

- Views
  - Virtual
  - Materialized
  - Updates
- Integrity Constraints
  - Not null, unique, check
  - Foreign keys: referential integrity
- Access control
  - Users, roles
  - Privileges
  - **GRANT / REVOKE**
- Data types
  - Built-in types, Domains, Large Objects
  - UDTs
  - Indices



## End of Chapter 5

modified from:  
 Database System Concepts, 6<sup>th</sup> Ed.  
 ©Silberschatz, Korth and Sudarshan  
 See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## Outline

- Introduction
- Relational Data Model
- Formal Relational Languages (relational algebra)
- **SQL - Advanced**
- Database Design
- Transaction Processing, Recovery, and Concurrency Control
- Storage and File Structures
- Indexing and Hashing
- Query Processing and Optimization



## Figure 4.01

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



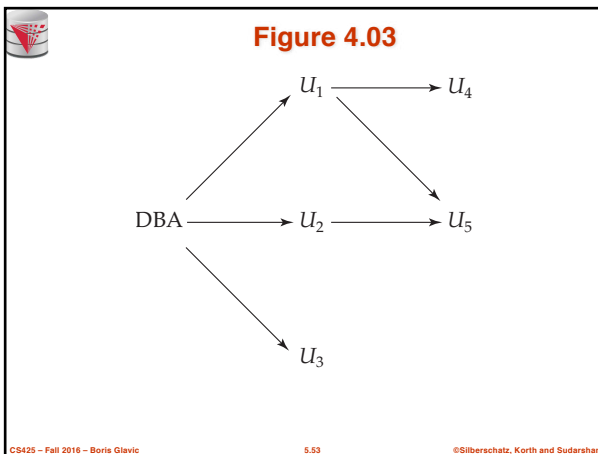
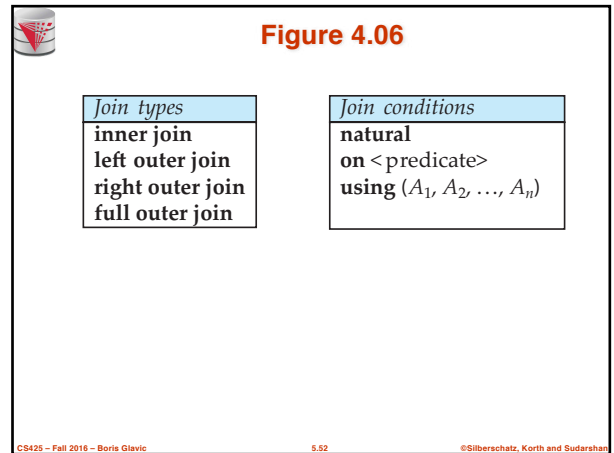
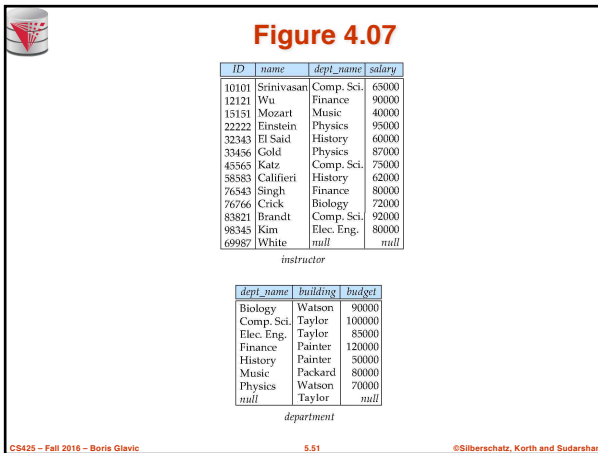
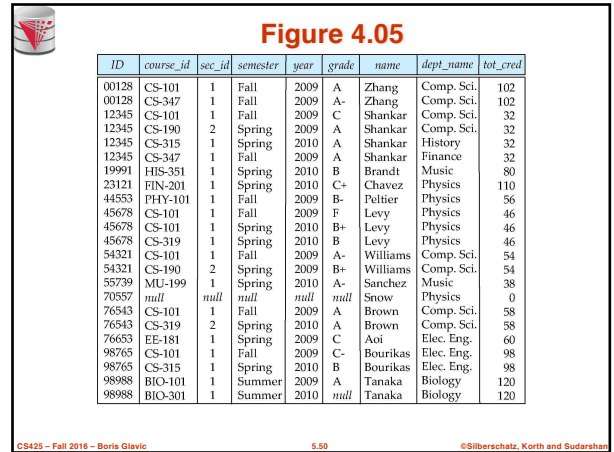
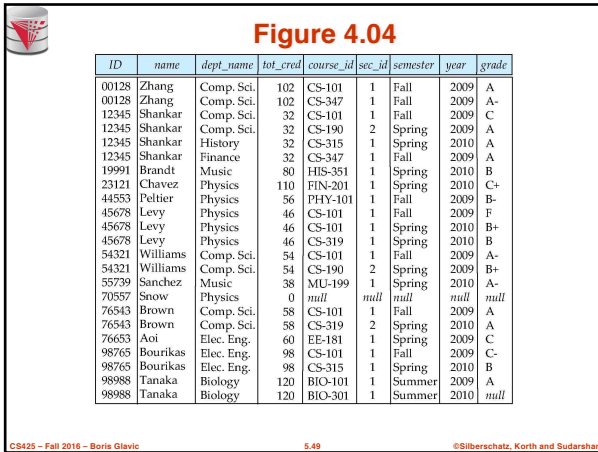
## Figure 4.02

ID	course_id	sec_id	semester	year	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



## Figure 4.03

ID	name	dept_name	tot_cred	course_id	sec_id	semester	year	grade
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2009	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2009	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2009	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2009	A
12345	Shankar	History	32	CS-315	1	Spring	2010	A
12345	Shankar	Finance	32	CS-347	1	Fall	2009	A
19991	Brandt	Music	80	HIS-351	1	Spring	2010	B
23121	Chavez	Physics	110	FIN-201	1	Spring	2010	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2009	B-
45678	Levy	Physics	46	CS-101	1	Fall	2009	F
45678	Levy	Physics	46	CS-101	1	Spring	2010	B+
45678	Levy	Physics	46	CS-319	1	Spring	2010	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2009	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2009	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2010	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2009	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2010	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2009	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2009	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2010	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2009	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2010	null





**CS425 – Fall 2013**  
**Boris Glavic**  
**Chapter 6: Advanced SQL**

modified from:  
 Database System Concepts, 6<sup>th</sup> Ed.  
 ©Silberschatz, Korth and Sudarshan  
 See [www.db-book.com](http://www.db-book.com) for conditions on reuse



**Chapter 6: Advanced SQL**

- Accessing SQL From a Programming Language
  - Dynamic SQL
    - ▶ JDBC and ODBC
  - Embedded SQL
- Functions and Procedural Constructs
- Triggers



**Textbook: Chapter 5**



**Accessing SQL From a Programming Language**



**JDBC and ODBC**

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
  - Connect with the database server
  - Send SQL commands to the database server
  - Fetch tuples of result one-by-one into program variables
- ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
  - Other API's such as ADO.NET sit on top of ODBC
- JDBC (Java Database Connectivity) works with Java




**Native APIs**

- Most DBMS also define DBMS specific APIs
- Oracle: OCI
- Postgres: libpq
- ...



**JDBC**

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




## JDBC Code

```

public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName("oracle.jdbc.driver.OracleDriver"); // load driver
        Connection conn = DriverManager.getConnection( // connect to server
            "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement(); // create Statement object
        ... Do Actual Work ...
        stmt.close(); // close Statement and release resources
        conn.close(); // close Connection and release resources
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle); // handle exceptions
    }
}

```

CS425 – Fall 2016 – Boris Glavic 5.7 ©Silberschatz, Korth and Sudarshan



## JDBC Code (Cont.)

- Update to database
 

```

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


```
- Execute query and fetch and print results
 

```

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

```


CS425 – Fall 2016 – Boris Glavic 5.8 ©Silberschatz, Korth and Sudarshan



## JDBC Code Details

- Result stores the current row position in the result
  - Pointing before the first row after executing the statement
  - `.next()` moves to the next tuple
    - Returns false if no more tuples
- Getting result fields:
  - `rs.getString("dept_name")` and `rs.getString(1)` equivalent if `dept_name` is the first attribute in select result.
- Dealing with Null values
  - `int a = rs.getInt("a");`  
if (`rs.isNull()`) `Systems.out.println("Got null value");`

CS425 – Fall 2016 – Boris Glavic 5.9 ©Silberschatz, Korth and Sudarshan



## Prepared Statement


- `PreparedStatement pStmt = conn.prepareStatement("insert into instructor values(?,?,?)");`

```

pStmt.setString(1, "88877"); pStmt.setString(2, "Perry");
pStmt.setString(3, "Finance"); pStmt.setInt(4, 125000);
pStmt.executeUpdate();
pStmt.setString(1, "88878");
pStmt.executeUpdate();

```
- For queries, use `pStmt.executeQuery()`, which returns a `ResultSet`
- **WARNING:** always use prepared statements when taking an input from the user and adding it to a query
  - **NEVER create a query by concatenating strings which you get as inputs**
  - "insert into instructor values(' " + ID + "', ' " + name + "', ' " + dept name + "', ' " + balance + "');"
    - What if name is "D' Souza"?

CS425 – Fall 2016 – Boris Glavic 5.10 ©Silberschatz, Korth and Sudarshan



## SQL Injection

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


```

"select * from instructor where name = 'X\'' or \'Y\' = \'Y\'

```

  - **Always use prepared statements, with user inputs as parameters**

CS425 – Fall 2016 – Boris Glavic 5.11 ©Silberschatz, Korth and Sudarshan



## Metadata Features

- `ResultSet` metadata
- E.g., after executing query to get a `ResultSet` `rs`:
 

```

ResultSetMetaData rsmd = rs.getMetaData();
for(int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i));
    System.out.println(rsmd.getColumnTypeName(i));
}

```
- How is this useful?

CS425 – Fall 2016 – Boris Glavic 5.12 ©Silberschatz, Korth and Sudarshan



## Metadata (Cont)

- Database metadata
- DatabaseMetaData dbmd = conn.getMetaData();  

```
ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
// Arguments to getColumns: Catalog, Schema-pattern, Table-pattern,
// and Column-Pattern
// Returns: One row for each column; row has a number of attributes
// such as COLUMN_NAME, TYPE_NAME
while( rs.next() ) {
    System.out.println(rs.getString("COLUMN_NAME"),
                       rs.getString("TYPE_NAME"));
}

```
- And where is this useful?

CS425 - Fall 2016 - Boris Glavic

5.13

©Silberschatz, Korth and Sudarshan



## Transaction Control in JDBC

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

CS425 - Fall 2016 - Boris Glavic

5.14

©Silberschatz, Korth and Sudarshan



## Other JDBC Features

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

CS425 - Fall 2016 - Boris Glavic

5.15

©Silberschatz, Korth and Sudarshan



## SQLJ

- JDBC is dynamic, errors cannot be caught by compiler
- SQLJ: embedded SQL in Java
  - #sql iterator deptInfolter ( String dept name, int avgSal);  
deptInfolter iter = null;  
#sql iter = { select dept\_name, avg(salary) from instructor  
group by dept name };  
while (iter.next()) {  
String deptName = iter.dept\_name();  
int avgSal = iter.avgSal();  
System.out.println(deptName + " " + avgSal);  
}  
iter.close();

CS425 - Fall 2016 - Boris Glavic

5.16

©Silberschatz, Korth and Sudarshan



## ODBC

- Open DataBase Connectivity(ODBC) standard
  - standard for application program to communicate with a database server.
  - application program interface (API) to
    - ▶ open a connection with a database,
    - ▶ send queries and updates,
    - ▶ get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC
- Was defined originally for Basic and C, versions available for many languages.

CS425 - Fall 2016 - Boris Glavic

5.17

©Silberschatz, Korth and Sudarshan



## ODBC (Cont.)

- Each database system supporting ODBC provides a "driver" library that must be linked with the client program.
- When client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results.
- ODBC program first allocates an SQL environment, then a database connection handle.
- Opens database connection using SQLConnect(). Parameters for SQLConnect:
  - connection handle,
  - the server to which to connect
  - the user identifier,
  - password
- Must also specify types of arguments:
  - SQL\_NTS denotes previous argument is a null-terminated string.

CS425 - Fall 2016 - Boris Glavic

5.18

©Silberschatz, Korth and Sudarshan

## ODBC Code

```

int ODBCexample()
{
    RETCODE error;
    HENV env; /* environment */
    HDBC conn; /* database connection */
    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, "db.yale.edu", SQL_NTS, "avi", SQL_NTS,
               "avipasswd", SQL_NTS);
    { .... Do actual work ... }

    SQLDisconnect(conn);
    SQLFreeConnect(conn);
    SQLFreeEnv(env);
}

```

CS425 - Fall 2016 - Boris Glavic 5.19 ©Silberschatz, Korth and Sudarshan

## ODBC Code (Cont.)

- Program sends SQL commands to database by using SQLExecDirect
- Result tuples are fetched using SQLFetch()
- SQLBindCol() binds C language variables to attributes of the query result
  - When a tuple is fetched, its attribute values are automatically stored in corresponding C variables.
  - Arguments to SQLBindCol()
    - ▶ ODBC stmt variable, attribute position in query result
    - ▶ The type conversion from SQL to C.
    - ▶ The address of the variable.
    - ▶ For variable-length types like character arrays,
      - The maximum length of the variable
      - Location to store actual length when a tuple is fetched.
      - Note: A negative value returned for the length field indicates null value
- Good programming requires checking results of every function call for errors; we have omitted most checks for brevity.

CS425 - Fall 2016 - Boris Glavic 5.20 ©Silberschatz, Korth and Sudarshan

## ODBC Code (Cont.)

```

Main body of program
char deptname[80];
float salary;
int lenOut1, lenOut2;
HSTMT stmt;
char * sqlquery = "select dept_name, sum (salary)
                  from instructor
                  group by dept_name";
SQLAllocStmt(conn, &stmt);
error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
if (error == SQL_SUCCESS) {
    SQLBindCol(stmt, 1, SQL_C_CHAR, deptname, 80, &lenOut1);
    SQLBindCol(stmt, 2, SQL_C_FLOAT, &salary, 0, &lenOut2);
    while (SQLFetch(stmt) == SQL_SUCCESS) {
        printf (" %s %g\n", deptname, salary);
    }
}
SQLFreeStmt(stmt, SQL_DROP);

```

CS425 - Fall 2016 - Boris Glavic 5.21 ©Silberschatz, Korth and Sudarshan

## ODBC Prepared Statements

- **Prepared Statement**
  - SQL statement prepared: compiled at the database
  - Can have placeholders: E.g. insert into account values(?, ?, ?)
  - Repeatedly executed with actual values for the placeholders
- To prepare a statement  
SQLPrepare(stmt, <SQL String>);
- To bind parameters  
SQLBindParameter(stmt, <parameter#>, ... type information and value omitted for simplicity..)
- To execute the statement  
retcode = SQLExecute( stmt);
- To avoid SQL injection security risk, do not create SQL strings directly using user input; instead use prepared statements to bind user inputs

CS425 - Fall 2016 - Boris Glavic 5.22 ©Silberschatz, Korth and Sudarshan

## More ODBC Features

- **Metadata features**
  - finding all the relations in the database and
  - finding the names and types of columns of a query result or a relation in the database.
- By default, each SQL statement is treated as a separate transaction that is committed automatically.
  - Can turn off automatic commit on a connection
    - ▶ SQLSetConnectOption(conn, SQL\_AUTOCOMMIT, 0);
  - Transactions must then be committed or rolled back explicitly by
    - ▶ SQLTransact(conn, SQL\_COMMIT) or
    - ▶ SQLTransact(conn, SQL\_ROLLBACK)

CS425 - Fall 2016 - Boris Glavic 5.23 ©Silberschatz, Korth and Sudarshan

## ODBC Conformance Levels

- Conformance levels specify subsets of the functionality defined by the standard.
  - Core
  - Level 1 requires support for metadata querying
  - Level 2 requires ability to send and retrieve arrays of parameter values and more detailed catalog information.
- SQL Call Level Interface (CLI) standard similar to ODBC interface, but with some minor differences.

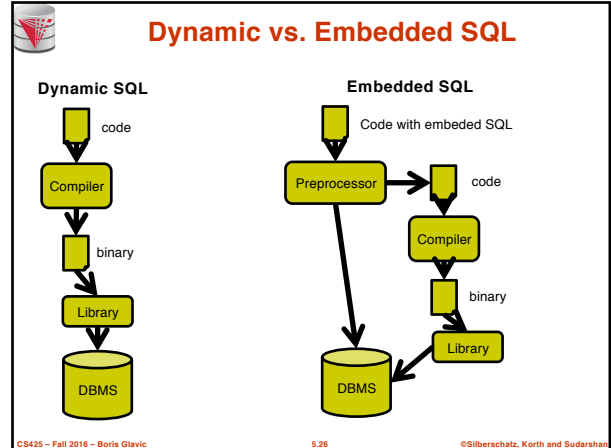
CS425 - Fall 2016 - Boris Glavic 5.24 ©Silberschatz, Korth and Sudarshan

## ADO.NET

- API designed for Visual Basic .NET and C#, providing database access facilities similar to JDBC/ODBC
- Partial example of ADO.NET code in C# using System, System.Data, System.Data.SqlClient;
 

```
SqlConnection conn = new SqlConnection(
    "Data Source=<IPAddr>, Initial Catalog=<Catalog>");
conn.Open();
SqlCommand cmd = new SqlCommand("select * from students",
    conn);
SqlDataReader rdr = cmd.ExecuteReader();
while(rdr.Read()) {
    Console.WriteLine(rdr[0], rdr[1]); /* Prints result attributes 1 & 2 */
}
rdr.Close(); conn.Close();
```
- Can also access non-relational data sources such as
  - OLE-DB, XML data, Entity framework

CS425 - Fall 2016 - Boris Glavic 5.25 ©Silberschatz, Korth and Sudarshan



## Embedded SQL

- The SQL standard defines embeddings 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 comprise *embedded SQL*.
- The basic form of these languages follows that of the System R embedding of SQL into PL/I.
- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor
 

```
EXEC SQL <embedded SQL statement > END_EXEC
```

Note: this varies by language (for example, the Java embedding uses # SQL { ... };)

CS425 - Fall 2016 - Boris Glavic 5.27 ©Silberschatz, Korth and Sudarshan

## Example Query

- From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable *credit\_amount*.
- Specify the query in SQL and declare a *cursor* for it
 

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

CS425 - Fall 2016 - Boris Glavic 5.28 ©Silberschatz, Korth and Sudarshan

## Embedded SQL (Cont.)

- The **open** statement causes the query to be evaluated
 

```
EXEC SQL open c END_EXEC
```
- The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.
 

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

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

```
EXEC SQL close c END_EXEC
```

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

CS425 - Fall 2016 - Boris Glavic 5.29 ©Silberschatz, Korth and Sudarshan


## Updates Through Cursors

- Can update tuples fetched by cursor by declaring that the cursor is for update
 

```
declare c cursor for
select *
from instructor
where dept_name = 'Music'
for update
```
- To update tuple at the current location of cursor *c*


```
update instructor
set salary = salary + 100
where current of c
```

CS425 - Fall 2016 - Boris Glavic 5.30 ©Silberschatz, Korth and Sudarshan



## Procedural Constructs in SQL


CS425 – Fall 2016 – Boris Glavic 5.31 ©Silberschatz, Korth and Sudarshan



## Procedural Extensions and Stored Procedures

- SQL provides a **module** language
  - Permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.
- Stored Procedures
  - Can store procedures in the database
  - then execute them using the **call** statement
  - permit external applications to operate on the database without knowing about internal details
- Object-oriented aspects of these features are covered in Chapter 22 (Object Based Databases) in the textbook


CS425 – Fall 2016 – Boris Glavic 5.32 ©Silberschatz, Korth and Sudarshan



## Why have procedural extensions?

- Shipping data between a database server and application program (e.g., through network connection) is costly
- Converting data from the database internal format into a format understood by the application programming language is costly
- Example:
  - Use Java to retrieve all users and their friend-relationships from a friends relation representing a world-wide social network with 10,000,000 users
  - Compute the transitive closure
    - All pairs of users connects through a path of friend relationships. E.g., (Peter, Magret) if Peter is a friend of Walter who is a friend of Magret
  - Return pairs of users from Chicago – say 4000 pairs
  - 1) cannot be expressed (efficiently) as SQL query, 2) result is small
    - -> save by executing this on the DB server


CS425 – Fall 2016 – Boris Glavic 5.33 ©Silberschatz, Korth and Sudarshan



## Functions and Procedures

- SQL:1999 supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external programming language.
  - Functions are particularly useful with specialized data types such as images and geometric objects.
    - Example: functions to check if polygons overlap, or to compare images for similarity.
  - Some database systems support **table-valued functions**, which can return a relation as a result.
- SQL:1999 also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999.

CS425 – Fall 2016 – Boris Glavic 5.34 ©Silberschatz, Korth and Sudarshan




## SQL Functions

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

```
create function dept_count (dept_name varchar(20))
returns integer
begin
  declare d_count integer;
  select count (*) into d_count
  from instructor
  where instructor.dept_name = dept_name;
  return d_count;
end
```
- Find the department name and budget of all departments with more that 12 instructors.
 

```
select dept_name, budget
from department
where dept_count (dept_name) > 1
```

CS425 – Fall 2016 – Boris Glavic 5.35 ©Silberschatz, Korth and Sudarshan



## Table Functions

- SQL:2003 added functions that return a relation as a result
- Example: Return all accounts owned by a given customer
 

```
create function instructors_of (dept_name char(20))
returns table ( ID varchar(5),
              name varchar(20),
              dept_name varchar(20),
              salary numeric(8,2))

return table
(select ID, name, dept_name, salary
from instructor
where instructor.dept_name = instructors_of.dept_name)
```
- Usage
 

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

CS425 – Fall 2016 – Boris Glavic 5.36 ©Silberschatz, Korth and Sudarshan





## SQL Procedures

- The `dept_count` function could instead be written as procedure:
 

```
create procedure dept_count_proc (in dept_name varchar(20),
                                out d_count integer)
begin
  select count(*) into d_count
  from instructor
  where instructor.dept_name = dept_count_proc.dept_name
end
```
- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the `call` statement.
 

```
declare d_count integer;
call dept_count_proc( 'Physics', d_count);
```
- Procedures and functions can be invoked also from dynamic SQL
- SQL:1999 allows more than one function/procedure of the same name (called name **overloading**), as long as the number of arguments differ, or at least the types of the arguments differ

CS425 – Fall 2016 – Boris Glavic

5.37

©Silberschatz, Korth and Sudarshan



## Procedural Constructs

- Warning: most database systems implement their own variant of the standard syntax below
  - read your system manual to see what works on your system
- Compound statement: **begin ... end**,
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within a compound statements
- **While** and **repeat** statements :
 

```
declare n integer default 0;
while n < 10 do
  set n = n + 1
end while

repeat
  set n = n - 1
until n = 0
end repeat
```

CS425 – Fall 2016 – Boris Glavic

5.38

©Silberschatz, Korth and Sudarshan



## Procedural Constructs (Cont.)

- **For** loop
  - Permits iteration over all results of a query
  - Example:
 

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

CS425 – Fall 2016 – Boris Glavic

5.39

©Silberschatz, Korth and Sudarshan



## Procedural Constructs (cont.)

- Conditional statements (**if-then-else**)
 

SQL:1999 also supports a **case** statement similar to C case statement
- Example procedure: registers student after ensuring classroom capacity is not exceeded
  - Returns 0 on success and -1 if capacity is exceeded
  - See book for details
- Signaling of exception conditions, and declaring handlers for exceptions
 

```
declare out_of_classroom_seats condition
declare exit handler for out_of_classroom_seats
begin
  ...
  ... signal out_of_classroom_seats
end
```

  - The handler here is **exit** -- causes enclosing **begin..end** to be exited
  - Other actions possible on exception

CS425 – Fall 2016 – Boris Glavic

5.40

©Silberschatz, Korth and Sudarshan



## External Language Functions/Procedures

- SQL:1999 permits the use of functions and procedures written in other languages such as C or C++
- Declaring external language procedures and functions
 

```
create procedure dept_count_proc(in dept_name varchar(20),
                                out count integer)
language C
external name ' /usr/avi/bin/dept_count_proc'

create function dept_count(dept_name varchar(20))
returns integer
language C
external name ' /usr/avi/bin/dept_count'
```

CS425 – Fall 2016 – Boris Glavic

5.41

©Silberschatz, Korth and Sudarshan



## External Language Routines (Cont.)

- Benefits of external language functions/procedures:
  - more efficient for many operations, and more expressive power.
- Drawbacks
  - Code to implement function may need to be loaded into database system and executed in the database system's address space.
    - ▶ risk of accidental corruption of database structures
    - ▶ security risk, allowing users access to unauthorized data
  - There are alternatives, which give good security at the cost of potentially worse performance.
  - Direct execution in the database system's space is used when efficiency is more important than security.

CS425 – Fall 2016 – Boris Glavic

5.42

©Silberschatz, Korth and Sudarshan



## Security with External Language Routines

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

CS425 - Fall 2016 - Boris Glavic

5.43

©Silberschatz, Korth and Sudarshan



## Triggers

CS425 - Fall 2016 - Boris Glavic

5.44

©Silberschatz, Korth and Sudarshan



## Triggers

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

CS425 - Fall 2016 - Boris Glavic

5.45

©Silberschatz, Korth and Sudarshan



## Trigger Example

- E.g. *time\_slot\_id* is not a primary key of *timeslot*, so we cannot create a foreign key constraint from *section* to *timeslot*.
- Alternative: use triggers on *section* and *timeslot* to enforce integrity constraints
 

```
create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
  select time_slot_id
  from time_slot)) /* time_slot_id not present in time_slot */
begin
  rollback
end;
```

CS425 - Fall 2016 - Boris Glavic

5.46

©Silberschatz, Korth and Sudarshan



## Trigger Example Cont.

```
create trigger timeslot_check2 after delete on timeslot
referencing old row as orow
for each row
when (orow.time_slot_id not in (
  select time_slot_id
  from time_slot)
/* last tuple for time slot id deleted from time slot */
and orow.time_slot_id in (
  select time_slot_id
  from section)) /* and time_slot_id still referenced from section */
begin
  rollback
end;
```

CS425 - Fall 2016 - Boris Glavic

5.47

©Silberschatz, Korth and Sudarshan



## Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - E.g., **after update of takes on grade**
- Values of attributes before and after an update can be referenced
  - **referencing old row as** : for deletes and updates
  - **referencing new row as** : for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. E.g. convert blank grades to null.
 

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
when (nrow.grade = ' ')
begin atomic
  set nrow.grade = null;
end;
```

CS425 - Fall 2016 - Boris Glavic

5.48

©Silberschatz, Korth and Sudarshan



## Trigger to Maintain credits\_earned value

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

CS425 – Fall 2016 – Boris Glavic

5.49

©Silberschatz, Korth and Sudarshan



## Statement Level Triggers

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

CS425 – Fall 2016 – Boris Glavic

5.50

©Silberschatz, Korth and Sudarshan



## When Not To Use Triggers

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

CS425 – Fall 2016 – Boris Glavic

5.51

©Silberschatz, Korth and Sudarshan



## When Not To Use Triggers

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

CS425 – Fall 2016 – Boris Glavic

5.52

©Silberschatz, Korth and Sudarshan



## Recursive Queries

CS425 – Fall 2016 – Boris Glavic

5.53

©Silberschatz, Korth and Sudarshan



## Recursion in SQL

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

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

from *rec\_prereq*;

This example view, *rec\_prereq*, is called the *transitive closure* of the *prereq* relation

CS425 – Fall 2016 – Boris Glavic

5.54

©Silberschatz, Korth and Sudarshan



## The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *prereq* with itself
    - This can give only a fixed number of levels of managers
    - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
    - Alternative: write a procedure to iterate as many times as required
      - See procedure *findAllPrereqs* in book

CS425 – Fall 2016 – Boris Glavic

5.55

©Silberschatz, Korth and Sudarshan



## The Power of Recursion

- Computing transitive closure using iteration, adding successive tuples to *rec\_prereq*
  - The next slide shows a *prereq* relation
  - Each step of the iterative process constructs an extended version of *rec\_prereq* from its recursive definition.
  - The final result is called the *fixed point* of the recursive view definition.
- Recursive views are required to be **monotonic**. That is,
  - if we add tuples to *prereq* the view *rec\_prereq* contains all of the tuples it contained before, plus possibly more

CS425 – Fall 2016 – Boris Glavic

5.56

©Silberschatz, Korth and Sudarshan



## Example of Fixed-Point Computation

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

Iteration Number	Tuples in cl
0	
1	(CS-301)
2	(CS-301), (CS-201)
3	(CS-301), (CS-201)
4	(CS-301), (CS-201), (CS-101)
5	(CS-301), (CS-201), (CS-101)

CS425 – Fall 2016 – Boris Glavic

5.57

©Silberschatz, Korth and Sudarshan



## Another Recursion Example

- Given relation *manager(employee\_name, manager\_name)*
- Find all employee-manager pairs, where the employee reports to the manager directly or indirectly (that is manager's manager, manager's manager's manager, etc.)
 

```
with recursive empl (employee_name, manager_name) as (
  select employee_name, manager_name
  from manager
  union
  select manager.employee_name, empl.manager_name
  from manager, empl
  where manager.manager_name = empl.employee_name)
select *
from empl
```

This example view, *empl*, is the *transitive closure* of the *manager* relation

CS425 – Fall 2016 – Boris Glavic

5.58

©Silberschatz, Korth and Sudarshan



## Recap

- Programming Language Interfaces for Databases
  - Dynamic SQL (e.g., JDBC, ODBC)
  - Embedded SQL
  - SQL Injection
- Procedural Extensions of SQL
  - Functions and Procedures
- External Functions/Procedures
  - Written in programming language (e.g., C)
- Triggers
  - Events (insert, ...)
  - Conditions (WHEN)
  - per statement / per row
  - Accessing old/new table/row versions
- Recursive Queries

CS425 – Fall 2016 – Boris Glavic

5.59

©Silberschatz, Korth and Sudarshan



## End of Chapter

modified from:  
 Database System Concepts, 6<sup>th</sup> Ed.  
 ©Silberschatz, Korth and Sudarshan  
 See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## Outline

- Introduction
- Relational Data Model
- Formal Relational Languages (relational algebra)
- SQL - Advanced
- **Database Design – ER model**
- Transaction Processing, Recovery, and Concurrency Control
- Storage and File Structures
- Indexing and Hashing
- Query Processing and Optimization



## CS425 – Fall 2013 Boris Glavic Chapter 7: Entity-Relationship Model

Partially taken from  
Klaus R. Dittrich

modified from:  
Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on reuse



## Chapter 7: Entity-Relationship Model

- Design Process
- Modeling
- Constraints
- E-R Diagram
- Design Issues
- Weak Entity Sets
- Extended E-R Features
- Design of the Bank Database
- Reduction to Relation Schemas
- Database Design
- UML

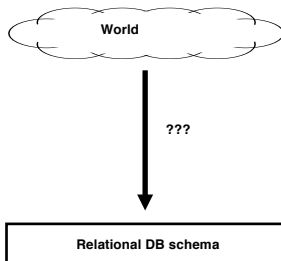
CS425 – Fall 2016 – Boris Glavic

7.2

©Silberschatz, Korth and Sudarshan



## Database Design



CS425 – Fall 2016 – Boris Glavic

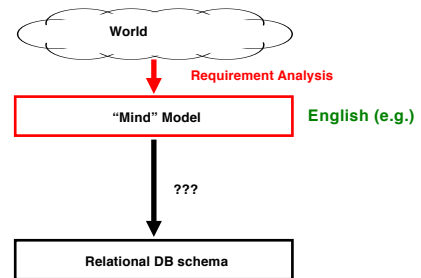
7.3

©Silberschatz, Korth and Sudarshan



## Database Design

- First: need to develop a "mind"-model based on a requirement analysis



CS425 – Fall 2016 – Boris Glavic

7.4

©Silberschatz, Korth and Sudarshan



## Requirement Analysis Example Zoo

- The zoo stores information about animals, cages, and zoo keepers.
- Animals are of a certain species and have a name. For each animal we want to record its weight and age.
- Each cage is located in a section of the zoo. Cages can house animals, but there may be cages that are currently empty. Cages have a size in square meter.
- Zoo keepers are identified by their social security number. We store a first name, last name, and for each zoo keeper. Zoo keepers are assigned to cages they have to take care of (clean, ...). Each cage that is not empty has a zoo keeper assigned to it. A zoo keeper can take care of several cages. Each zoo keeper takes care of at least one cage.

CS425 – Fall 2016 – Boris Glavic

7.5

©Silberschatz, Korth and Sudarshan



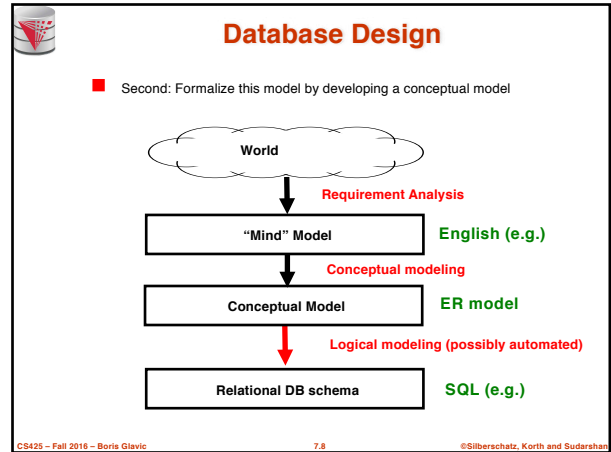
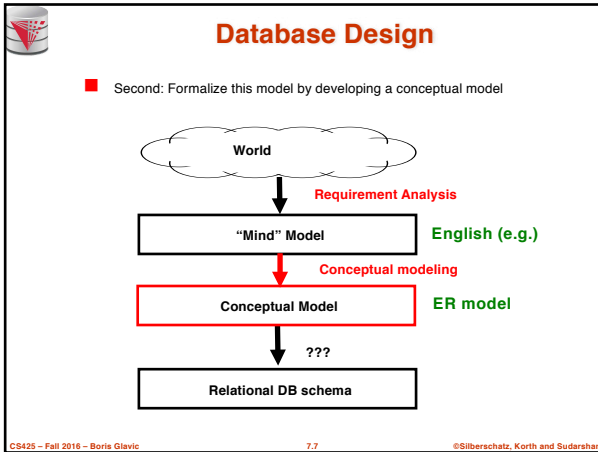
## Requirement Analysis Example Music Collection

- Let's do it!

CS425 – Fall 2016 – Boris Glavic

7.6

©Silberschatz, Korth and Sudarshan



- ## Modeling – ER model
- A *database* can be modeled as:
    - a collection of entities,
    - relationship among entities.
  - An **entity** is an object that exists and is distinguishable from other objects.
    - Example: specific person, company, event, plant
  - Entities have **attributes**
    - Example: people have *names* and *addresses*
  - An **entity set** is a set of entities of the same type that share the same properties.
    - Example: set of all persons, companies, trees, holidays
- CS425 – Fall 2016 – Boris Glavic 7.9 ©Silberschatz, Korth and Sudarshan

## Entity Sets *instructor* and *student*

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

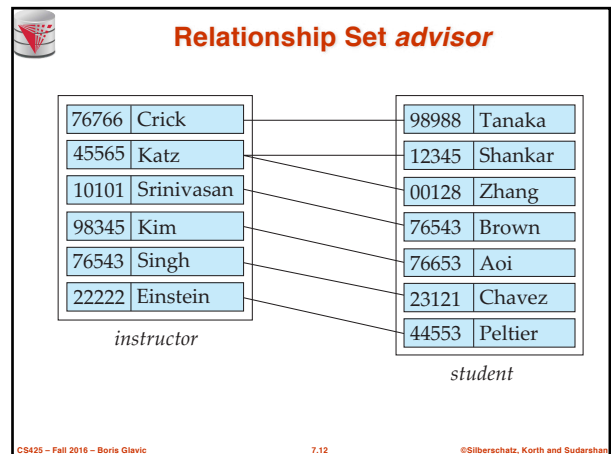
*instructor*

student-ID	student_name
98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

*student*

CS425 – Fall 2016 – Boris Glavic 7.10 ©Silberschatz, Korth and Sudarshan

- ## Relationship Sets
- A **relationship** is an association among several entities
- Example:
- |                 |                  |                   |
|-----------------|------------------|-------------------|
| 44553 (Peltier) | <i>advisor</i>   | 22222 (Einstein)  |
| student entity  | relationship set | instructor entity |
- A **relationship set** is a mathematical relation among  $n \geq 2$  entities, each taken from entity sets
 
$$\{(e_1, e_2, \dots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$
 where  $(e_1, e_2, \dots, e_n)$  is a relationship
    - Example:
 
$$(44553, 22222) \in \textit{advisor}$$
- CS425 – Fall 2016 – Boris Glavic 7.11 ©Silberschatz, Korth and Sudarshan







### Mapping Cardinalities Example

**Person**    **Birth certificate**

A            B

(a)

One to one

**Advisor**    **Student**

A            B

(b)

One to many

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

CS425 – Fall 2016 – Boris Glavic    7.19    ©Silberschatz, Korth and Sudarshan

### Mapping Cardinalities

A            B

(a)

Many to one

A            B

(b)

Many to many

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

CS425 – Fall 2016 – Boris Glavic    7.20    ©Silberschatz, Korth and Sudarshan

### Mapping Cardinalities Example

**Employee**    **Department**

A            B

(a)

Many to one

**Student**    **Course**

A            B

(b)

Many to many

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

CS425 – Fall 2016 – Boris Glavic    7.21    ©Silberschatz, Korth and Sudarshan

### Mapping Cardinality Constraints Cont.

- What if we allow some elements to not be mapped to another element?
  - E.g., 0:1 – 1
- For a binary relationship set the mapping cardinality must be one of the following types:
 

<ul style="list-style-type: none"> <li>■ <b>1-1</b></li> <li>● 0:1-1</li> <li>● 0:1-1</li> <li>● 1-0:1</li> <li>● 0:1-0:1</li> </ul>	<ul style="list-style-type: none"> <li>■ <b>N-1</b></li> <li>● N-1</li> <li>● N-0:1</li> <li>● 0:N-1</li> <li>● 0:N-0:1</li> </ul>
<ul style="list-style-type: none"> <li>■ <b>1-N</b></li> <li>● 0:1-N</li> <li>● 0:1-0:N</li> <li>● 1-N</li> <li>● 1-0:N</li> </ul>	<ul style="list-style-type: none"> <li>■ <b>N-M</b></li> <li>● N-M</li> <li>● N-0:M</li> <li>● 0:N-M</li> <li>● 0:N-0:M</li> </ul>

CS425 – Fall 2016 – Boris Glavic    7.22    ©Silberschatz, Korth and Sudarshan

### Mapping Cardinality Constraints Cont.

- Typical Notation
  - (0:1) – (1:N)

CS425 – Fall 2016 – Boris Glavic    7.23    ©Silberschatz, Korth and Sudarshan

### Keys

- A **super key** of an entity set is a set of one or more attributes whose values uniquely determine each entity.
- A **candidate key** of an entity set is a minimal super key
  - *ID* is candidate key of *instructor*
  - *course\_id* is candidate key of *course*
- Although several candidate keys may exist, one of the candidate keys is selected to be the **primary key**.
- **Note: Basically the same as for relational model**

CS425 – Fall 2016 – Boris Glavic    7.24    ©Silberschatz, Korth and Sudarshan

## Keys for Relationship Sets

- The combination of primary keys of the participating entity sets forms a super key of a relationship set.
  - $(s\_id, i\_id)$  is the super key of *advisor*
  - **NOTE:** this means **a pair of entities can have at most one relationship in a particular relationship set.**
    - Example: if we wish to track multiple meeting dates between a student and her advisor, we cannot assume a relationship for each meeting. We can use a multivalued attribute though or model meeting as a separate entity
- Must consider the mapping cardinality of the relationship set when deciding what are the candidate keys
- Need to consider semantics of relationship set in selecting the *primary key* in case of more than one candidate key

CS425 – Fall 2016 – Boris Glavic 7.25 ©Silberschatz, Korth and Sudarshan

## Keys for Relationship Sets Cont.

- Must consider the mapping cardinality of the relationship set when deciding what are the candidate keys
  - 1-1: both primary keys are candidate keys
    - Example: **hasBc:** (Person-Birthcertificate)
  - N-1: the N side is the candidate key
    - Example: **worksFor:** (Instructor-Department)
  - N-M: the combination of both primary keys
    - Example: **takes:** (Student-Course)

CS425 – Fall 2016 – Boris Glavic 7.26 ©Silberschatz, Korth and Sudarshan

## Redundant Attributes

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

CS425 – Fall 2016 – Boris Glavic 7.27 ©Silberschatz, Korth and Sudarshan

## E-R Diagrams

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

CS425 – Fall 2016 – Boris Glavic 7.28 ©Silberschatz, Korth and Sudarshan

## Entity With Composite, Multivalued, and Derived Attributes

*instructor*

ID  
name  
first\_name  
middle\_initial  
last\_name  
address  
street  
street\_number  
street\_name  
apt\_number  
city  
state  
zip  
{ phone\_number }  
date\_of\_birth  
age ( )

CS425 – Fall 2016 – Boris Glavic 7.29 ©Silberschatz, Korth and Sudarshan

## Entity With Composite, Multivalued, and Derived Attributes

*instructor*

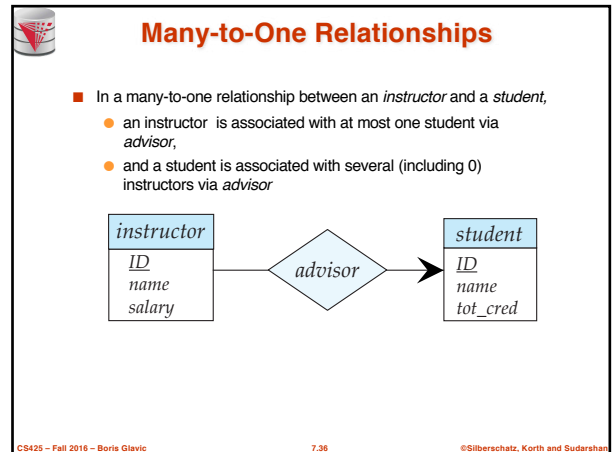
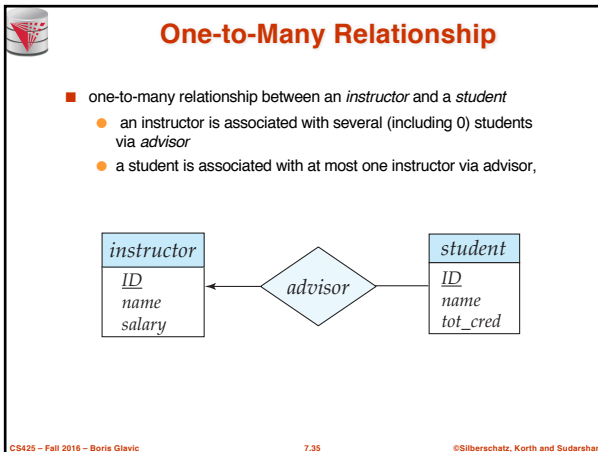
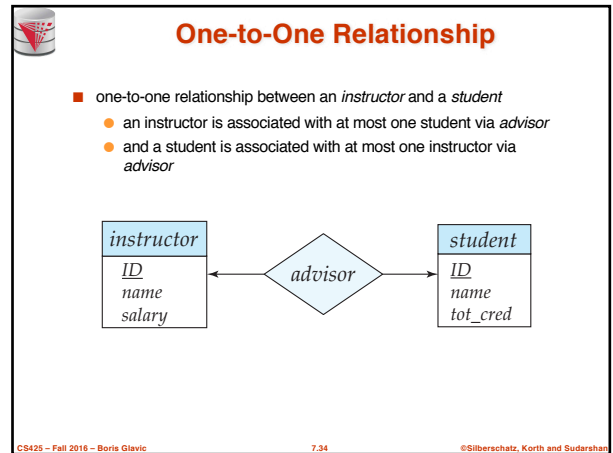
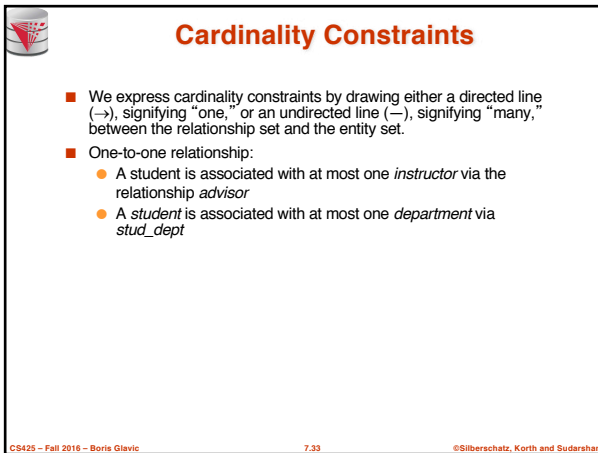
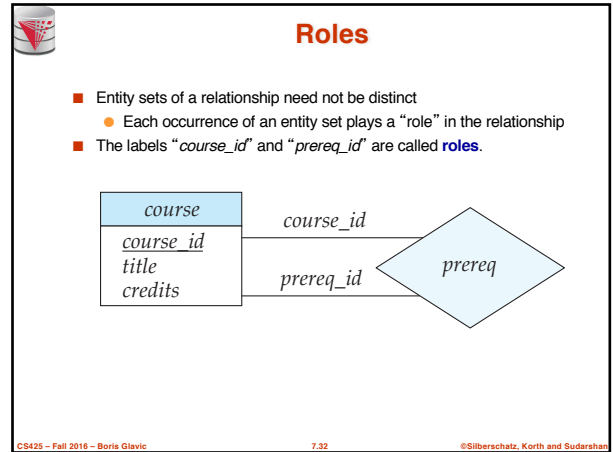
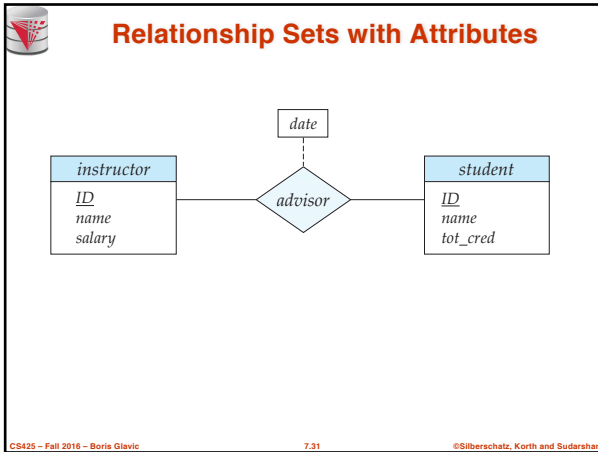
ID  
name  
first\_name  
middle\_initial  
last\_name  
address  
street  
street\_number  
street\_name  
apt\_number  
city  
state  
zip  
{ phone\_number }  
date\_of\_birth  
age ( )

Multi-valued → { phone\_number }

→ composite (first\_name, middle\_initial, last\_name)

→ derived (age ( ))

CS425 – Fall 2016 – Boris Glavic 7.30 ©Silberschatz, Korth and Sudarshan



### Many-to-Many Relationship

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

CS425 – Fall 2016 – Boris Glavic 7.37 ©Silberschatz, Korth and Sudarshan

### Participation of an Entity Set in a Relationship Set

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

CS425 – Fall 2016 – Boris Glavic 7.38 ©Silberschatz, Korth and Sudarshan

### Alternative Notation for Cardinality Limits

- Cardinality limits can also express participation constraints

CS425 – Fall 2016 – Boris Glavic 7.39 ©Silberschatz, Korth and Sudarshan

### Alternative Notation for Cardinality Limits

- Alternative Notation

CS425 – Fall 2016 – Boris Glavic 7.40 ©Silberschatz, Korth and Sudarshan

### E-R Diagram with a Ternary Relationship

CS425 – Fall 2016 – Boris Glavic 7.41 ©Silberschatz, Korth and Sudarshan

### Cardinality Constraints on Ternary Relationship

- We allow at most one arrow out of a ternary (or greater degree) relationship to indicate a cardinality constraint
- E.g., an arrow from *proj\_guide* to *instructor* indicates each student has at most one guide for a project
- If there is more than one arrow, there are two ways of defining the meaning.
  - E.g., a ternary relationship *R* between *A*, *B* and *C* with arrows to *B* and *C* could mean
    - each *A* entity is associated with a unique entity from *B* and *C*
    - each pair of entities from (*A*, *B*) is associated with a unique *C* entity, and each pair (*A*, *C*) is associated with a unique *B*
  - Each alternative has been used in different formalisms
  - To avoid confusion we outlaw more than one arrow
- Better to use cardinality constraints such as (0,n)

CS425 – Fall 2016 – Boris Glavic 7.42 ©Silberschatz, Korth and Sudarshan



## Let's design an ER-model for parts of the university database

Partially taken from Klaus R. Dittrich

modified from: Database System Concepts, 6<sup>th</sup> Ed. ©Silberschatz, Korth and Sudarshan See [www.db-book.com](http://www.db-book.com) for conditions on reuse



## Lets design an ER-model for parts of the university database

- 1) Identify Entities
- 2) Identify Relationship
- 3) Determine Attributes
- 4) Determine Cardinality Constraints

Partially taken from Klaus R. Dittrich

modified from: Database System Concepts, 6<sup>th</sup> Ed. ©Silberschatz, Korth and Sudarshan See [www.db-book.com](http://www.db-book.com) for conditions on reuse



## Weak Entity Sets

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

CS425 – Fall 2016 – Boris Glavic 7.45 ©Silberschatz, Korth and Sudarshan



## Weak Entity Sets (Cont.)

- We underline the discriminator of a weak entity set with a dashed line.
- We put the identifying relationship of a weak entity in a double diamond.
- Primary key for *section* – (*course\_id*, *sec\_id*, *semester*, *year*)



CS425 – Fall 2016 – Boris Glavic 7.46 ©Silberschatz, Korth and Sudarshan



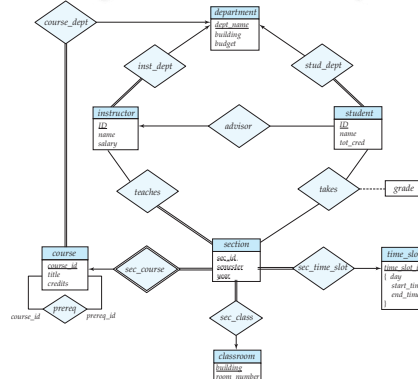
## Weak Entity Sets (Cont.)

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


CS425 – Fall 2016 – Boris Glavic 7.47 ©Silberschatz, Korth and Sudarshan



## E-R Diagram for a University Enterprise




CS425 – Fall 2016 – Boris Glavic 7.48 ©Silberschatz, Korth and Sudarshan



## Reduction to Relational Schemas


CS425 – Fall 2016 – Boris Glavic 7.49  
©Silberschatz, Korth and Sudarshan



## Reduction to Relation Schemas


- Entity sets and relationship sets can be expressed uniformly as *relation schemas* that represent the contents of the database.
- A database which conforms to an E-R diagram can be represented by a collection of relation schemas.
- For each entity set and relationship set there is a unique relation schema that is assigned the name of the corresponding entity set or relationship set.

CS425 – Fall 2016 – Boris Glavic 7.50  
©Silberschatz, Korth and Sudarshan




## Representing Entity Sets With Simple Attributes

- A strong entity set reduces to a schema with the same attributes *student*(ID, name, tot\_cred)
- A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set *section* (course\_id, sec\_id, sem, year)

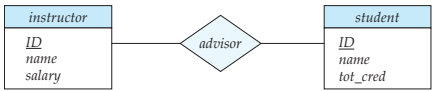


CS425 – Fall 2016 – Boris Glavic 7.51  
©Silberschatz, Korth and Sudarshan




## Representing Relationship Sets

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

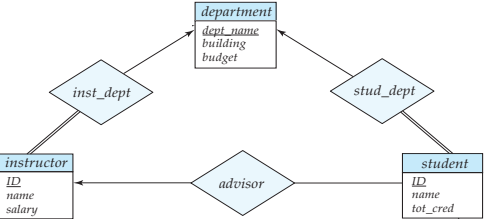


CS425 – Fall 2016 – Boris Glavic 7.52  
©Silberschatz, Korth and Sudarshan




## Redundancy of Schemas

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



CS425 – Fall 2016 – Boris Glavic 7.53  
©Silberschatz, Korth and Sudarshan



## Redundancy of Schemas (Cont.)

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

CS425 – Fall 2016 – Boris Glavic 7.54  
©Silberschatz, Korth and Sudarshan

## Composite and Multivalued Attributes

**instructor**

ID  
name  
  first\_name  
  middle\_initial  
  last\_name  
address  
  street  
  street\_number  
  street\_name  
  apt\_number  
  city  
  state  
  zip  
{ phone\_number }  
date\_of\_birth  
age ( )

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

CS425 - Fall 2016 - Boris Glavic 7.55 ©Silberschatz, Korth and Sudarshan

## Composite and Multivalued Attributes

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

CS425 - Fall 2016 - Boris Glavic 7.56 ©Silberschatz, Korth and Sudarshan

## Multivalued Attributes (Cont.)

- Special case: entity *time\_slot* has only one attribute other than the primary-key attribute, and that attribute is multivalued
  - Optimization: Don't create the relation corresponding to the entity, just create the one corresponding to the multivalued attribute
  - time\_slot*(*time\_slot\_id*, *day*, *start\_time*, *end\_time*)
  - Caveat: *time\_slot* attribute of *section* (from *sec\_time\_slot*) cannot be a foreign key due to this optimization

**section**

sec\_id  
semester  
year

sec\_time\_slot

**time\_slot**

time\_slot\_id  
{ day  
start\_time  
end\_time }

CS425 - Fall 2016 - Boris Glavic 7.57 ©Silberschatz, Korth and Sudarshan

## Design Issues

- Use of entity sets vs. attributes
 

**instructor**

ID  
name  
salary  
phone\_number

inst\_phone

**phone**

phone\_number  
location
- Designing phone as an entity allow for primary key constraints for phone
- Designing phone as an entity allow phone numbers to be used in relationships with other entities (e.g., student)
- Use of phone as an entity allows extra information about phone numbers

CS425 - Fall 2016 - Boris Glavic 7.58 ©Silberschatz, Korth and Sudarshan

## Design Issues

- Use of entity sets vs. relationship sets
  - Possible guideline is to designate a relationship set to describe an action that occurs between entities
  - Possible hint: the relationship only relates entities, but does not have an existence by itself. E.g., hasAddress: (department-address)

**section**

sec\_id  
semester  
year

section\_reg

**registration**

...

student\_reg

**student**

ID  
name  
tot\_cred

CS425 - Fall 2016 - Boris Glavic 7.59 ©Silberschatz, Korth and Sudarshan

## Design Issues

- Binary versus n-ary relationship sets
  - Although it is possible to replace any nonbinary (*n*-ary, for *n* > 2) relationship set by a number of distinct binary relationship sets + an artificial entity set, a *n*-ary relationship set shows more clearly that several entities participate in a single relationship.
- Placement of relationship attributes
  - e.g., attribute *date* as attribute of *advisor* or as attribute of *student*
  - Does not work for *N-M* relationships!

CS425 - Fall 2016 - Boris Glavic 7.60 ©Silberschatz, Korth and Sudarshan



## Binary Vs. Non-Binary Relationships

- Some relationships that appear to be non-binary may be better represented using binary relationships
  - E.g., A ternary relationship *parents*, relating a child to his/her father and mother, is best replaced by two binary relationships, *father* and *mother*
    - Using two binary relationships allows partial information (e.g., only mother being known)
  - But there are some relationships that are naturally non-binary
    - Example: *proj\_guide*

CS425 – Fall 2016 – Boris Glavic

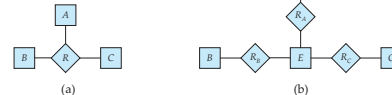
7.61

©Silberschatz, Korth and Sudarshan



## Converting Non-Binary Relationships to Binary Form

- In general, any non-binary relationship can be represented using binary relationships by creating an artificial entity set.
  - Replace  $R$  between 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  $(a, b, c)$  in  $R$ , create
    - a new entity  $e_i$  in the entity set  $E$
    - add  $(e_i, a)$  to  $R_A$
    - add  $(e_i, b)$  to  $R_B$
    - add  $(e_i, c)$  to  $R_C$



CS425 – Fall 2016 – Boris Glavic

7.62

©Silberschatz, Korth and Sudarshan



## Converting Non-Binary Relationships (Cont.)

- Also need to translate constraints
  - Translating all constraints may not be possible
  - There may be instances in the translated schema that cannot correspond to any instance of  $R$ 
    - Exercise: add constraints to the relationships  $R_A$ ,  $R_B$  and  $R_C$  to ensure that a newly created entity corresponds to exactly one entity in each of entity sets  $A$ ,  $B$  and  $C$
  - We can avoid creating an identifying attribute by making  $E$  a weak entity set (described shortly) identified by the three relationship sets

CS425 – Fall 2016 – Boris Glavic

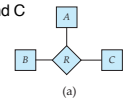
7.63

©Silberschatz, Korth and Sudarshan



## Converting Non-Binary Relationships: Is the New Entity Set E Necessary?

- Yes, because a non-binary relationship stores more information than any number of binary relationships
  - Consider again the example (a) below
  - Replace  $R$  with three binary relationships:
    - $R_{AB}$ , relating  $A$  and  $B$
    - $R_{BC}$ , relating  $B$  and  $C$
    - $R_{AC}$ , relating  $A$  and  $C$
  - For each relationship  $(a, b, c)$  in  $R$ , create
    - 1. add  $(a, b)$  to  $R_{AB}$
    - 2. add  $(b, c)$  to  $R_{BC}$
    - 3. add  $(a, c)$  to  $R_{AC}$
  - Consider  $R = \text{order}$ ,  $A = \text{supplier}$ ,  $B = \text{item}$ ,  $C = \text{customer}$   
 (Gunnar, chainsaw, Bob) – Bob ordered a chainsaw from Gunnar  
 ->  
 (Gunnar, chainsaw), (chainsaw, Bob), (Gunnar, Bob)  
 Gunnar supplies chainsaws, Bob ordered a chainsaw, Bob ordered something from Gunnar. E.g., we do not know what Bob ordered from Gunnar.



CS425 – Fall 2016 – Boris Glavic

7.64

©Silberschatz, Korth and Sudarshan



## ER-model to Relational Summary

- Rule 1) Strong entity  $E$** 
  - Create relation with attributes of  $E$
  - Primary key is equal to the PK of  $E$
- Rule 2) Weak entity  $W$  identified by  $E$  through relationship  $R$** 
  - Create relation with attributes of  $W$  and  $R$  and  $PK(E)$ .
  - Set PK to discriminator attributes combined with  $PK(E)$ .  $PK(E)$  is a foreign key to  $E$ .
- Rule 3) Binary relationship  $R$  between  $A$  and  $B$ : one-to-one**
  - If no side is total add PK of  $A$  to as foreign key in  $B$  or the other way around. Add any attributes of the relationship  $R$  to  $A$  respective  $B$ .
  - If one side is total add PK of the other-side as foreign key. Add any attributes of the relationship  $R$  to the total side.
  - If both sides are total merge the two relation into a new relation  $E$  and choose either  $PK(A)$  as  $PK(B)$  as the new PK. Add any attributes of the relationship  $R$  to the new relation  $E$ .

CS425 – Fall 2016 – Boris Glavic

7.65

©Silberschatz, Korth and Sudarshan



## ER-model to Relational Summary (Cont.)

- Rule 4) Binary relationship  $R$  between  $A$  and  $B$ : one-to-many/many-to-one**
  - Add PK of the "one" side as foreign key to the "many" side.
  - Add any attributes of the relationship  $R$  to the "many" side.
- Rule 5) Binary relationship  $R$  between  $A$  and  $B$ : many-to-many**
  - Create a new relation  $R$ .
  - Add PK's of  $A$  and  $B$  as attributes + plus all attributes of  $R$ .
  - The primary key of the relationship is  $PK(A) + PK(B)$ . The PK attributes of  $A/B$  form a foreign key to  $A/B$
- Rule 6) N-ary relationship  $R$  between  $E_1 \dots E_n$** 
  - Create a new relation.
  - Add all the PK's of  $E_1 \dots E_n$ . Add all attributes of  $R$  to the new relation.
  - The primary key or  $R$  is  $PK(E_1) \dots PK(E_n)$ . Each  $PK(E_i)$  is a foreign key to the corresponding relation.


CS425 – Fall 2016 – Boris Glavic

7.66

©Silberschatz, Korth and Sudarshan








## Extended ER Features

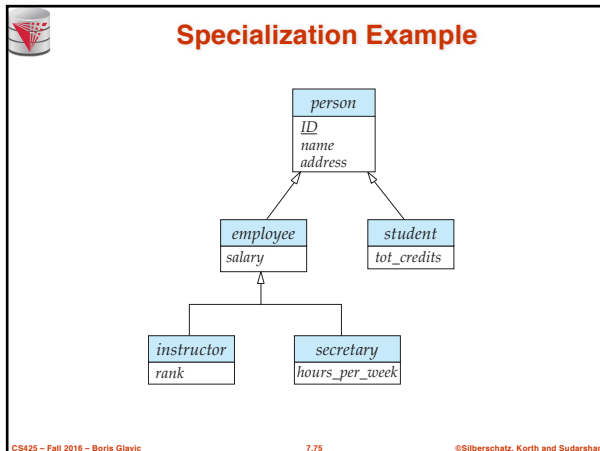

CS425 – Fall 2016 – Boris Glavic 7.73 ©Silberschatz, Korth and Sudarshan



## Extended E-R Features: Specialization

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


CS425 – Fall 2016 – Boris Glavic 7.74 ©Silberschatz, Korth and Sudarshan

## Extended ER Features: Generalization

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


CS425 – Fall 2016 – Boris Glavic 7.76 ©Silberschatz, Korth and Sudarshan



## Specialization and Generalization (Cont.)

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

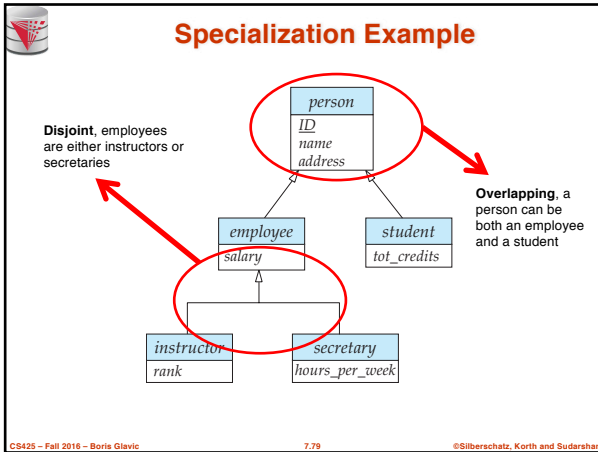
CS425 – Fall 2016 – Boris Glavic 7.77 ©Silberschatz, Korth and Sudarshan



## Design Constraints on a Specialization/Generalization

- Constraint on which entities can be members of a given lower-level entity set.
  - condition-defined
    - Example: all customers over 65 years are members of *senior-citizen* entity set; *senior-citizen* ISA *person*.
  - user-defined
- Constraint on whether or not entities may belong to more than one lower-level entity set within a single generalization.
  - **Disjoint**
    - an entity can belong to only one lower-level entity set
    - Noted in E-R diagram by having multiple lower-level entity sets link to the same triangle
  - **Overlapping**
    - an entity can belong to more than one lower-level entity set

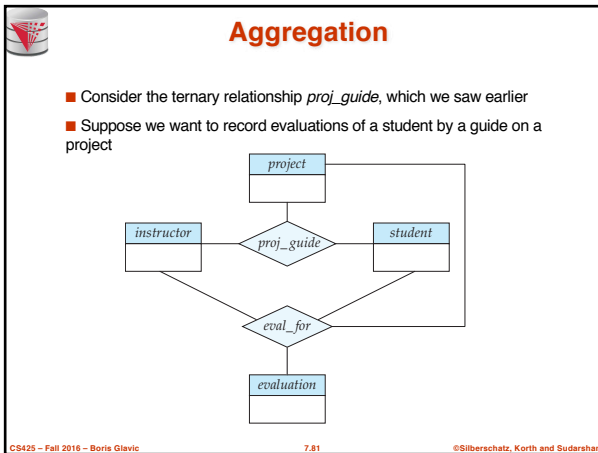
CS425 – Fall 2016 – Boris Glavic 7.78 ©Silberschatz, Korth and Sudarshan



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

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

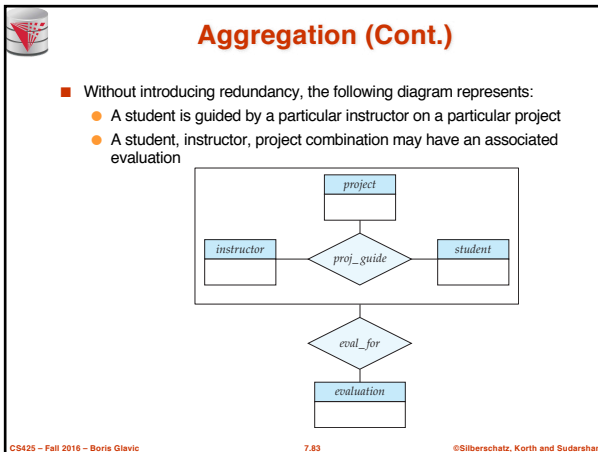
CS425 – Fall 2016 – Boris Glavic 7.80 ©Silberschatz, Korth and Sudarshan



### Aggregation (Cont.)

- Relationship sets *eval\_for* and *proj\_guide* represent overlapping information
  - Every *eval\_for* relationship corresponds to a *proj\_guide* relationship
  - However, some *proj\_guide* relationships may not correspond to any *eval\_for* relationships
    - ▶ So we can't discard the *proj\_guide* relationship
- Eliminate this redundancy via *aggregation*
  - Treat relationship as an abstract entity
  - Allows relationships between relationships
  - Abstraction of relationship into new entity

CS425 – Fall 2016 – Boris Glavic 7.82 ©Silberschatz, Korth and Sudarshan



### Representing Specialization via Schemas

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

schema	attributes
person	ID, name, street, city
student	ID, tot_cred
employee	ID, salary

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

CS425 – Fall 2016 – Boris Glavic 7.84 ©Silberschatz, Korth and Sudarshan

## Representing Specialization as Schemas (Cont.)

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

schema	attributes
person	ID, name, street, city
student	ID, name, street, city, tot_cred
employee	ID, name, street, city, salary
  - If specialization is total, the schema for the generalized entity set (*person*) not required to store information
    - Can be defined as a "view" relation containing union of specialization relations
    - But explicit schema may still be needed for foreign key constraints
  - Drawback: *name*, *street* and *city* may be stored redundantly for people who are both students and employees

CS425 - Fall 2016 - Boris Glavic 7.85 ©Silberschatz, Korth and Sudarshan

## Representing Specialization as Schemas (Cont.)

- Method 3:
  - Form a single relation schema for each entity set with all local and inherited attributes
    - For total and disjoint specialization add a single "type" attribute that stores the type of an entity
 

schema	attributes
person	ID, type, name, street, city, tot_cred, salary
    - For partial and/or overlapping specialization add multiple boolean "type" attributes
 

schema	attributes
person	ID, isEmployee, isStudent, name, street, city, tot_cred, salary
  - Drawback: large number of NULL values, potentially large relation

CS425 - Fall 2016 - Boris Glavic 7.86 ©Silberschatz, Korth and Sudarshan

## Schemas Corresponding to Aggregation

- To represent aggregation, create a schema containing
  - primary key of the aggregated relationship,
  - the primary key of the associated entity set
  - any descriptive attributes

CS425 - Fall 2016 - Boris Glavic 7.87 ©Silberschatz, Korth and Sudarshan

## Schemas Corresponding to Aggregation (Cont.)

- For example, to represent aggregation manages between relationship *works\_on* and entity set *manager*, create a schema *eval\_for*(*s\_ID*, *project\_id*, *i\_ID*, *evaluation\_id*)

CS425 - Fall 2016 - Boris Glavic 7.88 ©Silberschatz, Korth and Sudarshan

## ER-model to Relational Summary (Cont.)

- Rule 8)** Specialization of *E* into  $S_1, \dots, S_n$  (method 1)
  - Create a relation for *E* with all attributes of *E*. The PK of *E* is the PK.
  - For each  $S_i$  create a relation with PK(*E*) as PK and foreign key to relation for *E*. Add all attributes of  $S_i$  that do not exist in *E*.
- Rule 9)** Specialization of *E* into  $S_1, \dots, S_n$  (method 2)
  - Create a relation for *E* with all attributes of *E*. The PK of *E* is the PK.
  - For each  $S_i$  create a relation with PK(*E*) as PK and foreign key to relation for *E*. Add all attributes of  $S_i$ .
- Rule 10)** Specialization of *E* into  $S_1, \dots, S_n$  (method 3)
  - Create a new relation with all attributes from *E* and  $S_1, \dots, S_n$ .
  - Add single attribute type or a boolean type attribute for each  $S_i$
  - The primary key is PK(*E*)

CS425 - Fall 2016 - Boris Glavic 7.89 ©Silberschatz, Korth and Sudarshan

## ER-model to Relational Summary (Cont.)

- Rule 11)** Aggregation: Relationship  $R_1$  relates entity sets  $E_1, \dots, E_n$ . This is related by relationship *A* to an entity set *B*
  - Create a relation for *A* with attributes PK( $E_1$ ) ... PK( $E_n$ ) + all attributes from *A* + PK(*B*). PK are all attributes except the ones from *A*

CS425 - Fall 2016 - Boris Glavic 7.90 ©Silberschatz, Korth and Sudarshan



## ER Design Decisions

- The use of an attribute or entity set to represent an object.
- Whether a real-world concept is best expressed by an entity set or a relationship set.
- The use of a ternary relationship versus a pair of binary relationships.
- The use of a strong or weak entity set.
- The use of specialization/generalization – contributes to modularity in the design.
- The use of aggregation – can treat the aggregate entity set as a single unit without concern for the details of its internal structure.

CS425 – Fall 2016 – Boris Glavic

7.91

©Silberschatz, Korth and Sudarshan



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

Partially taken from  
Klaus R. Dittrich

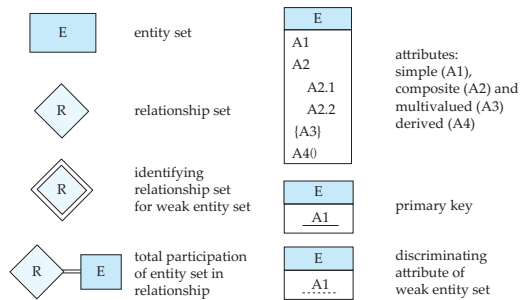
modified from:

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

©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## Summary of Symbols Used in E-R Notation



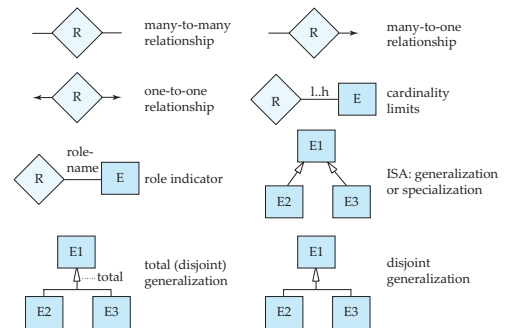
CS425 – Fall 2016 – Boris Glavic

7.93

©Silberschatz, Korth and Sudarshan



## Symbols Used in ER Notation (Cont.)



CS425 – Fall 2016 – Boris Glavic

7.94

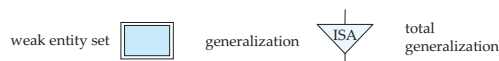
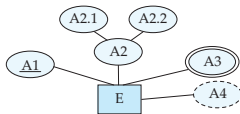
©Silberschatz, Korth and Sudarshan



## Alternative ER Notations

- Chen, IDE1FX, ...

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



CS425 – Fall 2016 – Boris Glavic

7.95

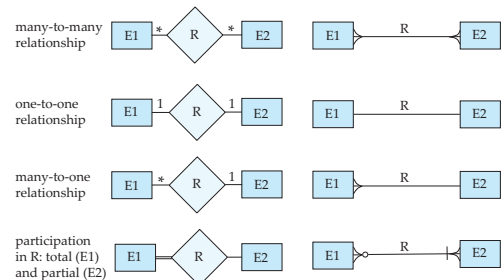
©Silberschatz, Korth and Sudarshan



## Alternative ER Notations

Chen

IDE1FX (Crows feet notation)



CS425 – Fall 2016 – Boris Glavic

7.96

©Silberschatz, Korth and Sudarshan

## UML

- **UML:** Unified Modeling Language
- UML has many components to graphically model different aspects of an entire software system
- UML Class Diagrams correspond to E-R Diagram, but several differences.

CS425 - Fall 2016 - Boris Glavic 7.97 ©Silberschatz, Korth and Sudarshan

## ER vs. UML Class Diagrams

ER Diagram Notation	Equivalent in UML
 entity with attributes (simple, composite, multivalued, derived)	 class with simple attributes and methods (attribute prefixes: + = public, - = private, \$ = protected)
 binary relationship	
 relationship attributes	
 cardinality constraints	

\*Note reversal of position in cardinality constraint depiction

CS425 - Fall 2016 - Boris Glavic 7.98 ©Silberschatz, Korth and Sudarshan

## ER vs. UML Class Diagrams

ER Diagram Notation	Equivalent in UML
 n-ary relationships	
 overlapping generalization	
 disjoint generalization	

\*Generalization can use merged or separate arrows independent of disjoint/overlapping

CS425 - Fall 2016 - Boris Glavic 7.99 ©Silberschatz, Korth and Sudarshan

## UML Class Diagrams (Cont.)

- Binary relationship sets are represented in UML by just drawing a line connecting the entity sets. The relationship set name is written adjacent to the line.
- The role played by an entity set in a relationship set may also be specified by writing the role name on the line, adjacent to the entity set.
- The relationship set name may alternatively be written in a box, along with attributes of the relationship set, and the box is connected, using a dotted line, to the line depicting the relationship set.

CS425 - Fall 2016 - Boris Glavic 7.100 ©Silberschatz, Korth and Sudarshan

## Recap

- ER-model
  - Entities
    - Strong
    - Weak
  - Attributes
    - Simple vs. Composite
    - Single-valued vs. Multi-valued
  - Relationships
    - Degree (binary vs. N-ary)
  - Cardinality constraints
  - Specialization/Generalization
    - Total vs. partial
    - Disjoint vs. overlapping
  - Aggregation

CS425 - Fall 2016 - Boris Glavic 7.101 ©Silberschatz, Korth and Sudarshan

## Recap Cont.

- ER-Diagrams
  - Alternative notations
- UML-Diagrams
- Design decisions
  - Multi-valued attribute vs. entity
  - Entity vs. relationship
  - Binary vs. N-ary relationships
  - Placement of relationship attributes
  - Total 1-1 vs. single entity
- ER to relational model
  - Translation rules

CS425 - Fall 2016 - Boris Glavic 7.102 ©Silberschatz, Korth and Sudarshan



## End of Chapter 7

Partially taken from  
Klaus R. Dittrich

modified from:  
Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on reuse



## Outline

- Introduction
- Relational Data Model
- Formal Relational Languages (relational algebra)
- SQL - Advanced
- **Database Design – Database modelling**
- Transaction Processing, Recovery, and Concurrency Control
- Storage and File Structures
- Indexing and Hashing
- Query Processing and Optimization

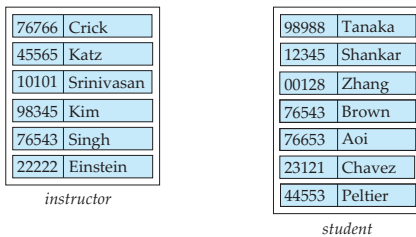
CS425 – Fall 2016 – Boris Glavic

7.104

©Silberschatz, Korth and Sudarshan



### Figure 7.01



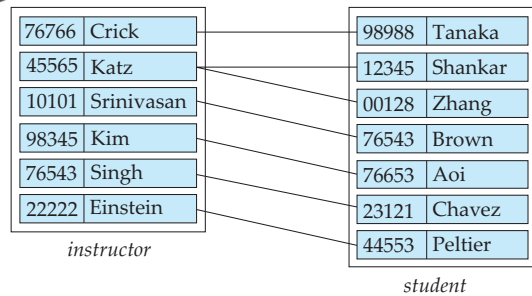
CS425 – Fall 2016 – Boris Glavic

7.105

©Silberschatz, Korth and Sudarshan



### Figure 7.02



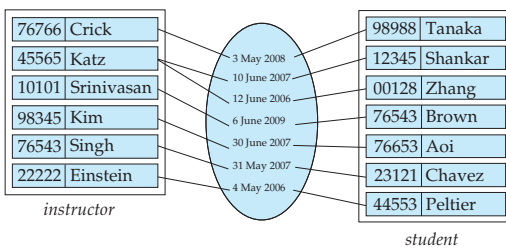
CS425 – Fall 2016 – Boris Glavic

7.106

©Silberschatz, Korth and Sudarshan



### Figure 7.03



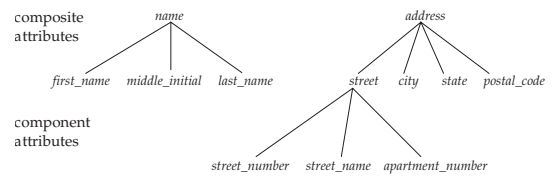
CS425 – Fall 2016 – Boris Glavic

7.107

©Silberschatz, Korth and Sudarshan



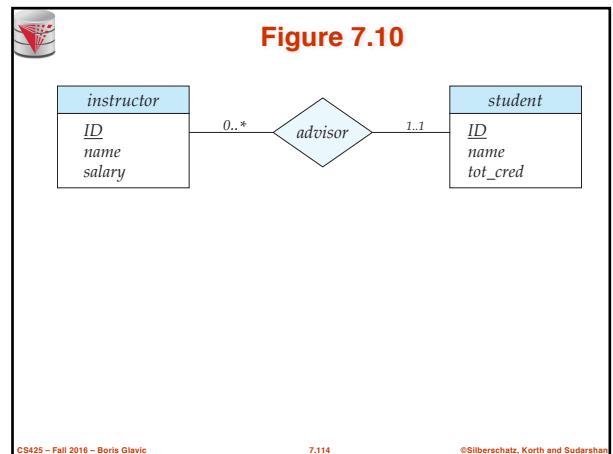
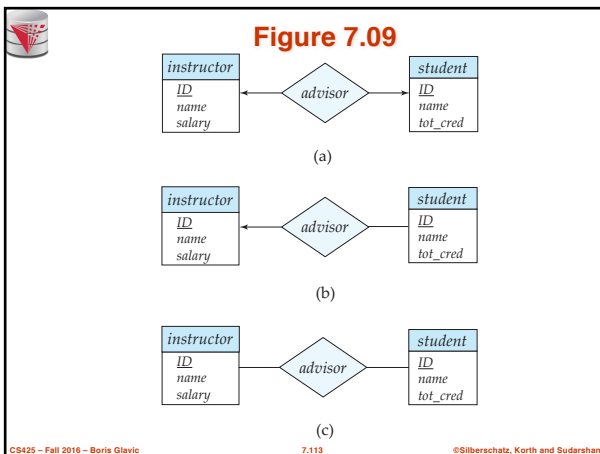
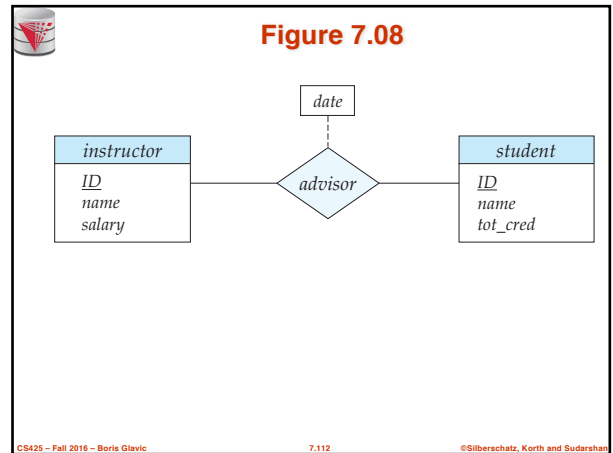
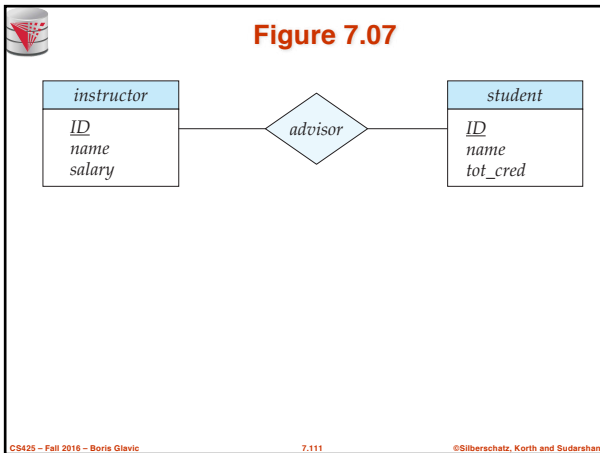
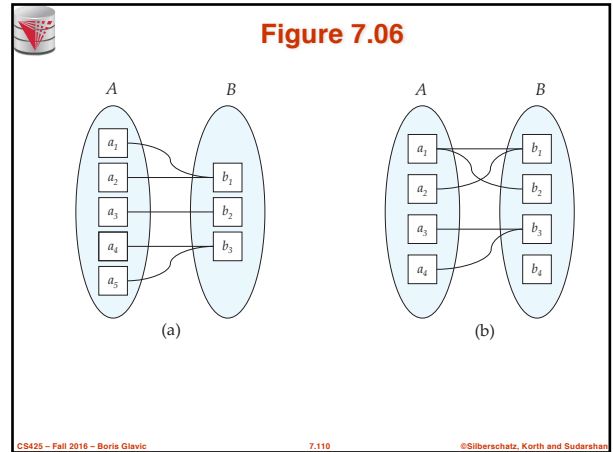
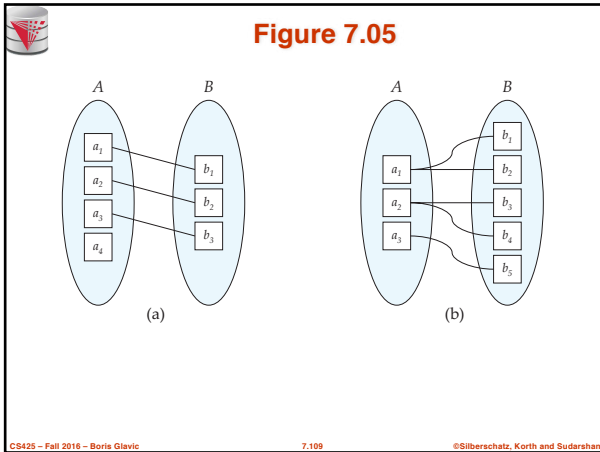
### Figure 7.04



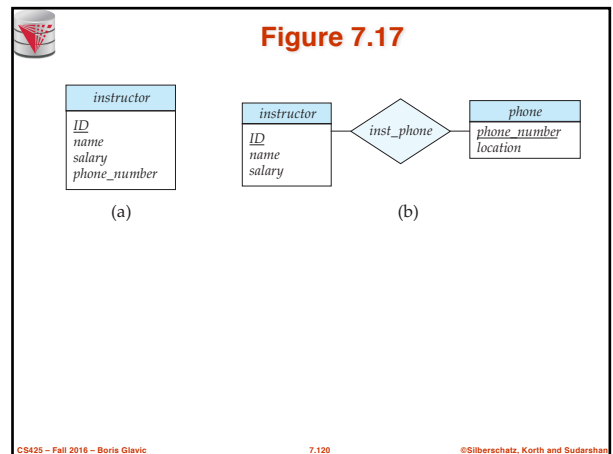
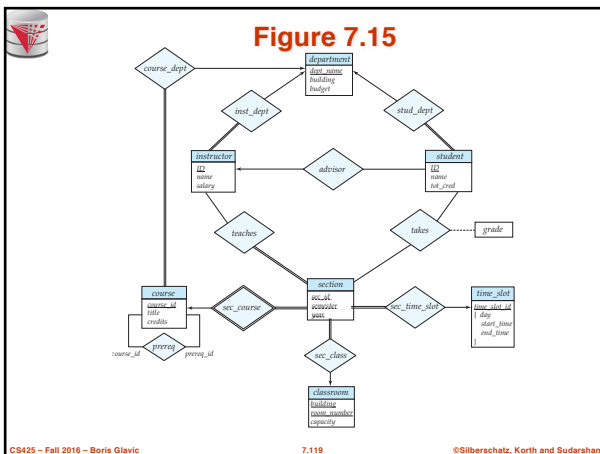
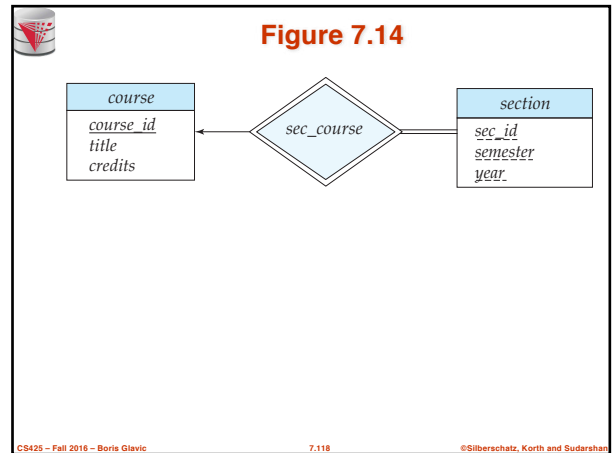
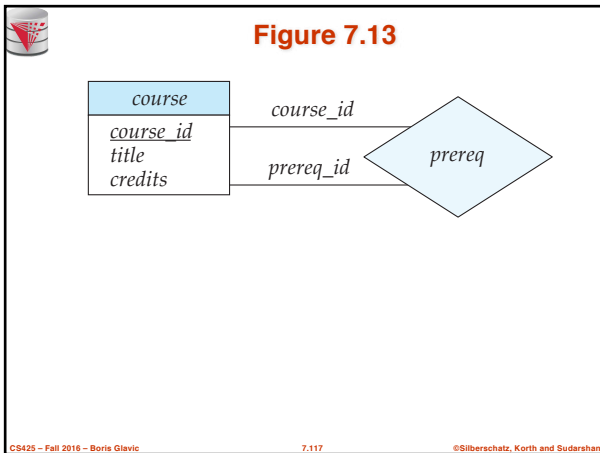
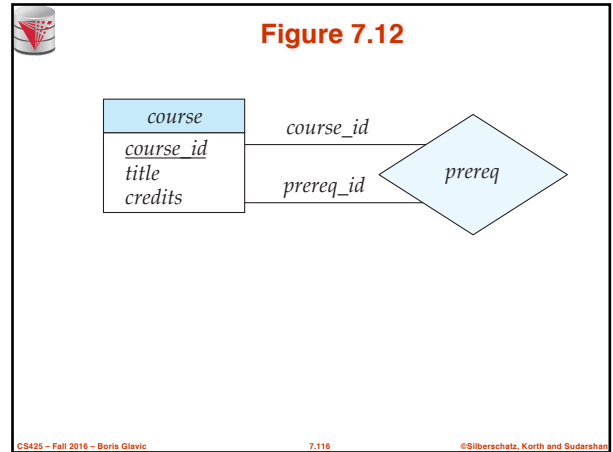
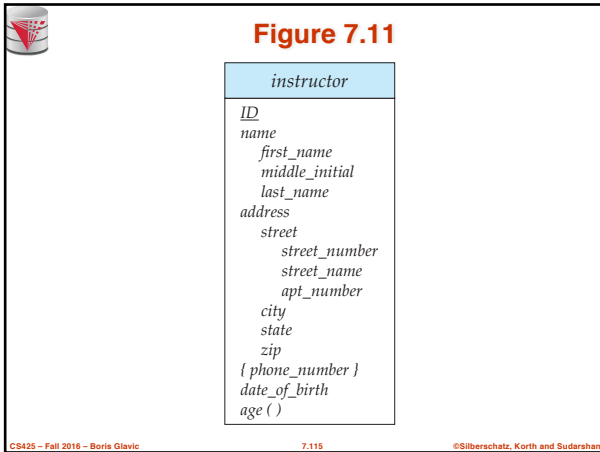
CS425 – Fall 2016 – Boris Glavic

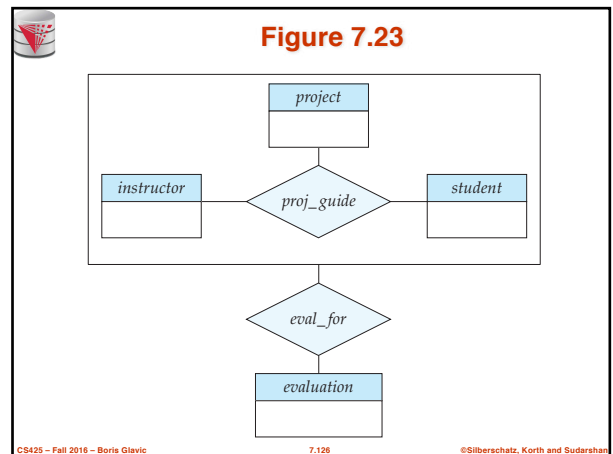
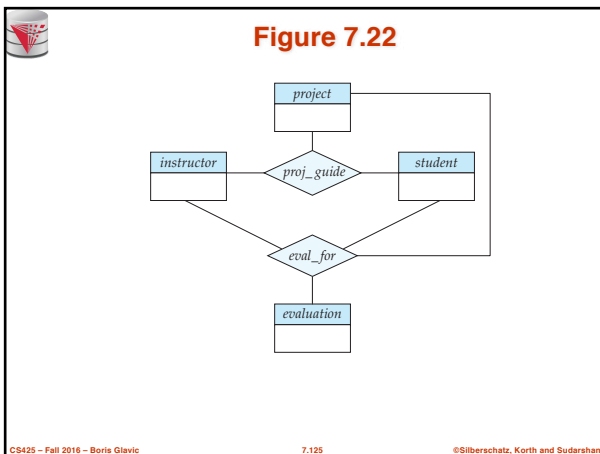
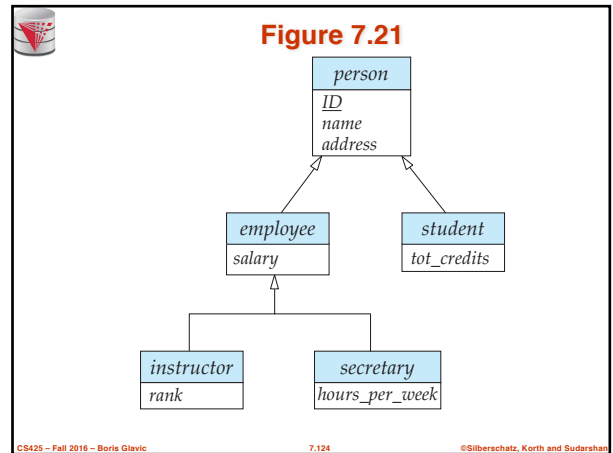
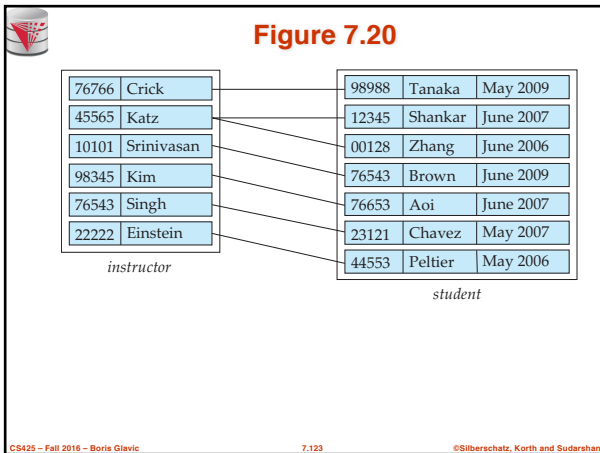
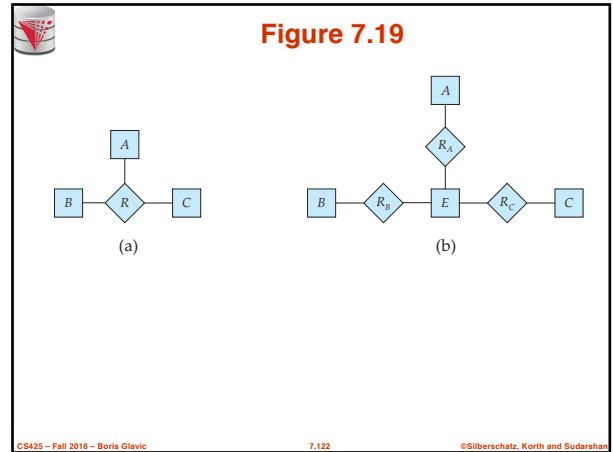
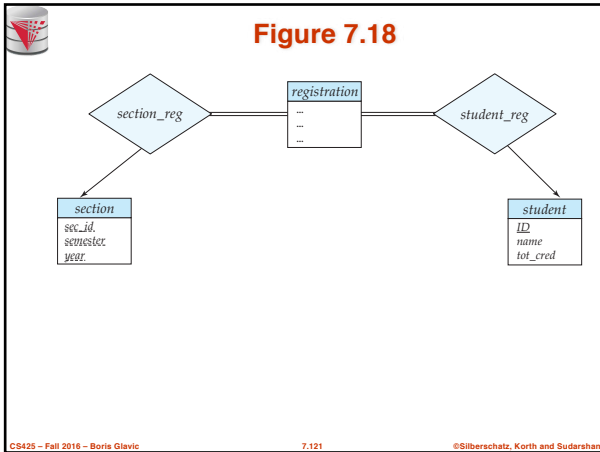
7.108

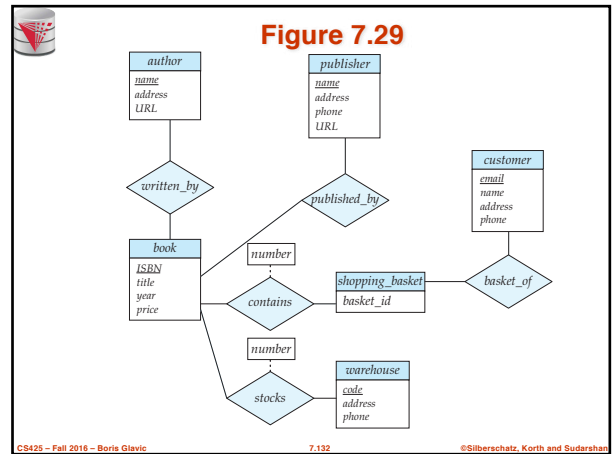
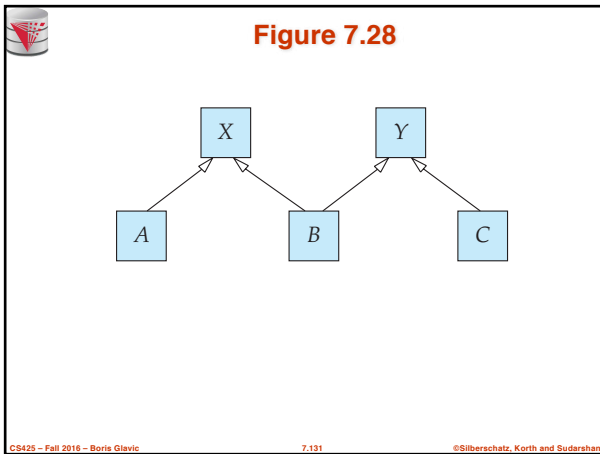
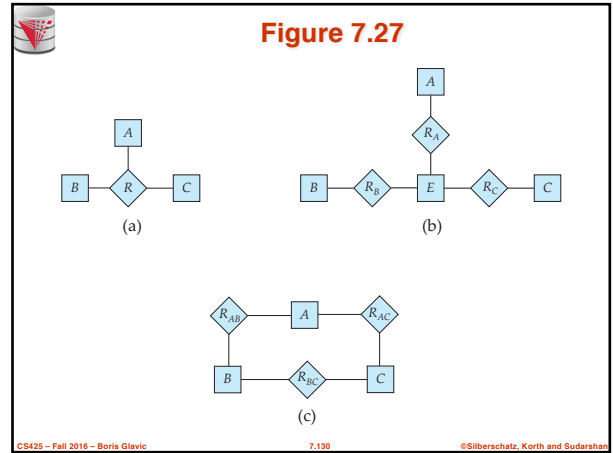
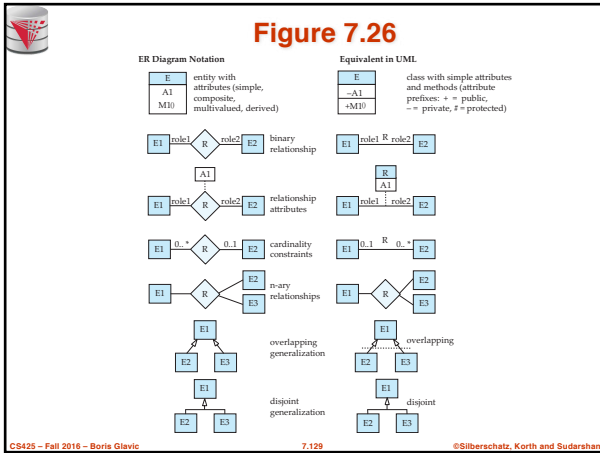
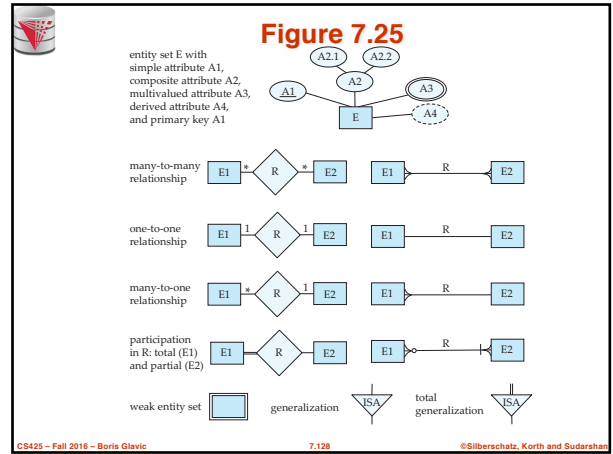
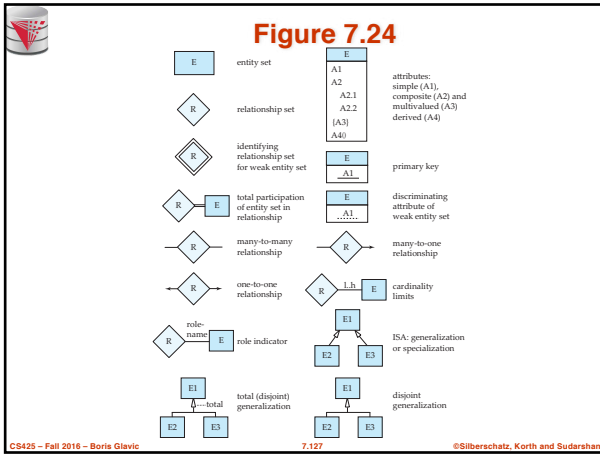
©Silberschatz, Korth and Sudarshan













CS425 – Fall 2013  
Boris Glavic  
Chapter 8: Relational Database Design

modified from:

Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



Chapter 8: Relational Database Design

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- Decomposition Using Functional Dependencies
- Functional Dependency Theory
- Algorithms for Functional Dependencies
- Decomposition Using Multivalued Dependencies
- More Normal Form
- Database-Design Process
- Modeling Temporal Data

CS425 – Fall 2016 – Boris Glavic

8.2

©Silberschatz, Korth and Sudarshan



What is Good Design?  
1) Easier: What is Bad Design?

modified from:

Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



Combine Schemas?

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

ID	name	salary	dept_name	building	budget
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

CS425 – Fall 2016 – Boris Glavic

8.4

©Silberschatz, Korth and Sudarshan



Redundancy is Bad!

- Update Physics Department
  - multiple tuples to update
  - Efficiency + potential for errors
- Delete Physics Department
  - update multiple tuples
  - Efficiency + potential for errors
- Departments without instructor or instructors without departments
  - Need dummy department and dummy instructor
  - Makes aggregation harder and error prone.

ID	name	salary	dept_name	building	budget
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

CS425 – Fall 2016 – Boris Glavic

8.5

©Silberschatz, Korth and Sudarshan



A Combined Schema Without Repetition

- Combining is not always bad!
- Consider combining relations
  - *sec\_class(sec\_id, building, room\_number)* and
  - *section(course\_id, sec\_id, semester, year)*
 into one relation
  - *section(course\_id, sec\_id, semester, year, building, room\_number)*
- No repetition in this case

CS425 – Fall 2016 – Boris Glavic

8.6

©Silberschatz, Korth and Sudarshan

## What About Smaller Schemas?

- Suppose we had started with *inst\_dept*. How would we know to split up (**decompose**) it into *instructor* and *department*?
- Write a rule "if there were a schema (*dept\_name*, *building*, *budget*), then *dept\_name* would be a candidate key"
- Denote as a **functional dependency**:  
 $dept\_name \rightarrow building, budget$
- In *inst\_dept*, because *dept\_name* is not a candidate key, the building and budget of a department may have to be repeated.
  - This indicates the need to decompose *inst\_dept*
- Not all decompositions are good. Suppose we decompose *employee*(*ID*, *name*, *street*, *city*, *salary*) into *employee1*(*ID*, *name*) and *employee2*(*name*, *street*, *city*, *salary*)
- The next slide shows how we lose information -- we cannot reconstruct the original *employee* relation -- and so, this is a **lossy decomposition**.

CS425 – Fall 2016 – Boris Glavic 8.7 ©Silberschatz, Korth and Sudarshan

## A Lossy Decomposition

The original *employee* table has 3 rows. It is decomposed into *employee1* (ID, name) and *employee2* (name, street, city, salary). A natural join of these two tables results in a table with 4 rows, including two spurious rows that were not in the original table.

CS425 – Fall 2016 – Boris Glavic 8.8 ©Silberschatz, Korth and Sudarshan

## Example of Lossless-Join Decomposition

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

The original table  $R$  has 2 rows. It is decomposed into  $R_1$  and  $R_2$ . The natural join of  $R_1$  and  $R_2$  reconstructs the original table  $R$ .

CS425 – Fall 2016 – Boris Glavic 8.9 ©Silberschatz, Korth and Sudarshan

## Goals of Lossless-Join Decomposition

- Lossless-Join decomposition means splitting a table in a way so that we do not lose information
  - That means we should be able to reconstruct the original table from the decomposed table using joins

The original table  $R$  has 2 rows. It is decomposed into  $R_1$  and  $R_2$ . The natural join of  $R_1$  and  $R_2$  does not reconstruct the original table  $R$ .

CS425 – Fall 2016 – Boris Glavic 8.10 ©Silberschatz, Korth and Sudarshan

## Goal – Devise a Theory for the Following

- Decide whether a particular relation  $R$  is in "good" form.
- In the case that a relation  $R$  is not in "good" form, decompose it into a set of relations  $\{R_1, R_2, \dots, R_n\}$  such that
  - each relation is in good form
  - the decomposition is a lossless-join decomposition
- Our theory is based on:
  - Models of dependency between attribute values
    - functional dependencies
    - multivalued dependencies
  - Concept of **lossless decomposition**
  - Normal Forms** Based On
    - Atomicity of values
    - Avoidance of redundancy
    - Lossless decomposition

CS425 – Fall 2016 – Boris Glavic 8.11 ©Silberschatz, Korth and Sudarshan

## Modeling Dependencies between Attribute Values:

### Functional Dependencies

### Multivalued Dependencies

modified from:  
 Database System Concepts, 6<sup>th</sup> Ed.  
 ©Silberschatz, Korth and Sudarshan  
 See [www.db-book.com](http://www.db-book.com) for conditions on re-use

## Functional Dependencies

- Constraints on the set of legal instances for a relation schema.
- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
- A functional dependency is a generalization of the notion of a *key*.
  - Thus, every key is a functional dependency

CS425 – Fall 2016 – Boris Glavic 8.13 ©Silberschatz, Korth and Sudarshan

## Functional Dependencies (Cont.)

- Let  $R$  be a relation schema  
 $\alpha \subseteq R$  and  $\beta \subseteq R$
- The **functional dependency**  $\alpha \rightarrow \beta$  **holds on**  $R$  if and only if for any legal relations  $r(R)$ , whenever any two tuples  $t_1$  and  $t_2$  of  $r$  agree on the attributes  $\alpha$ , they also agree on the attributes  $\beta$ . That is,
 
$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$
- Example: Consider  $r(A,B)$  with the following instance of  $r$ .
 

1	4
1	5
3	7
- On this instance,  $A \rightarrow B$  does **NOT** hold, but  $B \rightarrow A$  does hold.

CS425 – Fall 2016 – Boris Glavic 8.14 ©Silberschatz, Korth and Sudarshan

## Functional Dependencies (Cont.)

- Let  $R$  be a relation schema  
 $\alpha \subseteq R$  and  $\beta \subseteq R$
- The **functional dependency**  $\alpha \rightarrow \beta$  **holds on**  $R$  if and only if for any legal relations  $r(R)$ , whenever any two tuples  $t_1$  and  $t_2$  of  $r$  agree on the attributes  $\alpha$ , they also agree on the attributes  $\beta$ . That is,
 
$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$
- Example: Consider  $r(A,B)$  with the following instance of  $r$ .
 

1	4	$A = 1$ and $B = 4$ $A = 1$ and $B = 5$
1	5	
3	7	
- On this instance,  $A \rightarrow B$  does **NOT** hold, but  $B \rightarrow A$  does hold.

CS425 – Fall 2016 – Boris Glavic 8.15 ©Silberschatz, Korth and Sudarshan

## Functional Dependencies (Cont.)

- $K$  is a superkey for relation schema  $R$  if and only if  $K \rightarrow R$
- $K$  is a candidate key for  $R$  if and only if
  - $K \rightarrow R$ , and
  - for no  $\alpha \subset K$ ,  $\alpha \rightarrow R$
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:  $inst\_dept(ID, name, salary, dept\_name, building, budget)$ . We expect these functional dependencies to hold:
 
$$dept\_name \rightarrow building$$
 and
 
$$ID \rightarrow building$$
 but would not expect the following to hold:
 
$$dept\_name \rightarrow salary$$

CS425 – Fall 2016 – Boris Glavic 8.16 ©Silberschatz, Korth and Sudarshan

## Use of Functional Dependencies


- We use functional dependencies to:
  - test relations to see if they are legal under a given set of functional dependencies.
    - If a relation  $r$  is legal under a set  $F$  of functional dependencies, we say that  $r$  **satisfies**  $F$ .
  - specify constraints on the set of legal relations
    - We say that  $F$  **holds on**  $R$  if all legal relations on  $R$  satisfy the set of functional dependencies  $F$ .
- Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.
  - For example, a specific instance of *instructor* may, by chance, satisfy  $name \rightarrow ID$ .

CS425 – Fall 2016 – Boris Glavic 8.17 ©Silberschatz, Korth and Sudarshan

## Functional Dependencies (Cont.)

- A functional dependency is **trivial** if it is satisfied by all instances of a relation
  - Example:
    - $ID, name \rightarrow ID$
    - $name \rightarrow name$
  - In general,  $\alpha \rightarrow \beta$  is trivial if  $\beta \subseteq \alpha$ .


CS425 – Fall 2016 – Boris Glavic 8.18 ©Silberschatz, Korth and Sudarshan



## Closure of a Set of Functional Dependencies

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


CS425 – Fall 2016 – Boris Glavic 8.19 ©Silberschatz, Korth and Sudarshan



## Functional-Dependency Theory

- We now consider the formal theory that tells us which functional dependencies are implied logically by a given set of functional dependencies.
- How do we get the initial set of FDs?
  - Semantics of the domain we are modelling
  - Has to be provided by a human (the designer)
- Example:
  - Relation Citizen(SSN, FirstName, LastName, Address)
  - We know that SSN is unique and a person has a unique SSN
  - Thus,  $SSN \rightarrow FirstName, LastName$


CS425 – Fall 2016 – Boris Glavic 8.20 ©Silberschatz, Korth and Sudarshan



## Closure of a Set of Functional Dependencies

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


CS425 – Fall 2016 – Boris Glavic 8.21 ©Silberschatz, Korth and Sudarshan



## Example

- $R = (A, B, C, G, H, I)$   
 $F = \{ A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H \}$
- some members of  $F^+$ 
  - $A \rightarrow H$ 
    - ▶ by transitivity from  $A \rightarrow B$  and  $B \rightarrow H$
  - $AG \rightarrow I$ 
    - ▶ by augmenting  $A \rightarrow C$  with  $G$ , to get  $AG \rightarrow CG$  and then transitivity with  $CG \rightarrow I$
  - $CG \rightarrow HI$ 
    - ▶ by augmenting  $CG \rightarrow I$  to infer  $CG \rightarrow CGI$ , and augmenting of  $CG \rightarrow H$  to infer  $CGI \rightarrow HI$ , and then transitivity


CS425 – Fall 2016 – Boris Glavic 8.22 ©Silberschatz, Korth and Sudarshan



## Prove Additional Implications

- Prove or disprove the following rules from Armstrong's axioms
  - 1)  $A \rightarrow B, C$  implies  $A \rightarrow B$  and  $A \rightarrow C$
  - 2)  $A \rightarrow B$  and  $A \rightarrow C$  implies  $A \rightarrow B, C$
  - 3)  $A, B \rightarrow B, C$  implies  $A \rightarrow C$
  - 4)  $A \rightarrow B$  and  $C \rightarrow D$  implies  $A, C \rightarrow B, D$

CS425 – Fall 2016 – Boris Glavic 8.23 ©Silberschatz, Korth and Sudarshan



## Procedure for Computing $F^+$

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

```

 $F^+ = F$ 
repeat
  for each functional dependency  $f$  in  $F^+$ 
    apply reflexivity and augmentation rules on  $f$ 
    add the resulting functional dependencies to  $F^+$ 
  for each pair of functional dependencies  $f_1$  and  $f_2$  in  $F^+$ 
    if  $f_1$  and  $f_2$  can be combined using transitivity
      then add the resulting functional dependency to  $F^+$ 
until  $F^+$  does not change any further
      
```
- NOTE: We shall see an alternative more efficient procedure for this task later

CS425 – Fall 2016 – Boris Glavic 8.24 ©Silberschatz, Korth and Sudarshan

## Closure of Functional Dependencies (Cont.)

- Additional rules:
  - If  $\alpha \rightarrow \beta$  holds and  $\alpha \rightarrow \gamma$  holds, then  $\alpha \rightarrow \beta\gamma$  holds (**union**)
  - If  $\alpha \rightarrow \beta\gamma$  holds, then  $\alpha \rightarrow \beta$  holds and  $\alpha \rightarrow \gamma$  holds (**decomposition**)
  - If  $\alpha \rightarrow \beta$  holds and  $\gamma\beta \rightarrow \delta$  holds, then  $\alpha\gamma \rightarrow \delta$  holds (**pseudotransitivity**)

The above rules can be inferred from Armstrong's axioms.

CS425 - Fall 2016 - Boris Glavic 8.25 ©Silberschatz, Korth and Sudarshan

## Closure of Attribute Sets

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

```

result :=  $\alpha$ ;
while (changes to result) do
  for each  $\beta \rightarrow \gamma$  in  $F$  do
    begin
      if  $\beta \subseteq \text{result}$  then result := result  $\cup$   $\gamma$ 
    end

```

CS425 - Fall 2016 - Boris Glavic 8.26 ©Silberschatz, Korth and Sudarshan

## Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$
- $(AG)^+$ 
  - result = AG
  - result = ABCG ( $A \rightarrow C$  and  $A \rightarrow B$ )
  - result = ABCGH ( $CG \rightarrow H$  and  $CG \subseteq AGBC$ )
  - result = ABCGHI ( $CG \rightarrow I$  and  $CG \subseteq AGBC$ )
- Is AG a candidate key?
  - Is AG a superkey?
    - Does  $AG \rightarrow R$ ?  $\equiv$  Is  $(AG)^+ \supseteq R$
  - Is any subset of AG a superkey?
    - Does  $A \rightarrow R$ ?  $\equiv$  Is  $(A)^+ \supseteq R$
    - Does  $G \rightarrow R$ ?  $\equiv$  Is  $(G)^+ \supseteq R$

CS425 - Fall 2016 - Boris Glavic 8.27 ©Silberschatz, Korth and Sudarshan

## Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- Testing for superkey:
  - To test if  $\alpha$  is a superkey, we compute  $\alpha^+$  and check if  $\alpha^+$  contains all attributes of  $R$ .
- Testing functional dependencies
  - To check if a functional dependency  $\alpha \rightarrow \beta$  holds (or, in other words, is in  $F^+$ ), just check if  $\beta \subseteq \alpha^+$ .
  - That is, we compute  $\alpha^+$  by using attribute closure, and then check if it contains  $\beta$ .
  - Is a simple and cheap test, and very useful
- Computing closure of  $F$ 
  - For each  $\gamma \subseteq R$ , we find the closure  $\gamma^+$ , and for each  $S \subseteq \gamma^+$ , we output a functional dependency  $\gamma \rightarrow S$ .

CS425 - Fall 2016 - Boris Glavic 8.28 ©Silberschatz, Korth and Sudarshan

## $O(n)$ Algorithm for Attribute Closure

- Data Structures
  - Enumerate the FDs and attributes
  - int[] c**: an integer array with one element per FD that is initialized to the size of the LHS of the FD
  - list<int>[] rhs**: an array of lists with one element per FD. The element stores the numeric ID of the attributes of the FDs RHS
  - list<int>[] lhs**: an array of lists of integers, one element per attribute. The element for each attribute stores the numeric IDs of the FDs that have the attribute in its LHS
  - set<int> aplus**: a set storing the attributes currently established to be implied by  $A$
  - stack<int> todo**: a stack of attributes to be processed next

CS425 - Fall 2016 - Boris Glavic 8.29 ©Silberschatz, Korth and Sudarshan

## $O(n)$ Algorithm for Attribute Closure

- Algorithm
  - Initialize **c**, **rhs**, **lhs**, **aplus** to the emptyset, **todo** to  $A$

```

while(!todo.isEmpty) {
  curA = todo.pop();
  aplus.add(curA); // add curA to result
  for fd in lhs[curA] { // update how many attribute found for LHS
    c[fd]--; // found a LHS attr for fd
    if (c[fd] == 0) {
      remove(lhs[curA], fd); // avoid firing twice
      for newA in rhs[fd] { // add implied attributes
        if (!aplus[newA]) // if attribute is new add to todo
          todo.push(newA);
        aplus.add(newA);
      }
    }
  }
}

```

CS425 - Fall 2016 - Boris Glavic 8.30 ©Silberschatz, Korth and Sudarshan





## Canonical Cover

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
  - For example:  $A \rightarrow C$  is redundant in:  $\{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$
  - Parts of a functional dependency may be redundant
    - ▷ E.g.: on RHS:  $\{A \rightarrow B, B \rightarrow C, A \rightarrow CD\}$  can be simplified to
 
$$\{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$$
    - ▷ E.g.: on LHS:  $\{A \rightarrow B, B \rightarrow C, AC \rightarrow D\}$  can be simplified to
 
$$\{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$$
- Intuitively, a **canonical cover** of  $F$  is a "minimal" set of functional dependencies equivalent to  $F$ , having no redundant dependencies or redundant parts of dependencies

CS425 – Fall 2016 – Boris Glavic

8.31

©Silberschatz, Korth and Sudarshan



## Extraneous Attributes

- Consider a set  $F$  of functional dependencies and the functional dependency  $\alpha \rightarrow \beta$  in  $F$ .
  - Attribute  $A$  is **extraneous** in  $\alpha$  if  $A \in \alpha$  and  $F$  logically implies  $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ .
  - Attribute  $A$  is **extraneous** in  $\beta$  if  $A \in \beta$  and the set of functional dependencies  $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$  logically implies  $F$ .
- *Note:* implication in the opposite direction is trivial in each of the cases above, since a "stronger" functional dependency always implies a weaker one
- Example: Given  $F = \{A \rightarrow C, AB \rightarrow C\}$ 
  - $B$  is extraneous in  $AB \rightarrow C$  because  $\{A \rightarrow C, AB \rightarrow C\}$  logically implies  $A \rightarrow C$  (i.e. the result of dropping  $B$  from  $AB \rightarrow C$ ).
- Example: Given  $F = \{A \rightarrow C, AB \rightarrow CD\}$ 
  - $C$  is extraneous in  $AB \rightarrow CD$  since  $AB \rightarrow C$  can be inferred even after deleting  $C$

CS425 – Fall 2016 – Boris Glavic

8.32

©Silberschatz, Korth and Sudarshan



## Testing if an Attribute is Extraneous

- Consider a set  $F$  of functional dependencies and the functional dependency  $\alpha \rightarrow \beta$  in  $F$ .
- To test if attribute  $A \in \alpha$  is extraneous in  $\alpha$ 
  1. compute  $(\{\alpha\} - A)^+$  using the dependencies in  $F$
  2. check that  $(\{\alpha\} - A)^+$  contains  $\beta$ ; if it does,  $A$  is extraneous in  $\alpha$
- To test if attribute  $A \in \beta$  is extraneous in  $\beta$ 
  1. compute  $\alpha^+$  using only the dependencies in  $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ ,
  2. check that  $\alpha^+$  contains  $A$ ; if it does,  $A$  is extraneous in  $\beta$

CS425 – Fall 2016 – Boris Glavic

8.33

©Silberschatz, Korth and Sudarshan



## Canonical Cover

- A **canonical cover** for  $F$  is a set of dependencies  $F_c$  such that
  - $F$  logically implies all dependencies in  $F_c$ , and
  - $F_c$  logically implies all dependencies in  $F$ , and
  - No functional dependency in  $F_c$  contains an extraneous attribute, and
  - Each left side of functional dependency in  $F_c$  is unique.
- To compute a canonical cover for  $F$ :
  - repeat** Use the union rule to replace any dependencies in  $F$   $\alpha_1 \rightarrow \beta_1$  and  $\alpha_1 \rightarrow \beta_2$  with  $\alpha_1 \rightarrow \beta_1 \beta_2$
  - Find a functional dependency  $\alpha \rightarrow \beta$  with an extraneous attribute either in  $\alpha$  or in  $\beta$
  - Note:* test for extraneous attributes done using  $F_c$ , not  $F'$
  - If an extraneous attribute is found, delete it from  $\alpha \rightarrow \beta$
  - until**  $F$  does not change
- *Note:* Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied

CS425 – Fall 2016 – Boris Glavic

8.34

©Silberschatz, Korth and Sudarshan



## Computing a Canonical Cover

- $R = (A, B, C)$   
 $F = \{A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C\}$
- Combine  $A \rightarrow BC$  and  $A \rightarrow B$  into  $A \rightarrow BC$ 
  - Set is now  $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- $A$  is extraneous in  $AB \rightarrow C$ 
  - Check if the result of deleting  $A$  from  $AB \rightarrow C$  is implied by the other dependencies
    - ▷ Yes: in fact,  $B \rightarrow C$  is already present!
  - Set is now  $\{A \rightarrow BC, B \rightarrow C\}$
- $C$  is extraneous in  $A \rightarrow BC$ 
  - Check if  $A \rightarrow C$  is logically implied by  $A \rightarrow B$  and the other dependencies
    - ▷ Yes: using transitivity on  $A \rightarrow B$  and  $B \rightarrow C$ .
    - Can use attribute closure of  $A$  in more complex cases
- The canonical cover is:
 
$$\begin{array}{l} A \rightarrow B \\ B \rightarrow C \end{array}$$

CS425 – Fall 2016 – Boris Glavic

8.35

©Silberschatz, Korth and Sudarshan




## Lossless Join-Decomposition Dependency Preservation

modified from:

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


©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## So Far

- Theory of dependencies
- What is missing?
  - When is a decomposition loss-less
    - ↳ Lossless-join decomposition
    - ↳ Dependencies on the input are preserved
- What else is missing?
  - Define what constitutes a good relation
    - ↳ Normal forms
  - How to check for a good relation
    - ↳ Test normal forms
  - How to achieve a good relation
    - ↳ Translate into normal form
    - ↳ Involves decomposition

CS425 – Fall 2016 – Boris Glavic 8.37 ©Silberschatz, Korth and Sudarshan




## Lossless-join Decomposition

- For the case of  $R = (R_1, R_2)$ , we require that for all possible relation instances  $r$  on schema  $R$ 

$$r = \prod_{R_1}(r) \bowtie \prod_{R_2}(r)$$
- A decomposition of  $R$  into  $R_1$  and  $R_2$  is lossless join if at least one of the following dependencies is in  $F^+$ :
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$
- The above functional dependencies are a **sufficient** condition for lossless join decomposition; the dependencies are a **necessary** condition only if all constraints are functional dependencies


CS425 – Fall 2016 – Boris Glavic 8.38 ©Silberschatz, Korth and Sudarshan



## Example

- $R = (A, B, C)$   
 $F = \{A \rightarrow B, B \rightarrow C\}$ 
  - Can be decomposed in two different ways
- $R_1 = (A, B), R_2 = (B, C)$ 
  - Lossless-join decomposition:
 
$$R_1 \cap R_2 = \{B\} \text{ and } B \rightarrow BC$$
  - Dependency preserving
- $R_1 = (A, B), R_2 = (A, C)$ 
  - Lossless-join decomposition:
 
$$R_1 \cap R_2 = \{A\} \text{ and } A \rightarrow AB$$
  - Not dependency preserving  
 (cannot check  $B \rightarrow C$  without computing  $R_1 \bowtie R_2$ )


CS425 – Fall 2016 – Boris Glavic 8.39 ©Silberschatz, Korth and Sudarshan



## Dependency Preservation

- Let  $F_i$  be the set of dependencies  $F^+$  that include only attributes in  $R_i$ .
  - ↳ A decomposition is **dependency preserving**, if
 
$$(F_1 \cup F_2 \cup \dots \cup F_n)^+ = F^+$$
  - ↳ If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.


CS425 – Fall 2016 – Boris Glavic 8.40 ©Silberschatz, Korth and Sudarshan



## Testing for Dependency Preservation

- To check if a dependency  $\alpha \rightarrow \beta$  is preserved in a decomposition of  $R$  into  $R_1, R_2, \dots, R_n$ , we apply the following test (with attribute closure done with respect to  $F$ )
  - $result = \alpha$   
**while** (changes to  $result$ ) do  
   **for each**  $R_i$  in the decomposition  
      $t = (result \cap R_i)^+ \cap R_i$   
      $result = result \cup t$
  - If  $result$  contains all attributes in  $\beta$ , then the functional dependency  $\alpha \rightarrow \beta$  is preserved.
- We apply the test on all dependencies in  $F$  to check if a decomposition is dependency preserving
- This procedure (attribute closure) takes polynomial time, instead of the exponential time required to compute  $F^+$  and  $(F_1 \cup F_2 \cup \dots \cup F_n)^+$

CS425 – Fall 2016 – Boris Glavic 8.41 ©Silberschatz, Korth and Sudarshan



## Example

- $R = (A, B, C)$   
 $F = \{A \rightarrow B, B \rightarrow C\}$   
 Key =  $\{A\}$
- Decomposition  $R_1 = (A, B), R_2 = (B, C)$ 
  - Lossless-join decomposition
  - Dependency preserving

CS425 – Fall 2016 – Boris Glavic 8.42 ©Silberschatz, Korth and Sudarshan



## Normal Forms

modified from:

Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## So Far

- Theory of dependencies
- Decompositions and ways to check whether they are “good”
  - Lossless
  - Dependency preserving
- What is missing?
  - Define what constitutes a good relation
    - Normal forms
  - How to check for a good relation
    - Test normal forms
  - How to achieve a good relation
    - Translate into normal form
    - Involves decomposition

CS425 – Fall 2016 – Boris Glavic

8.44

©Silberschatz, Korth and Sudarshan



## Goals of Normalization

- Let  $R$  be a relation scheme with a set  $F$  of functional dependencies.
- Decide whether a relation scheme  $R$  is in “good” form.
- In the case that a relation scheme  $R$  is not in “good” form, decompose it into a set of relation scheme  $\{R_1, R_2, \dots, R_n\}$  such that
  - each relation scheme is in good form
  - the decomposition is a lossless-join decomposition
  - Preferably, the decomposition should be dependency preserving.

CS425 – Fall 2016 – Boris Glavic

8.45

©Silberschatz, Korth and Sudarshan



## First Normal Form

- A domain is **atomic** if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Set of names, composite attributes
    - Identification numbers like CS101 that can be broken up into parts
- A relational schema  $R$  is in **first normal form** if the domains of all attributes of  $R$  are atomic
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
  - Example: Set of accounts stored with each customer, and set of owners stored with each account
  - We assume all relations are in first normal form
  - (revisited in Chapter 22 of the textbook: Object Based Databases)

CS425 – Fall 2016 – Boris Glavic

8.46

©Silberschatz, Korth and Sudarshan



## First Normal Form (Cont' d)

- Atomicity is actually a property of how the elements of the domain are used.
  - Example: Strings would normally be considered indivisible
  - Suppose that students are given roll numbers which are strings of the form CS0012 or EE1127
  - If the first two characters are extracted to find the department, the domain of roll numbers is not atomic.
  - Doing so is a bad idea: leads to encoding of information in application program rather than in the database.

CS425 – Fall 2016 – Boris Glavic

8.47

©Silberschatz, Korth and Sudarshan



## Second Normal Form

- A relation schema  $R$  in 1NF is in **second normal form (2NF)** iff
  - No non-prime attribute depends on parts of a candidate key
  - An attribute is non-prime if it does not belong to any candidate key for  $R$

CS425 – Fall 2016 – Boris Glavic

8.48

©Silberschatz, Korth and Sudarshan



## Second Normal Form Example

- $R(A,B,C,D)$ 
  - $A,B \rightarrow C,D$
  - $A \rightarrow C$
  - $B \rightarrow D$
- $\{A,B\}$  is the only candidate key
- $R$  is not in 2NF, because  $A \rightarrow C$  where  $A$  is part of a candidate key and  $C$  is not part of a candidate key
- Interpretation  $R(A,B,C,D)$  is **Advisor**(InstrSSN, StudentCWID, InstrName, StudentName)
  - Indication that we are putting stuff together that does not belong together

CS425 – Fall 2016 – Boris Glavic

8.49

©Silberschatz, Korth and Sudarshan



## Second Normal Form Interpretation

- Why is a dependency on parts of a candidate key bad?
  - That is why is a relation that is not in 2NF bad?
- 1) A dependency on part of a candidate key indicates potential for redundancy
  - **Advisor**(InstrSSN, StudentCWID, InstrName, StudentName)
  - StudentCWID  $\rightarrow$  StudentName
  - If a student is advised by multiple instructors we record his name several times
- 2) A dependency on parts of a candidate key shows that some attributes are unrelated to other parts of a candidate key
  - That means the table should be split

CS425 – Fall 2016 – Boris Glavic

8.50

©Silberschatz, Korth and Sudarshan



## 2NF is What We Want?

- **Instructor**(Name, Salary, DepName, DepBudget) =  $I(A,B,C,D)$ 
  - $A \rightarrow B,C,D$
  - $C \rightarrow D$
- {Name} is the only candidate key
- $I$  is in 2NF
- However, as we have seen before  $I$  still has update redundancy that can cause update anomalies
  - We repeat the budget of a department if there is more than one instructor working for that department

CS425 – Fall 2016 – Boris Glavic

8.51

©Silberschatz, Korth and Sudarshan



## Third Normal Form

- A relation schema  $R$  is in **third normal form (3NF)** if for all:
  - $\alpha \rightarrow \beta$  in  $F^R$
 at least one of the following holds:
  - $\alpha \rightarrow \beta$  is trivial (i.e.,  $\beta \in \alpha$ )
  - $\alpha$  is a superkey for  $R$
  - Each attribute  $A$  in  $\beta - \alpha$  is contained in a candidate key for  $R$ .  
(NOTE: each attribute may be in a different candidate key)
- Alternatively,
  - Every attribute depends directly on a candidate key, i.e., for every attribute  $A$  there is a dependency  $X \rightarrow A$ , but no dependency  $Y \rightarrow A$  where  $Y$  is not a candidate key

CS425 – Fall 2016 – Boris Glavic

8.52

©Silberschatz, Korth and Sudarshan



## 3NF Example

- **Instructor**(Name, Salary, DepName, DepBudget) =  $I(A,B,C,D)$ 
  - $A \rightarrow B,C,D$
  - $C \rightarrow D$
- {Name} is the only candidate key
- $I$  is in 2NF
- $I$  is not in 3NF

CS425 – Fall 2016 – Boris Glavic

8.53

©Silberschatz, Korth and Sudarshan



## Testing for 3NF

- Optimization: Need to check only FDs in  $F$ , need not check all FDs in  $F^R$ .
- Use attribute closure to check for each dependency  $\alpha \rightarrow \beta$ , if  $\alpha$  is a superkey.
- If  $\alpha$  is not a superkey, we have to verify if each attribute in  $\beta$  is contained in a candidate key of  $R$ 
  - this test is rather more expensive, since it involve finding candidate keys
  - testing for 3NF has been shown to be NP-hard
  - Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time

CS425 – Fall 2016 – Boris Glavic

8.54

©Silberschatz, Korth and Sudarshan

### 3NF Decomposition Algorithm

```

Let  $F_c$  be a canonical cover for  $F$ ;
 $i := 0$ ;
for each functional dependency  $\alpha \rightarrow \beta$  in  $F_c$  do
  if none of the schemas  $R_i, 1 \leq i \leq i$  contains  $\alpha, \beta$ 
    then begin
       $i := i + 1$ ;
       $R_i := \alpha, \beta$ 
    end
  if none of the schemas  $R_i, 1 \leq i \leq i$  contains a candidate key for  $R$ 
    then begin
       $i := i + 1$ ;
       $R_i :=$  any candidate key for  $R$ ;
    end
  /* Optionally, remove redundant relations */
repeat
  if any schema  $R_j$  is contained in another schema  $R_k$ 
    then /* delete  $R_j$  */
       $R_j = R_k$ ;
       $i := i - 1$ ;
return  $(R_1, R_2, \dots, R_i)$ 

```

CS425 – Fall 2016 – Boris Glavic 8.55 ©Silberschatz, Korth and Sudarshan

### 3NF Decomposition Algorithm (Cont.)

- Above algorithm ensures:
  - each relation schema  $R_i$  is in 3NF
  - decomposition is dependency preserving and lossless-join
  - Proof of correctness is at end of this presentation ([click here](#))

CS425 – Fall 2016 – Boris Glavic 8.56 ©Silberschatz, Korth and Sudarshan

### 3NF Decomposition: An Example

- Relation schema:
  - $cust\_banker\_branch = (customer\_id, employee\_id, branch\_name, type)$
- The functional dependencies for this relation schema are:
  1.  $customer\_id, employee\_id \rightarrow branch\_name, type$
  2.  $employee\_id \rightarrow branch\_name$
  3.  $customer\_id, branch\_name \rightarrow employee\_id$
- We first compute a canonical cover
  - $branch\_name$  is extraneous in the r.h.s. of the 1<sup>st</sup> dependency
  - No other attribute is extraneous, so we get  $F_c =$ 
    - $customer\_id, employee\_id \rightarrow type$
    - $employee\_id \rightarrow branch\_name$
    - $customer\_id, branch\_name \rightarrow employee\_id$

CS425 – Fall 2016 – Boris Glavic 8.57 ©Silberschatz, Korth and Sudarshan

### 3NF Decomposition Example (Cont.)

- The **for** loop generates following 3NF schema:
  - $(customer\_id, employee\_id, type)$
  - $(employee\_id, branch\_name)$
  - $(customer\_id, branch\_name, employee\_id)$
- Observe that  $(customer\_id, employee\_id, type)$  contains a candidate key of the original schema, so no further relation schema needs be added
- At end of **for** loop, detect and delete schemas, such as  $(employee\_id, branch\_name)$ , which are subsets of other schemas
  - result will not depend on the order in which FDs are considered
- The resultant simplified 3NF schema is:
  - $(customer\_id, employee\_id, type)$
  - $(customer\_id, branch\_name, employee\_id)$

CS425 – Fall 2016 – Boris Glavic 8.58 ©Silberschatz, Korth and Sudarshan

### Another 3NF Example

- Relation  $dept\_advisor$ :
  - $dept\_advisor(s\_ID, i\_ID, dept\_name)$
  - $F = \{s\_ID, dept\_name \rightarrow i\_ID, i\_ID \rightarrow dept\_name\}$
  - Two candidate keys:  $s\_ID, dept\_name$ , and  $i\_ID, s\_ID$
  - $R$  is in 3NF
    - $s\_ID, dept\_name \rightarrow i\_ID$ 
      - $dept\_name$  is a superkey
    - $i\_ID \rightarrow dept\_name$ 
      - $dept\_name$  is contained in a candidate key

CS425 – Fall 2016 – Boris Glavic 8.59 ©Silberschatz, Korth and Sudarshan

### Redundancy in 3NF

- There is some redundancy in this schema  $dept\_advisor(s\_ID, i\_ID, dept\_name)$
- Example of problems due to redundancy in 3NF
  - $R = (J, K, L)$
  - $F = \{JK \rightarrow L, L \rightarrow K\}$

J	L	K
$j_1$	$h$	$k_1$
$j_2$	$h$	$k_1$
$j_3$	$h$	$k_1$
null	$l_2$	$k_2$

- repetition of information (e.g., the relationship  $h, k_1$ )
  - $(i\_ID, dept\_name)$
- need to use null values (e.g., to represent the relationship  $l_2, k_2$  where there is no corresponding value for  $J$ ).
- $(i\_ID, dept\_name)$  if there is no separate relation mapping instructors to departments

CS425 – Fall 2016 – Boris Glavic 8.60 ©Silberschatz, Korth and Sudarshan



## Boyce-Codd Normal Form

A relation schema  $R$  is in BCNF with respect to a set  $F$  of functional dependencies if for all functional dependencies in  $F^+$  of the form

$$\alpha \rightarrow \beta$$

where  $\alpha \subseteq R$  and  $\beta \subseteq R$ , at least one of the following holds:

- $\alpha \rightarrow \beta$  is trivial (i.e.,  $\beta \subseteq \alpha$ )
- $\alpha$  is a superkey for  $R$

Example schema *not* in BCNF:

`instr_dept (ID, name, salary, dept_name, building, budget)`

because  $dept\_name \rightarrow building, budget$  holds on `instr_dept`, but  $dept\_name$  is not a superkey

CS425 – Fall 2016 – Boris Glavic

8.61

©Silberschatz, Korth and Sudarshan



## BCNF and Dependency Preservation

- If a relation is in BCNF it is in 3NF
- Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation
- Because it is **not always** possible to achieve **both BCNF and dependency preservation**, we usually consider normally *third normal form*.

CS425 – Fall 2016 – Boris Glavic

8.62

©Silberschatz, Korth and Sudarshan



## Testing for BCNF

- To check if a non-trivial dependency  $\alpha \rightarrow \beta$  causes a violation of BCNF
  1. compute  $\alpha^+$  (the attribute closure of  $\alpha$ ), and
  2. verify that it includes all attributes of  $R$ , that is, it is a superkey of  $R$ .
- **Simplified test:** To check if a relation schema  $R$  is in BCNF, it suffices to check only the dependencies in the given set  $F$  for violation of BCNF, rather than checking all dependencies in  $F^+$ .
  - If none of the dependencies in  $F$  causes a violation of BCNF, then none of the dependencies in  $F^+$  will cause a violation of BCNF either.
- However, **simplified test using only  $F$  is incorrect when testing a relation in a decomposition of  $R$** 
  - Consider  $R = (A, B, C, D, E)$ , with  $F = \{A \rightarrow B, BC \rightarrow D\}$ 
    - ▶ Decompose  $R$  into  $R_1 = (A, B)$  and  $R_2 = (A, C, D, E)$
    - ▶ Neither of the dependencies in  $F$  contain only attributes from  $(A, C, D, E)$  so we might be misled into thinking  $R_2$  satisfies BCNF.
    - ▶ In fact, dependency  $AC \rightarrow D$  in  $F^+$  shows  $R_2$  is not in BCNF.

CS425 – Fall 2016 – Boris Glavic

8.63

©Silberschatz, Korth and Sudarshan



## Testing Decomposition for BCNF

- To check if a relation  $R_i$  in a decomposition of  $R$  is in BCNF,
  - Either test  $R_i$  for BCNF with respect to the **restriction** of  $F$  to  $R_i$  (that is, all FDs in  $F^+$  that contain only attributes from  $R_i$ )
  - or use the original set of dependencies  $F$  that hold on  $R$ , but with the following test:
    - for every set of attributes  $\alpha \subseteq R_i$ , check that  $\alpha^+$  (the attribute closure of  $\alpha$ ) either includes no attribute of  $R - \alpha$ , or includes all attributes of  $R$ .
    - ▶ If the condition is violated by some  $\alpha \rightarrow \beta$  in  $F$ , the dependency  $\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$  can be shown to hold on  $R_i$ , and  $R_i$  violates BCNF.
    - ▶ We use above dependency to decompose  $R_i$

CS425 – Fall 2016 – Boris Glavic

8.64

©Silberschatz, Korth and Sudarshan



## Decomposing a Schema into BCNF

- Suppose we have a schema  $R$  and a non-trivial dependency  $\alpha \rightarrow \beta$  causes a violation of BCNF. We decompose  $R$  into:
  - $(\alpha \cup \beta)$
  - $(R - (\beta - \alpha))$
- In our example,
  - $\alpha = dept\_name$
  - $\beta = building, budget$
  - and `instr_dept` is replaced by
    - $(\alpha \cup \beta) = (dept\_name, building, budget)$
    - $(R - (\beta - \alpha)) = (ID, name, salary, dept\_name)$

CS425 – Fall 2016 – Boris Glavic

8.65

©Silberschatz, Korth and Sudarshan



## BCNF Decomposition Algorithm

```

result := {R};
done := false;
compute F+;
while (not done) do
  if (there is a schema Ri in result that is not in BCNF)
    then begin
      let α → β be a nontrivial functional dependency that
        holds on Ri such that α → Ri is not in F+,
        and α ∩ β = ∅;
      result := (result - Ri) ∪ (Ri - β) ∪ (α, β);
    end
  else done := true;

```

Note: each  $R_i$  is in BCNF, and decomposition is lossless-join.

CS425 – Fall 2016 – Boris Glavic

8.66

©Silberschatz, Korth and Sudarshan

### Example of BCNF Decomposition

- $R = (A, B, C)$   
 $F = \{A \rightarrow B, B \rightarrow C\}$   
 Key =  $\{A\}$
- $R$  is not in BCNF ( $B \rightarrow C$  but  $B$  is not superkey)
- Decomposition
  - $R_1 = (B, C)$
  - $R_2 = (A, B)$

CS425 – Fall 2016 – Boris Glavic 8.67 ©Silberschatz, Korth and Sudarshan

### Example of BCNF Decomposition

- $class(course\_id, title, dept\_name, credits, sec\_id, semester, year, building, room\_number, capacity, time\_slot\_id)$
- Functional dependencies:
  - $course\_id \rightarrow title, dept\_name, credits$
  - $building, room\_number \rightarrow capacity$
  - $course\_id, sec\_id, semester, year \rightarrow building, room\_number, time\_slot\_id$
- A candidate key  $\{course\_id, sec\_id, semester, year\}$ .
- BCNF Decomposition:
  - $course\_id \rightarrow title, dept\_name, credits$  holds but  $course\_id$  is not a superkey.
  - We replace  $class$  by:
    - $course(course\_id, title, dept\_name, credits)$
    - $class-1(course\_id, sec\_id, semester, year, building, room\_number, capacity, time\_slot\_id)$

CS425 – Fall 2016 – Boris Glavic 8.68 ©Silberschatz, Korth and Sudarshan

### BCNF Decomposition (Cont.)

- $course$  is in BCNF
  - How do we know this?
- $building, room\_number \rightarrow capacity$  holds on  $class-1$ 
  - but  $\{building, room\_number\}$  is not a superkey for  $class-1$ .
  - We replace  $class-1$  by:
    - $classroom(building, room\_number, capacity)$
    - $section(course\_id, sec\_id, semester, year, building, room\_number, time\_slot\_id)$
- $classroom$  and  $section$  are in BCNF.

CS425 – Fall 2016 – Boris Glavic 8.69 ©Silberschatz, Korth and Sudarshan

### BCNF and Dependency Preservation

It is not always possible to get a BCNF decomposition that is dependency preserving

- $R = (J, K, L)$   
 $F = \{JK \rightarrow L, L \rightarrow K\}$   
 Two candidate keys =  $JK$  and  $JL$
- $R$  is not in BCNF
- Any decomposition of  $R$  will fail to preserve  $JK \rightarrow L$

This implies that testing for  $JK \rightarrow L$  requires a join

CS425 – Fall 2016 – Boris Glavic 8.70 ©Silberschatz, Korth and Sudarshan

### How good is BCNF?

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a relation
  - $inst\_info(ID, child\_name, phone)$
  - where an instructor may have more than one phone and can have multiple children

ID	child_name	phone
99999	David	512-555-1234
99999	David	512-555-4321
99999	William	512-555-1234
99999	William	512-555-4321


$inst\_info$

CS425 – Fall 2016 – Boris Glavic 8.71 ©Silberschatz, Korth and Sudarshan

### How good is BCNF? (Cont.)

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if we add a phone 981-992-3443 to 99999, we need to add two tuples
  - (99999, David, 981-992-3443)
  - (99999, William, 981-992-3443)

CS425 – Fall 2016 – Boris Glavic 8.72 ©Silberschatz, Korth and Sudarshan



## How good is BCNF? (Cont.)

- Therefore, it is better to decompose *inst\_info* into:


	<i>ID</i>	<i>child_name</i>
<i>inst_child</i>	99999	David
	99999	David
	99999	William
	99999	William

	<i>ID</i>	<i>phone</i>
<i>inst_phone</i>	99999	512-555-1234
	99999	512-555-4321
	99999	512-555-1234
	99999	512-555-4321

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF), which we shall see later.


CS425 – Fall 2016 – Boris Glavic
8.73
©Silberschatz, Korth and Sudarshan



## Comparison of BCNF and 3NF

- It is always possible to decompose a relation into a set of relations that are in 3NF such that:
  - the decomposition is lossless
  - the dependencies are preserved
- It is always possible to decompose a relation into a set of relations that are in BCNF such that:
  - the decomposition is lossless
  - it may not be possible to preserve dependencies.


CS425 – Fall 2016 – Boris Glavic
8.74
©Silberschatz, Korth and Sudarshan



## Summary Normal Forms

- BCNF  $\rightarrow$  3NF  $\rightarrow$  2NF  $\rightarrow$  1NF
- **1NF**
  - atomic attributes
- **2NF**
  - no non-trivial dependencies of non-prime attributes on parts of the key
- **3NF**
  - no transitive non-trivial dependencies on the key
- **BCNF**
  - only non-trivial dependencies on a superkey


CS425 – Fall 2016 – Boris Glavic
8.75
©Silberschatz, Korth and Sudarshan



## Design Goals Revisited

- Goal for a relational database design is:
  - BCNF.
  - Lossless join.
  - Dependency preservation.
- If we cannot achieve this, we accept one of
  - Lack of dependency preservation
  - Redundancy due to use of 3NF
- Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys. Can specify FDs using assertions, but they are expensive to test, (and currently not supported by any of the widely used databases!)
- Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.

CS425 – Fall 2016 – Boris Glavic
8.76
©Silberschatz, Korth and Sudarshan




## Multivalued Dependencies and 4NF, 5NF

modified from:

Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use

CS425 – Fall 2016 – Boris Glavic
8.77
©Silberschatz, Korth and Sudarshan



## Multivalued Dependencies

- Suppose we record names of children, and phone numbers for instructors:
  - *inst\_child*(*ID*, *child\_name*)
  - *inst\_phone*(*ID*, *phone\_number*)
- If we were to combine these schemas to get
  - *inst\_info*(*ID*, *child\_name*, *phone\_number*)
  - Example data:  
(99999, David, 512-555-1234)  
(99999, David, 512-555-4321)  
(99999, William, 512-555-1234)  
(99999, William, 512-555-4321)
- This relation is in BCNF
  - Why?

CS425 – Fall 2016 – Boris Glavic
8.78
©Silberschatz, Korth and Sudarshan



## Multivalued Dependencies (MVDs)

- Let  $R$  be a relation schema and let  $\alpha \subseteq R$  and  $\beta \subseteq R$ . The **multivalued dependency**

$$\alpha \twoheadrightarrow \beta$$
holds on  $R$  if in any legal relation  $r(R)$ , for all pairs of tuples  $t_1$  and  $t_2$  in  $r$  such that  $t_1[\alpha] = t_2[\alpha]$ , there exist tuples  $t_3$  and  $t_4$  in  $r$  such that:
$$\begin{aligned} t_1[\alpha] &= t_2[\alpha] = t_3[\alpha] = t_4[\alpha] \\ t_3[\beta] &= t_1[\beta] \\ t_4[\beta] &= t_2[\beta] \\ t_3[R - \beta] &= t_4[R - \beta] \end{aligned}$$

CS425 – Fall 2016 – Boris Glavic 8.79 ©Silberschatz, Korth and Sudarshan

## MVD (Cont.)

- Tabular representation of  $\alpha \twoheadrightarrow \beta$

	$\alpha$	$\beta$	$R - \alpha - \beta$
$t_1$	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
$t_2$	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
$t_3$	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
$t_4$	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$

CS425 – Fall 2016 – Boris Glavic 8.80 ©Silberschatz, Korth and Sudarshan

## Example

- Let  $R$  be a relation schema with a set of attributes that are partitioned into 3 nonempty subsets.
$$Y, Z, W$$
- We say that  $Y \twoheadrightarrow Z$  ( $Y$  **multidetermines**  $Z$ ) if and only if for all possible relations  $r(R)$ 

$$\langle y_1, z_1, w_1 \rangle \in r \text{ and } \langle y_1, z_2, w_2 \rangle \in r$$
then
$$\langle y_1, z_1, w_2 \rangle \in r \text{ and } \langle y_1, z_2, w_1 \rangle \in r$$
- Note that since the behavior of  $Z$  and  $W$  are identical it follows that  $Y \twoheadrightarrow Z$  if  $Y \twoheadrightarrow W$

CS425 – Fall 2016 – Boris Glavic 8.81 ©Silberschatz, Korth and Sudarshan

## Example (Cont.)

- In our example:
$$\begin{aligned} ID &\twoheadrightarrow \text{child\_name} \\ ID &\twoheadrightarrow \text{phone\_number} \end{aligned}$$
- The above formal definition is supposed to formalize the notion that given a particular value of  $Y$  ( $ID$ ) it has associated with it a set of values of  $Z$  ( $\text{child\_name}$ ) and a set of values of  $W$  ( $\text{phone\_number}$ ), and these two sets are in some sense independent of each other.
- Note:
  - If  $Y \rightarrow Z$  then  $Y \twoheadrightarrow Z$
  - Indeed we have (in above notation)  $Z_1 = Z_2$ . The claim follows.

CS425 – Fall 2016 – Boris Glavic 8.82 ©Silberschatz, Korth and Sudarshan

## Use of Multivalued Dependencies

- We use multivalued dependencies in two ways:
  - To test relations to **determine** whether they are legal under a given set of functional and multivalued dependencies
  - To specify **constraints** on the set of legal relations. We shall thus concern ourselves *only* with relations that satisfy a given set of functional and multivalued dependencies.
- If a relation  $r$  fails to satisfy a given multivalued dependency, we can construct a relations  $r'$  that does satisfy the multivalued dependency by adding tuples to  $r$ .

CS425 – Fall 2016 – Boris Glavic 8.83 ©Silberschatz, Korth and Sudarshan

## Theory of MVDs

- From the definition of multivalued dependency, we can derive the following rule:
  - If  $\alpha \rightarrow \beta$ , then  $\alpha \twoheadrightarrow \beta$
That is, every functional dependency is also a multivalued dependency
- The **closure**  $D^+$  of  $D$  is the set of all functional and multivalued dependencies logically implied by  $D$ .
  - We can compute  $D^+$  from  $D$ , using the formal definitions of functional dependencies and multivalued dependencies.
  - We can manage with such reasoning for very simple multivalued dependencies, which seem to be most common in practice
  - For complex dependencies, it is better to reason about sets of dependencies using a system of inference rules (see Appendix C).

CS425 – Fall 2016 – Boris Glavic 8.84 ©Silberschatz, Korth and Sudarshan



## Fourth Normal Form

- A relation schema  $R$  is in **4NF** with respect to a set  $D$  of functional and multivalued dependencies if for all multivalued dependencies in  $D^*$  of the form  $\alpha \twoheadrightarrow \beta$ , where  $\alpha \subseteq R$  and  $\beta \subseteq R$ , at least one of the following hold:
  - $\alpha \twoheadrightarrow \beta$  is trivial (i.e.,  $\beta \subseteq \alpha$  or  $\alpha \cup \beta = R$ )
  - $\alpha$  is a superkey for schema  $R$
- If a relation is in 4NF it is in BCNF

CS425 – Fall 2016 – Boris Glavic

8.85

©Silberschatz, Korth and Sudarshan



## Restriction of Multivalued Dependencies

- The restriction of  $D$  to  $R_i$  is the set  $D_i$  consisting of
  - All functional dependencies in  $D^*$  that include only attributes of  $R_i$
  - All multivalued dependencies of the form  $\alpha \twoheadrightarrow (\beta \cap R_i)$  where  $\alpha \subseteq R_i$  and  $\alpha \twoheadrightarrow \beta$  is in  $D^*$

CS425 – Fall 2016 – Boris Glavic

8.86

©Silberschatz, Korth and Sudarshan



## 4NF Decomposition Algorithm

```

result := {R};
done := false;
compute D*;
Let Di denote the restriction of D* to Ri
while (not done)
  if (there is a schema Ri in result that is not in 4NF) then
    begin
      let α → β be a nontrivial multivalued dependency that holds
      on Ri such that α → Ri is not in Di, and α ∩ β = ∅;
      result := (result - Ri) ∪ (Ri - β) ∪ (α, β);
    end
  else done := true;
Note: each Ri is in 4NF, and decomposition is lossless-join

```

CS425 – Fall 2016 – Boris Glavic

8.87

©Silberschatz, Korth and Sudarshan



## Example

- $R = (A, B, C, G, H, I)$   
 $F = \{ A \twoheadrightarrow B, B \twoheadrightarrow HI, CG \twoheadrightarrow H \}$
- $R$  is not in 4NF since  $A \twoheadrightarrow B$  and  $A$  is not a superkey for  $R$
- Decomposition
  - a)  $R_1 = (A, B)$  ( $R_1$  is in 4NF)
  - b)  $R_2 = (A, C, G, H, I)$  ( $R_2$  is not in 4NF, decompose into  $R_3$  and  $R_4$ )
  - c)  $R_3 = (C, G, H)$  ( $R_3$  is in 4NF)
  - d)  $R_4 = (A, C, G, I)$  ( $R_4$  is not in 4NF, decompose into  $R_5$  and  $R_6$ )
  - $A \twoheadrightarrow B$  and  $B \twoheadrightarrow HI \rightarrow A \twoheadrightarrow HI$ , (MVD transitivity), and
  - and hence  $A \twoheadrightarrow I$  (MVD restriction to  $R_4$ )
  - e)  $R_5 = (A, I)$  ( $R_5$  is in 4NF)
  - f)  $R_6 = (A, C, G)$  ( $R_6$  is in 4NF)

CS425 – Fall 2016 – Boris Glavic

8.88

©Silberschatz, Korth and Sudarshan



## Further Normal Forms

- **Join dependencies** generalize multivalued dependencies
  - lead to **project-join normal form (PJNF)** (also called **fifth normal form**)
- A class of even more general constraints, leads to a normal form called **domain-key normal form**.
- Problem with these generalized constraints: are hard to reason with, and no set of sound and complete set of inference rules exists.
- Hence rarely used

CS425 – Fall 2016 – Boris Glavic

8.89

©Silberschatz, Korth and Sudarshan



## Final Thoughts on Design Process

modified from:

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

©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## Overall Database Design Process

- We have assumed schema  $R$  is given
  - $R$  could have been generated when converting an ER diagram to a set of tables.
  - $R$  could have been a single relation containing *all* attributes that are of interest (called **universal relation**).
  - Normalization breaks  $R$  into smaller relations.
  - $R$  could have been the result of some ad hoc design of relations, which we then test/convert to normal form.

CS425 – Fall 2016 – Boris Glavic

8.91

©Silberschatz, Korth and Sudarshan



## ER Model and Normalization

- When an ER diagram is carefully designed, identifying all entities correctly, the tables generated from the ER diagram should not need further normalization.
- However, in a real (imperfect) design, there can be functional dependencies from non-key attributes of an entity to other attributes of the entity
  - Example: an *employee* entity with attributes *department\_name* and *building*, and a functional dependency *department\_name* → *building*
  - Good design would have made department an entity
- Functional dependencies from non-key attributes of a relationship set possible, but rare --- most relationships are binary

CS425 – Fall 2016 – Boris Glavic

8.92

©Silberschatz, Korth and Sudarshan



## Denormalization for Performance

- May want to use non-normalized schema for performance
  - For example, displaying *prereqs* along with *course\_id*, and *title* requires join of *course* with *prereq*
- Alternative 1: Use denormalized relation containing attributes of *course* as well as *prereq* with all above attributes
  - faster lookup
  - extra space and extra execution time for updates
  - extra coding work for programmer and possibility of error in extra code
- Alternative 2: use a materialized view defined as *course prereq*
  - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors

CS425 – Fall 2016 – Boris Glavic

8.93

©Silberschatz, Korth and Sudarshan



## Other Design Issues

- Some aspects of database design are not caught by normalization
- Examples of bad database design, to be avoided:
  - Instead of *earnings* (*company\_id*, *year*, *amount*), use *earnings\_2004*, *earnings\_2005*, *earnings\_2006*, etc., all on the schema (*company\_id*, *earnings*).
    - ▶ Above are in BCNF, but make querying across years difficult and needs new table each year
  - *company\_year* (*company\_id*, *earnings\_2004*, *earnings\_2005*, *earnings\_2006*)
    - ▶ Also in BCNF, but also makes querying across years difficult and requires new attribute each year.
    - ▶ Is an example of a **crosstab**, where values for one attribute become column names
    - ▶ Used in spreadsheets, and in data analysis tools

CS425 – Fall 2016 – Boris Glavic

8.94

©Silberschatz, Korth and Sudarshan



## Recap

- Functional and Multi-valued Dependencies
  - Axioms
  - Closure
  - Minimal Cover
  - Attribute Closure
- Redundancy and lossless decomposition
- Normal-Forms
  - 1NF, 2NF, 3NF
  - BCNF
  - 4NF, 5NF

CS425 – Fall 2016 – Boris Glavic

8.95

©Silberschatz, Korth and Sudarshan



## End of Chapter

modified from:

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

©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## Proof of Correctness of 3NF Decomposition Algorithm

modified from:

Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## Correctness of 3NF Decomposition Algorithm

- 3NF decomposition algorithm is dependency preserving (since there is a relation for every FD in  $F_c$ )
- Decomposition is lossless
  - A candidate key ( $C$ ) is in one of the relations  $R_i$  in decomposition
  - Closure of candidate key under  $F_c$  must contain all attributes in  $R$ .
  - Follow the steps of attribute closure algorithm to show there is only one tuple in the join result for each tuple in  $R$

CS425 – Fall 2016 – Boris Glavic

8.98

©Silberschatz, Korth and Sudarshan



## Correctness of 3NF Decomposition Algorithm (Cont' d.)

Claim: if a relation  $R_i$  is in the decomposition generated by the above algorithm, then  $R_i$  satisfies 3NF.

- Let  $R_i$  be generated from the dependency  $\alpha \rightarrow \beta$
- Let  $\gamma \rightarrow B$  be any non-trivial functional dependency on  $R_i$ . (We need only consider FDs whose right-hand side is a single attribute.)
- Now,  $B$  can be in either  $\beta$  or  $\alpha$  but not in both. Consider each case separately.

CS425 – Fall 2016 – Boris Glavic

8.99

©Silberschatz, Korth and Sudarshan



## Correctness of 3NF Decomposition (Cont' d.)

- Case 1: If  $B$  in  $\beta$ :
  - If  $\gamma$  is a superkey, the 2nd condition of 3NF is satisfied
  - Otherwise  $\alpha$  must contain some attribute not in  $\gamma$
  - Since  $\gamma \rightarrow B$  is in  $F^c$  it must be derivable from  $F_c$ , by using attribute closure on  $\gamma$ .
  - Attribute closure not have used  $\alpha \rightarrow \beta$ . If it had been used,  $\alpha$  must be contained in the attribute closure of  $\gamma$ , which is not possible, since we assumed  $\gamma$  is not a superkey.
  - Now, using  $\alpha \rightarrow (\beta - \{B\})$  and  $\gamma \rightarrow B$ , we can derive  $\alpha \rightarrow B$  (since  $\gamma \subseteq \alpha$ , and  $B \notin \gamma$  since  $\gamma \rightarrow B$  is non-trivial)
  - Then,  $B$  is extraneous in the right-hand side of  $\alpha \rightarrow \beta$ ; which is not possible since  $\alpha \rightarrow \beta$  is in  $F_c$ .
  - Thus, if  $B$  is in  $\beta$  then  $\gamma$  must be a superkey, and the second condition of 3NF must be satisfied.

CS425 – Fall 2016 – Boris Glavic

8.100

©Silberschatz, Korth and Sudarshan



## Correctness of 3NF Decomposition (Cont' d.)

- Case 2:  $B$  is in  $\alpha$ .
  - Since  $\alpha$  is a candidate key, the third alternative in the definition of 3NF is trivially satisfied.
  - In fact, we cannot show that  $\gamma$  is a superkey.
  - This shows exactly why the third alternative is present in the definition of 3NF.

Q.E.D.

CS425 – Fall 2016 – Boris Glavic

8.101

©Silberschatz, Korth and Sudarshan



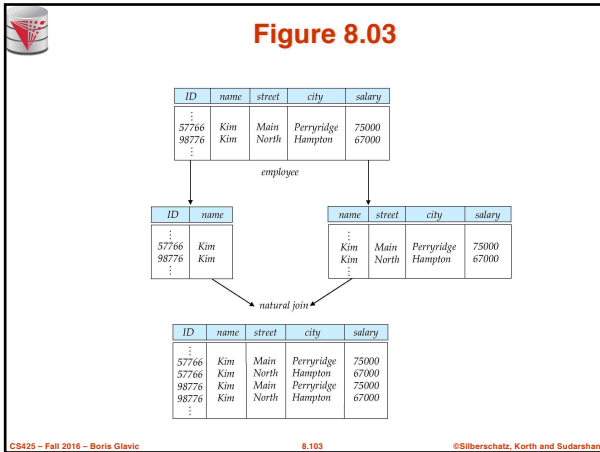
Figure 8.02

ID	name	salary	dept_name	building	budget
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

CS425 – Fall 2016 – Boris Glavic

8.102

©Silberschatz, Korth and Sudarshan



### Figure 8.04

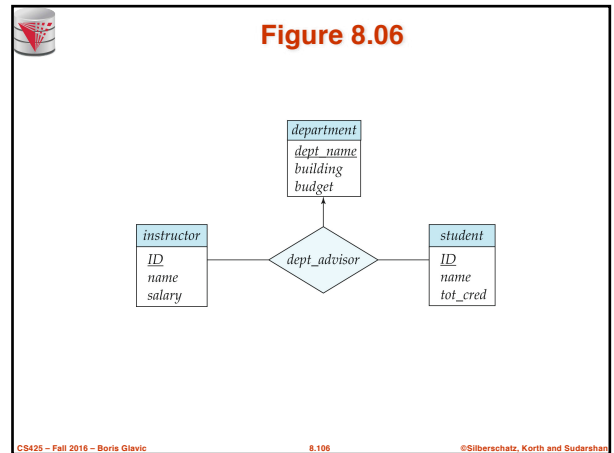
A	B	C	D
a <sub>1</sub>	b <sub>1</sub>	c <sub>1</sub>	d <sub>1</sub>
a <sub>1</sub>	b <sub>2</sub>	c <sub>1</sub>	d <sub>2</sub>
a <sub>2</sub>	b <sub>1</sub>	c <sub>2</sub>	d <sub>2</sub>
a <sub>2</sub>	b <sub>3</sub>	c <sub>2</sub>	d <sub>3</sub>
a <sub>3</sub>	b <sub>3</sub>	c <sub>2</sub>	d <sub>4</sub>

CS425 - Fall 2016 - Boris Glavic 8.104 ©Silberschatz, Korth and Sudarshan

### Figure 8.05

building	room_number	capacity
Packard	101	300
Painter	314	10
Taylor	3128	70
Watson	100	30
Watson	120	50

CS425 - Fall 2016 - Boris Glavic 8.105 ©Silberschatz, Korth and Sudarshan



### Figure 8.14

dept_name	ID	street	city
Physics	22222	North	Rye
Physics	22222	Main	Manchester
Finance	12121	Lake	Horseneck

CS425 - Fall 2016 - Boris Glavic 8.107 ©Silberschatz, Korth and Sudarshan

### Figure 8.15

dept_name	ID	street	city
Physics	22222	North	Rye
Math	22222	Main	Manchester

CS425 - Fall 2016 - Boris Glavic 8.108 ©Silberschatz, Korth and Sudarshan



**Figure 8.17**

A	B	C
$a_1$	$b_1$	$c_1$
$a_1$	$b_1$	$c_2$
$a_2$	$b_1$	$c_1$
$a_2$	$b_1$	$c_3$



## Chapter 9: Transactions

modified from:  
Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## Chapter 9: Transactions

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

CS425 – Fall 2016 – Boris Glavic

9.2

©Silberschatz, Korth and Sudarshan



## Transaction Concept

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

CS425 – Fall 2016 – Boris Glavic

9.3

©Silberschatz, Korth and Sudarshan



## Example of Fund Transfer

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

CS425 – Fall 2016 – Boris Glavic

9.4

©Silberschatz, Korth and Sudarshan



## Example of Fund Transfer (Cont.)

- Transaction to transfer \$50 from account A to account B:
  - `read(A)`
  - `A := A - 50`
  - `write(A)`
  - `read(B)`
  - `B := B + 50`
  - `write(B)`
- Consistency requirement** in above example:
  - the sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g. sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

CS425 – Fall 2016 – Boris Glavic

9.5

©Silberschatz, Korth and Sudarshan



## Example of Fund Transfer (Cont.)

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

<b>T1</b> 1. <code>read(A)</code> 2. <code>A := A - 50</code> 3. <code>write(A)</code>	<b>T2</b>  <code>read(A), read(B), print(A+B)</code>
4. <code>read(B)</code> 5. <code>B := B + 50</code> 6. <code>write(B)</code>	
- Isolation can be ensured trivially by running transactions **serially**
  - that is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

CS425 – Fall 2016 – Boris Glavic

9.6

©Silberschatz, Korth and Sudarshan

## ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions  $T_i$  and  $T_j$ , it appears to  $T_i$  that either  $T_j$  finished execution before  $T_i$  started, or  $T_j$  started execution after  $T_i$  finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

CS425 – Fall 2016 – Boris Glavic 9.7 ©Silberschatz, Korth and Sudarshan

## Transaction State

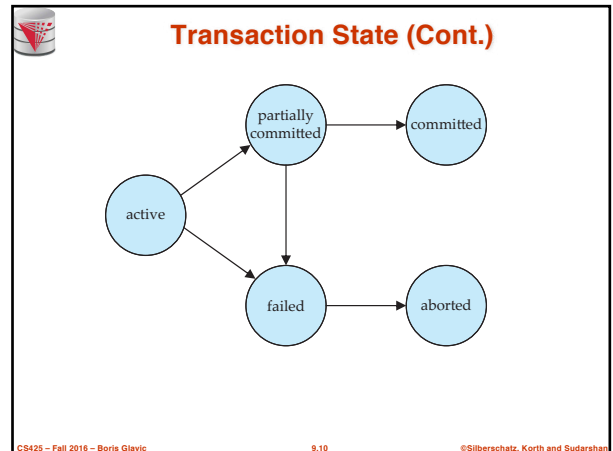
- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** – after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction
    - ↳ can be done only if no internal logical error
  - kill the transaction
- **Committed** – after successful completion.

CS425 – Fall 2016 – Boris Glavic 9.8 ©Silberschatz, Korth and Sudarshan

## Transaction Model

- Operations
  - Read(A) – read value of data item A
  - Write(A) – write a new value of data item A
  - Commit – commit changes of the transaction
  - Abort – Revert changes made by the transaction
- Data Items
  - Objects in the data base
  - Usually we consider tuples (rows) or disk pages

CS425 – Fall 2016 – Boris Glavic 9.9 ©Silberschatz, Korth and Sudarshan



## Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput*
    - ↳ E.g. one transaction can be using the CPU while another is reading from or writing to the disk
    - ↳ In multi-processor systems each statement can use one or more CPUs
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

CS425 – Fall 2016 – Boris Glavic 9.11 ©Silberschatz, Korth and Sudarshan

## Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

CS425 – Fall 2016 – Boris Glavic 9.12 ©Silberschatz, Korth and Sudarshan



### Schedule 1

- Let  $T_1$  transfer \$50 from  $A$  to  $B$ , and  $T_2$  transfer 10% of the balance from  $A$  to  $B$ .
- A serial schedule in which  $T_1$  is followed by  $T_2$ :

$T_1$	$T_2$
<pre>read (A) A := A - 50 write (A) read (B) B := B + 50 write (B) commit</pre>	<pre>read (A) temp := A * 0.1 A := A - temp write (A) read (B) B := B + temp write (B) commit</pre>

CS425 – Fall 2016 – Boris Glavic 9.13 ©Silberschatz, Korth and Sudarshan

### Schedule 2

- A serial schedule where  $T_2$  is followed by  $T_1$

$T_1$	$T_2$
<pre>read (A) A := A - 50 write (A) read (B) B := B + 50 write (B) commit</pre>	<pre>read (A) temp := A * 0.1 A := A - temp write (A) read (B) B := B + temp write (B) commit</pre>

CS425 – Fall 2016 – Boris Glavic 9.14 ©Silberschatz, Korth and Sudarshan

### Schedule 3

- Let  $T_1$  and  $T_2$  be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

$T_1$	$T_2$
<pre>read (A) A := A - 50 write (A)  read (B) B := B + 50 write (B) commit</pre>	<pre>read (A) temp := A * 0.1 A := A - temp write (A)  read (B) B := B + temp write (B) commit</pre>

In Schedules 1, 2 and 3, the sum  $A + B$  is preserved.

CS425 – Fall 2016 – Boris Glavic 9.15 ©Silberschatz, Korth and Sudarshan

### Schedule 4

- The following concurrent schedule does not preserve the value of  $(A + B)$ .

$T_1$	$T_2$
<pre>read (A) A := A - 50  write (A) read (B) B := B + 50 write (B) commit</pre>	<pre>read (A) temp := A * 0.1 A := A - temp write (A) read (B)  B := B + temp write (B) commit</pre>

CS425 – Fall 2016 – Boris Glavic 9.16 ©Silberschatz, Korth and Sudarshan

### Serializability

- Basic Assumption** – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  - conflict serializability
  - view serializability

CS425 – Fall 2016 – Boris Glavic 9.17 ©Silberschatz, Korth and Sudarshan

### Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

CS425 – Fall 2016 – Boris Glavic 9.18 ©Silberschatz, Korth and Sudarshan

## Conflicting Instructions

- Instructions  $l_i$  and  $l_j$  of transactions  $T_i$  and  $T_j$  respectively, **conflict** if and only if there exists some item  $Q$  accessed by both  $l_i$  and  $l_j$ , and at least one of these instructions wrote  $Q$ .
  1.  $l_i = \text{read}(Q)$ ,  $l_j = \text{read}(Q)$ .  $l_i$  and  $l_j$  don't conflict.
  2.  $l_i = \text{read}(Q)$ ,  $l_j = \text{write}(Q)$ . They conflict.
  3.  $l_i = \text{write}(Q)$ ,  $l_j = \text{read}(Q)$ . They conflict.
  4.  $l_i = \text{write}(Q)$ ,  $l_j = \text{write}(Q)$ . They conflict.
- Intuitively, a conflict between  $l_i$  and  $l_j$  forces a (logical) temporal order between them.
  - If  $l_i$  and  $l_j$  are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

CS425 – Fall 2016 – Boris Glavic 9.19 ©Silberschatz, Korth and Sudarshan

## Conflict Serializability

- If a schedule  $S$  can be transformed into a schedule  $S'$  by a series of swaps of non-conflicting instructions, we say that  $S$  and  $S'$  are **conflict equivalent**.
  - That is the order of each pair of conflicting operations in  $S$  and  $S'$  is the same
- We say that a schedule  $S$  is **conflict serializable** if it is conflict equivalent to a serial schedule

CS425 – Fall 2016 – Boris Glavic 9.20 ©Silberschatz, Korth and Sudarshan

## Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where  $T_2$  follows  $T_1$ , by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

$T_1$	$T_2$	$T_1$	$T_2$
read (A) write (A)	read (A) write (A)	read (A) write (A) read (B) write (B)	
read (B) write (B)	read (B) write (B)		read (A) write (A) read (B) write (B)
Schedule 3		Schedule 6	

CS425 – Fall 2016 – Boris Glavic 9.21 ©Silberschatz, Korth and Sudarshan

## Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

$T_3$	$T_4$
read (Q)	
write (Q)	write (Q)

- We are unable to swap instructions in the above schedule to obtain either the serial schedule  $\langle T_3, T_4 \rangle$ , or the serial schedule  $\langle T_4, T_3 \rangle$ .

CS425 – Fall 2016 – Boris Glavic 9.22 ©Silberschatz, Korth and Sudarshan

## View Serializability

- Let  $S$  and  $S'$  be two schedules with the same set of transactions.  $S$  and  $S'$  are **view equivalent** if the following three conditions are met, for each data item  $Q$ .
  1. If in schedule  $S$ , transaction  $T_i$  reads the initial value of  $Q$ , then in schedule  $S'$  also transaction  $T_i$  must read the initial value of  $Q$ .
  2. If in schedule  $S$  transaction  $T_i$  executes **read**( $Q$ ), and that value was produced by transaction  $T_j$  (if any), then in schedule  $S'$  also transaction  $T_i$  must read the value of  $Q$  that was produced by the same **write**( $Q$ ) operation of transaction  $T_j$ .
  3. The transaction (if any) that performs the final **write**( $Q$ ) operation in schedule  $S$  must also perform the final **write**( $Q$ ) operation in schedule  $S'$ .

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

CS425 – Fall 2016 – Boris Glavic 9.23 ©Silberschatz, Korth and Sudarshan

## View Serializability (Cont.)

- A schedule  $S$  is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

$T_{27}$	$T_{28}$	$T_{29}$
read (Q)		
write (Q)	write (Q)	
		write (Q)

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes**.

CS425 – Fall 2016 – Boris Glavic 9.24 ©Silberschatz, Korth and Sudarshan

### Other Notions of Serializability

- The schedule below produces same outcome as the serial schedule  $\langle T_1, T_5 \rangle$ , yet is not conflict equivalent or view equivalent to it.

$T_1$	$T_5$
read (A) $A := A - 50$ write (A)	
	read (B) $B := B - 10$ write (B)
read (B) $B := B + 50$ write (B)	
	read (A) $A := A + 10$ write (A)

- Determining such equivalence requires analysis of operations other than read and write.

CS425 – Fall 2016 – Boris Glavic 9.25 ©Silberschatz, Korth and Sudarshan

### Testing for Serializability

- Consider some schedule of a set of transactions  $T_1, T_2, \dots, T_n$
- Precedence graph** – a directed graph where the vertices are the transactions (names).
- We draw an arc from  $T_i$  to  $T_j$  if the two transaction conflict, and  $T_i$  accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- Example 1**

CS425 – Fall 2016 – Boris Glavic 9.26 ©Silberschatz, Korth and Sudarshan

### Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order  $n^2$  time, where  $n$  is the number of vertices in the graph.
  - (Better algorithms take order  $n + e$  where  $e$  is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a **topological sorting** of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be  $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
  - Are there others?

CS425 – Fall 2016 – Boris Glavic 9.27 ©Silberschatz, Korth and Sudarshan

### Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of NP-complete problems.
  - Thus existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.

CS425 – Fall 2016 – Boris Glavic 9.28 ©Silberschatz, Korth and Sudarshan

### Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- Recoverable schedule** – if a transaction  $T_j$  reads a data item previously written by a transaction  $T_i$ , then the commit operation of  $T_i$  appears before the commit operation of  $T_j$ .
- The following schedule (Schedule 11) is not recoverable if  $T_9$  commits immediately after the read

$T_8$	$T_9$
read (A) write (A)	
	read (A) commit
read (B)	

- If  $T_8$  should abort,  $T_9$  would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

CS425 – Fall 2016 – Boris Glavic 9.29 ©Silberschatz, Korth and Sudarshan

### Cascading Rollbacks

- Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

$T_{10}$	$T_{11}$	$T_{12}$
read (A) read (B) write (A)		
	read (A) write (A)	
abort		read (A)

If  $T_{10}$  fails,  $T_{11}$  and  $T_{12}$  must also be rolled back.

- Can lead to the undoing of a significant amount of work

CS425 – Fall 2016 – Boris Glavic 9.30 ©Silberschatz, Korth and Sudarshan



## Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur; for each pair of transactions  $T_i$  and  $T_j$  such that  $T_j$  reads a data item previously written by  $T_i$ , the commit operation of  $T_i$  appears before the read operation of  $T_j$ .
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

CS425 – Fall 2016 – Boris Glavic

9.31

©Silberschatz, Korth and Sudarshan



## Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.

CS425 – Fall 2016 – Boris Glavic

9.32

©Silberschatz, Korth and Sudarshan



## Concurrency Control (Cont.)

- Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

CS425 – Fall 2016 – Boris Glavic

9.33

©Silberschatz, Korth and Sudarshan



## Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids nonserializable schedules.
  - We study such protocols in Chapter 10.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.

CS425 – Fall 2016 – Boris Glavic

9.34

©Silberschatz, Korth and Sudarshan



## Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g. a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g. database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

CS425 – Fall 2016 – Boris Glavic

9.35

©Silberschatz, Korth and Sudarshan



## Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable — it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read, but successive reads of a record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.
- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
  - E.g. Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)

CS425 – Fall 2016 – Boris Glavic

9.36

©Silberschatz, Korth and Sudarshan



## Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g. in JDBC, `connection.setAutoCommit(false);`



## End of Chapter 10

modified from:  
 Database System Concepts, 6<sup>th</sup> Ed.  
 ©Silberschatz, Korth and Sudarshan  
 See [www.db-book.com](http://www.db-book.com) for conditions on re-use

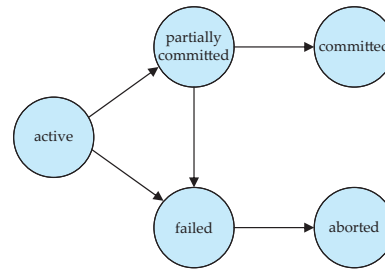


## Recap

- Transactions
  - ACID – Properties
- Schedules
  - Serial
  - Equivalence
    - Conflict-equivalent
    - View-equivalent
  - Serializability
    - = Equivalent to a serial schedule
  - Recoverable
  - Cascading Aborts
- Transactions in SQL



### Figure 14.01



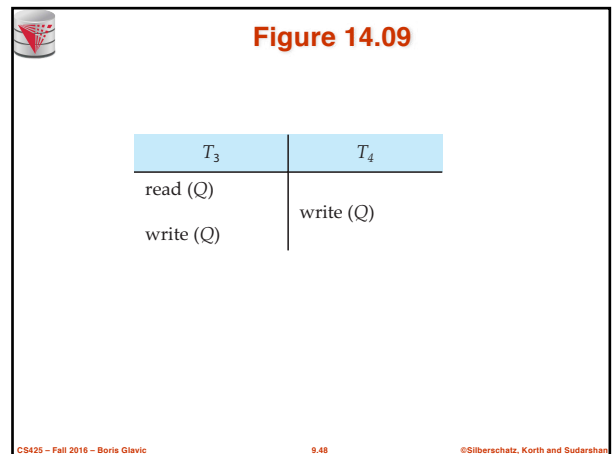
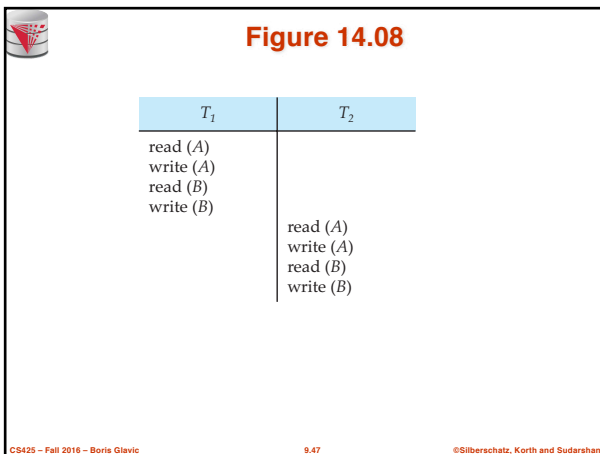
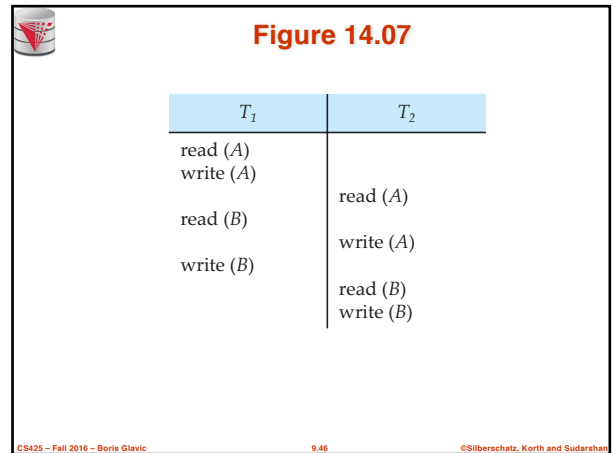
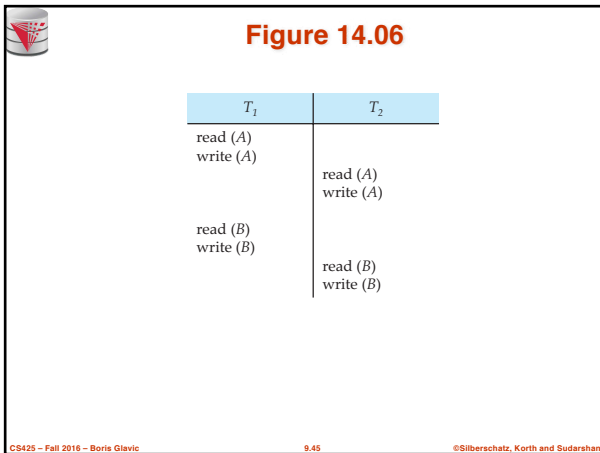
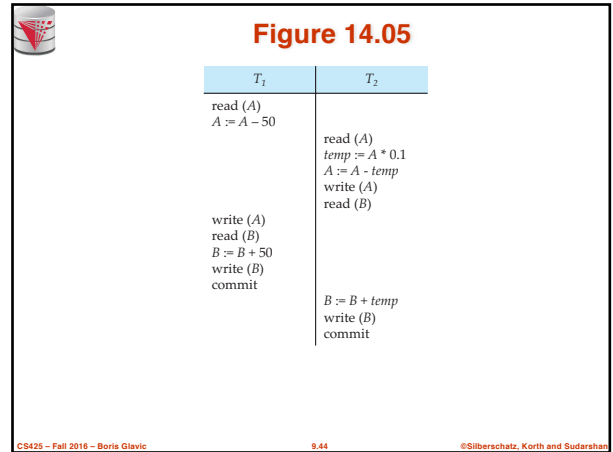
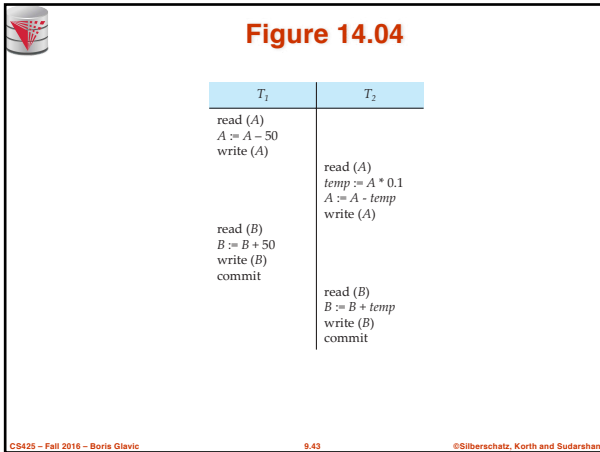
### Figure 14.02

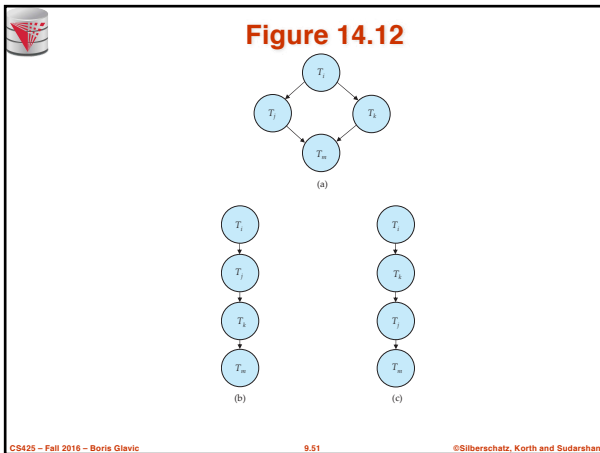
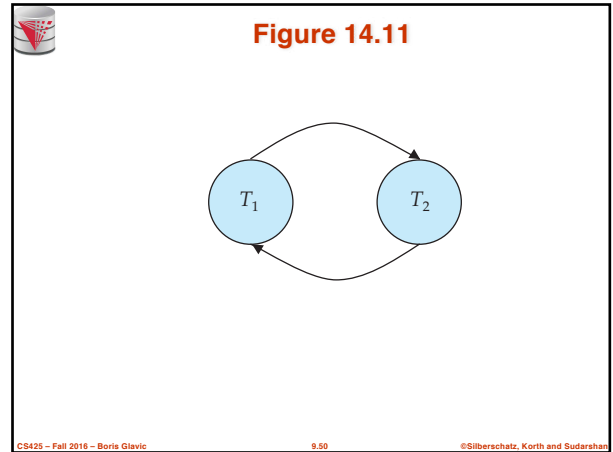
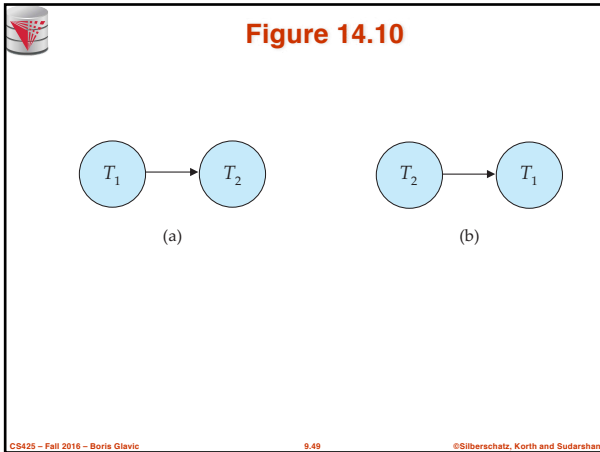
$T_1$	$T_2$
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit



### Figure 14.03

$T_1$	$T_2$
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit





### Figure 14.13

$T_1$	$T_2$
read (A) $A := A - 50$ write (A)	
	read (B) $B := B - 10$ write (B)
read (B) $B := B + 50$ write (B)	
	read (A) $A := A + 10$ write (A)

CS425 – Fall 2016 – Boris Glavic 9.52 ©Silberschatz, Korth and Sudarshan

### Figure 14.14

$T_8$	$T_9$
read (A) write (A)	
	read (A) commit
read (B)	

CS425 – Fall 2016 – Boris Glavic 9.53 ©Silberschatz, Korth and Sudarshan

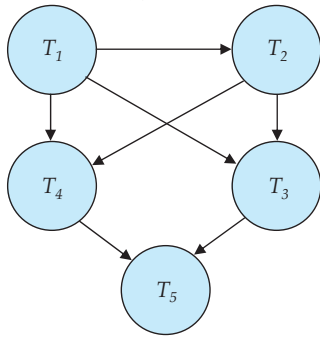
### Figure 14.15

$T_{10}$	$T_{11}$	$T_{12}$
read (A) read (B) write (A)		
	read (A) write (A)	
abort		read (A)

CS425 – Fall 2016 – Boris Glavic 9.54 ©Silberschatz, Korth and Sudarshan



**Figure 14.16**







## Chapter 10 : Concurrency Control

modified from:  
Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## Chapter 10: Concurrency Control

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiple Granularity
- Multiversion Schemes
- Insert and Delete Operations
- Concurrency in Index Structures

CS425 – Fall 2016 – Boris Glavic

10.2

©Silberschatz, Korth and Sudarshan



## Intuition of Lock-based Protocols

- Transactions have to acquire locks on data items before accessing them
- If a lock is hold by one transaction on a data item this restricts the ability of other transactions to acquire locks for that data item
- By locking a data item we want to ensure that no access to that data item is possible that would lead to non-serializable schedules
- The trick is to design a lock model and protocol that guarantees that
- Lock-based concurrency protocols are a form of **pessimistic concurrency control mechanism**
  - We avoid ever getting into a state that can lead to a non-serializable schedule
- Alternative concurrency control mechanism do not avoid conflicts, but determine later on (at commit time) whether committing a transaction would cause a non-serializable schedule to be generated
  - **Optimistic concurrency control mechanism**

CS425 – Fall 2016 – Boris Glavic

10.3

©Silberschatz, Korth and Sudarshan



## Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. **exclusive (X) mode**. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. **shared (S) mode**. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to concurrency-control manager.
  - Transaction do not access data items before having acquired a lock on that data item
  - Transactions release their locks on a data item only after they have accessed a data item

CS425 – Fall 2016 – Boris Glavic

10.4

©Silberschatz, Korth and Sudarshan



## Lock-Based Protocols (Cont.)

- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
  - but if any transaction holds an exclusive lock on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

CS425 – Fall 2016 – Boris Glavic

10.5

©Silberschatz, Korth and Sudarshan



## Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

```

Tz: lock-S(A);
    read (A);
    unlock(A);
    lock-S(B);
    read (B);
    unlock(B);
    display(A+B)
  
```

- Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

CS425 – Fall 2016 – Boris Glavic

10.6

©Silberschatz, Korth and Sudarshan



## Pitfalls of Lock-Based Protocols

- Consider the partial schedule

$T_3$	$T_4$
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

- Neither  $T_3$  nor  $T_4$  can make progress — executing **lock-S(B)** causes  $T_4$  to wait for  $T_3$  to release its lock on B, while executing **lock-X(A)** causes  $T_3$  to wait for  $T_4$  to release its lock on A.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of  $T_3$  or  $T_4$  must be rolled back and its locks released.

CS425 – Fall 2016 – Boris Glavic

10.7

©Silberschatz, Korth and Sudarshan



## Pitfalls of Lock-Based Protocols (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- Starvation** is also possible if the concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control managers can be designed to prevent starvation.

CS425 – Fall 2016 – Boris Glavic

10.8

©Silberschatz, Korth and Sudarshan



## The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

CS425 – Fall 2016 – Boris Glavic

10.9

©Silberschatz, Korth and Sudarshan



## The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking (S2PL)**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- Rigorous two-phase locking (SS2PL)** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

CS425 – Fall 2016 – Boris Glavic

10.10

©Silberschatz, Korth and Sudarshan



## The Two-Phase Locking Protocol (Cont.)

- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:  
Given a transaction  $T_i$  that does not follow two-phase locking, we can find a transaction  $T_j$  that uses two-phase locking, and a schedule for  $T_i$  and  $T_j$  that is not conflict serializable.

CS425 – Fall 2016 – Boris Glavic

10.11

©Silberschatz, Korth and Sudarshan



## Lock Conversions

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

CS425 – Fall 2016 – Boris Glavic

10.12

©Silberschatz, Korth and Sudarshan

## Automatic Acquisition of Locks

- A transaction  $T_i$  issues the standard read/write instruction, without explicit locking calls.
- The operation  $\text{read}(D)$  is processed as:
 

```

      if  $T_i$  has a lock on  $D$ 
      then
        read( $D$ )
      else begin
        if necessary wait until no other
        transaction has a lock-X on  $D$ 
        grant  $T_i$  a lock-S on  $D$ ;
        read( $D$ )
      end
      
```

CS425 – Fall 2016 – Boris Glavic 10.13 ©Silberschatz, Korth and Sudarshan

## Automatic Acquisition of Locks (Cont.)

- $\text{write}(D)$  is processed as:
 

```

      if  $T_i$  has a lock-X on  $D$ 
      then
        write( $D$ )
      else begin
        if necessary wait until no other trans. has any lock on  $D$ ,
        if  $T_i$  has a lock-S on  $D$ 
        then
          upgrade lock on  $D$  to lock-X
        else
          grant  $T_i$  a lock-X on  $D$ 
        write( $D$ )
      end;
      
```
- All locks are released after commit or abort

CS425 – Fall 2016 – Boris Glavic 10.14 ©Silberschatz, Korth and Sudarshan

## Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

CS425 – Fall 2016 – Boris Glavic 10.15 ©Silberschatz, Korth and Sudarshan

## Lock Table

- Black rectangles indicate granted locks, white ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

CS425 – Fall 2016 – Boris Glavic 10.16 ©Silberschatz, Korth and Sudarshan

## Deadlock Handling

- Consider the following two transactions:
 

```

      T1: write(X)   T2: write(Y)
        write(Y)   write(X)
      
```
- Schedule with deadlock
 

$T_1$	$T_2$
lock-X on A write(A)	
	lock-X on B write(B)
wait for lock-X on B	wait for lock-X on A

CS425 – Fall 2016 – Boris Glavic 10.17 ©Silberschatz, Korth and Sudarshan

## Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- Deadlock prevention** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration).
    - Not practical
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

CS425 – Fall 2016 – Boris Glavic 10.18 ©Silberschatz, Korth and Sudarshan



## More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.
  - Preemptive:** Transaction holding a lock is aborted to make lock available
- wait-die** scheme — non-preemptive
  - older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item
- wound-wait** scheme — preemptive
  - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - may be fewer rollbacks than *wait-die* scheme.

CS425 – Fall 2016 – Boris Glavic

10.19

©Silberschatz, Korth and Sudarshan



## Deadlock prevention (Cont.)

- Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.
- Timeout-Based Schemes:**
  - a transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - thus deadlocks are not possible
  - simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

CS425 – Fall 2016 – Boris Glavic

10.20

©Silberschatz, Korth and Sudarshan



## Deadlock Detection

- Deadlocks can be described as a *wait-for graph*, which consists of a pair  $G = (V, E)$ .
  - $V$  is a set of vertices (all the transactions in the system)
  - $E$  is a set of edges; each element is an ordered pair  $T_i \rightarrow T_j$ .
- If  $T_i \rightarrow T_j$  is in  $E$ , then there is a directed edge from  $T_i$  to  $T_j$ , implying that  $T_i$  is waiting for  $T_j$  to release a data item.
- When  $T_i$  requests a data item currently being held by  $T_j$ , then the edge  $T_i \rightarrow T_j$  is inserted in the wait-for graph. This edge is removed only when  $T_j$  is no longer holding a data item needed by  $T_i$ .
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

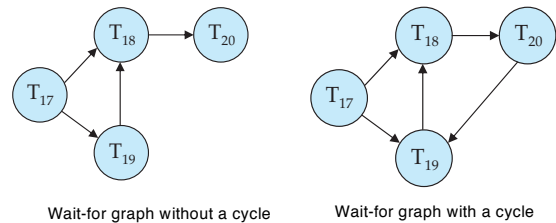
CS425 – Fall 2016 – Boris Glavic

10.21

©Silberschatz, Korth and Sudarshan



## Deadlock Detection (Cont.)



CS425 – Fall 2016 – Boris Glavic

10.22

©Silberschatz, Korth and Sudarshan



## Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to be rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
  - Rollback -- determine how far to roll back transaction
    - Total rollback:** Abort the transaction and then restart it.
      - More effective to roll back transaction only as far as necessary to break deadlock.
  - Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

CS425 – Fall 2016 – Boris Glavic

10.23

©Silberschatz, Korth and Sudarshan



## Weak Levels of Consistency

- Degree-two consistency:** differs from two-phase locking in that S-locks may be released at any time, and locks may be acquired at any time
  - X-locks must be held till end of transaction
  - Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur]
- Cursor stability:**
  - For reads, each tuple is locked, read, and lock is immediately released
  - X-locks are held till end of transaction
  - Special case of degree-two consistency

CS425 – Fall 2016 – Boris Glavic

10.24

©Silberschatz, Korth and Sudarshan

## Weak Levels of Consistency in SQL

- SQL allows non-serializable executions
  - Serializable**: is the default
  - Repeatable read**: allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained)
    - However, the phantom phenomenon need not be prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - Read committed**: same as degree two consistency, but most systems implement it as cursor-stability
  - Read uncommitted**: allows even uncommitted data to be read
- In many database systems, read committed is the default consistency level
  - has to be explicitly changed to serializable when required
    - set isolation level serializable**

CS425 – Fall 2016 – Boris Glavic 10.25 ©Silberschatz, Korth and Sudarshan

## Recap

- Concurrency Control**
  - Pessimistic**: Prevent bad things from happening
    - Locking Protocols
  - Optimistic**: Detect that bad things have happened and resolve the problem
- Two-Phase Locking (2PL)**
  - Two types of locks:
    - Shared (S) locks for read-only access
    - Exclusive (X) locks for write + read access
  - Lock compatibility
  - Transactions cannot acquire locks after they have released a lock
    - Divides transaction into growing and shrinking phase
  - Ensures conflict-serializability**
  - Cascading rollbacks are possible**
  - Deadlocks are possible**

CS425 – Fall 2016 – Boris Glavic 10.26 ©Silberschatz, Korth and Sudarshan

## Recap

- Strict Two-Phase Locking (S2PL)**
  - Exclusive locks are held until transaction commit
  - Prevents cascading rollbacks**
  - Deadlocks are still possible**
- Strict Strong Two-Phase Locking (SS2PL)**
  - All locks are held until transaction commit
  - Enables serializability in commit order**
- Deadlocks**
  - Deadlock Prevention**
    - Wait-die**: Younger transaction that waits for older is rolled back
    - Wound-wait**: If older waits for younger, then younger is rolled back
  - Deadlock Detection**
    - Cycle Detection in Waits-for graph
      - Expensive
    - Timeout

CS425 – Fall 2016 – Boris Glavic 10.27 ©Silberschatz, Korth and Sudarshan

## End of Chapter

Thanks to Alan Fekete and Sudhir Jorwekar for Snapshot Isolation examples

modified from:  
**Database System Concepts, 6<sup>th</sup> Ed.**  
 ©Silberschatz, Korth and Sudarshan  
 See [www.db-book.com](http://www.db-book.com) for conditions on re-use

## Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendants in the same mode.
- Granularity of locking** (level in tree where locking is done):
  - fine granularity** (lower in tree): high concurrency, high locking overhead
  - coarse granularity** (higher in tree): low locking overhead, low concurrency

CS425 – Fall 2016 – Boris Glavic 10.29 ©Silberschatz, Korth and Sudarshan

## Example of Granularity Hierarchy

```

graph TD
    DB --> A1
    DB --> A2
    A1 --> F1
    A1 --> F2
    A2 --> F3
    F1 --> r11
    F1 --> r12
    F1 --> r1n
    F2 --> r21
    F2 --> r2k
    F3 --> r31
    F3 --> r3m
  
```

The levels, starting from the coarsest (top) level are

- database
- area
- file
- record

CS425 – Fall 2016 – Boris Glavic 10.30 ©Silberschatz, Korth and Sudarshan

## Intention Lock Modes

- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  - **intention-shared (IS)**: indicates explicit locking at a lower level of the tree but only with shared locks.
  - **intention-exclusive (IX)**: indicates explicit locking at a lower level with exclusive or shared locks
  - **shared and intention-exclusive (SIX)**: the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
- intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

CS425 – Fall 2016 – Boris Glavic
10.31
©Silberschatz, Korth and Sudarshan

## Compatibility Matrix with Intention Lock Modes

- The compatibility matrix for all lock modes is:

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

CS425 – Fall 2016 – Boris Glavic
10.32
©Silberschatz, Korth and Sudarshan

## Multiple Granularity Locking Scheme

- Transaction  $T_i$  can lock a node  $Q$ , using the following rules:
  1. The lock compatibility matrix must be observed.
  2. The root of the tree must be locked first, and may be locked in any mode.
  3. A node  $Q$  can be locked by  $T_i$  in S or IS mode only if the parent of  $Q$  is currently locked by  $T_i$  in either IX or IS mode.
  4. A node  $Q$  can be locked by  $T_i$  in X, SIX, or IX mode only if the parent of  $Q$  is currently locked by  $T_i$  in either IX or SIX mode.
  5.  $T_i$  can lock a node only if it has not previously unlocked any node (that is,  $T_i$  is two-phase).
  6.  $T_i$  can unlock a node  $Q$  only if none of the children of  $Q$  are currently locked by  $T_i$ .
- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.
- **Lock granularity escalation**: in case there are too many locks at a particular level, switch to higher granularity S or X lock

CS425 – Fall 2016 – Boris Glavic
10.33
©Silberschatz, Korth and Sudarshan

## Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction  $T_i$  has time-stamp  $TS(T_i)$ , a new transaction  $T_j$  is assigned time-stamp  $TS(T_j)$  such that  $TS(T_j) < TS(T_i)$ .
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data  $Q$  two timestamp values:
  - **W-timestamp( $Q$ )** is the largest time-stamp of any transaction that executed **write( $Q$ )** successfully.
  - **R-timestamp( $Q$ )** is the largest time-stamp of any transaction that executed **read( $Q$ )** successfully.

CS425 – Fall 2016 – Boris Glavic
10.34
©Silberschatz, Korth and Sudarshan

## Timestamp-Based Protocols (Cont.)

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction  $T_i$  issues a **read( $Q$ )**
  1. If  $TS(T_i) \leq W\text{-timestamp}(Q)$ , then  $T_i$  needs to read a value of  $Q$  that was already overwritten.
    - Hence, the **read** operation is rejected, and  $T_i$  is rolled back.
  2. If  $TS(T_i) \geq W\text{-timestamp}(Q)$ , then the **read** operation is executed, and R-timestamp( $Q$ ) is set to  $\max(R\text{-timestamp}(Q), TS(T_i))$ .

CS425 – Fall 2016 – Boris Glavic
10.35
©Silberschatz, Korth and Sudarshan

## Timestamp-Based Protocols (Cont.)

- Suppose that transaction  $T_i$  issues **write( $Q$ )**.
  1. If  $TS(T_i) < R\text{-timestamp}(Q)$ , then the value of  $Q$  that  $T_i$  is producing was needed previously, and the system assumed that that value would never be produced.
    - Hence, the **write** operation is rejected, and  $T_i$  is rolled back.
  2. If  $TS(T_i) < W\text{-timestamp}(Q)$ , then  $T_i$  is attempting to write an obsolete value of  $Q$ .
    - Hence, this **write** operation is rejected, and  $T_i$  is rolled back.
  3. Otherwise, the **write** operation is executed, and W-timestamp( $Q$ ) is set to  $TS(T_i)$ .

CS425 – Fall 2016 – Boris Glavic
10.36
©Silberschatz, Korth and Sudarshan

## Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

	$T_1$	$T_2$	$T_3$	$T_4$	$T_5$
		read (Y)			read (X)
read (Y)			write (Y) write (Z)		
		read (Z) abort			read (Z)
read (X)			write (W) abort	read (W)	
					write (Y) write (Z)

CS425 – Fall 2016 – Boris Glavic 10.37 ©Silberschatz, Korth and Sudarshan

## Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.

CS425 – Fall 2016 – Boris Glavic 10.38 ©Silberschatz, Korth and Sudarshan

## Recoverability and Cascade Freedom

- Problem with timestamp-ordering protocol:
  - Suppose  $T_i$  aborts, but  $T_j$  has read a data item written by  $T_i$
  - Then  $T_j$  must abort; if  $T_j$  had been allowed to commit earlier, the schedule is not recoverable.
  - Further, any transaction that has read a data item written by  $T_j$  must abort
  - This can lead to cascading rollback --- that is, a chain of rollbacks
- Solution 1:
  - A transaction is structured such that its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp
- Solution 2: Limited form of locking: wait for data to be committed before reading it
- Solution 3: Use commit dependencies to ensure recoverability

CS425 – Fall 2016 – Boris Glavic 10.39 ©Silberschatz, Korth and Sudarshan

## Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete write operations may be ignored under certain circumstances.
- When  $T_i$  attempts to write data item  $Q$ , if  $TS(T_i) < W\text{-timestamp}(Q)$ , then  $T_i$  is attempting to write an obsolete value of  $Q$ .
  - Rather than rolling back  $T_i$  as the timestamp ordering protocol would have done, this {write} operation can be ignored.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency.
  - Allows some view-serializable schedules that are not conflict-serializable.

CS425 – Fall 2016 – Boris Glavic 10.40 ©Silberschatz, Korth and Sudarshan

## Validation-Based Protocol

- Execution of transaction  $T_i$  is done in three phases.
  - Read and execution phase:** Transaction  $T_i$  writes only to temporary local variables
  - Validation phase:** Transaction  $T_i$  performs a "validation test" to determine if local variables can be written without violating serializability.
  - Write phase:** If  $T_i$  is validated, the updates are applied to the database; otherwise,  $T_i$  is rolled back.
- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
  - Assume for simplicity that the validation and write phase occur together, atomically and serially
    - I.e., only one transaction executes validation/write at a time.
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation

CS425 – Fall 2016 – Boris Glavic 10.41 ©Silberschatz, Korth and Sudarshan

## Validation-Based Protocol (Cont.)

- Each transaction  $T_i$  has 3 timestamps
  - $Start(T_i)$  : the time when  $T_i$  started its execution
  - $Validation(T_i)$ : the time when  $T_i$  entered its validation phase
  - $Finish(T_i)$  : the time when  $T_i$  finished its write phase
- Serializability order is determined by timestamp given at validation time, to increase concurrency.
  - Thus  $TS(T_i)$  is given the value of  $Validation(T_i)$ .
- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.
  - because the serializability order is not pre-decided, and
  - relatively few transactions will have to be rolled back.

CS425 – Fall 2016 – Boris Glavic 10.42 ©Silberschatz, Korth and Sudarshan

## Validation Test for Transaction $T_j$

- If for all  $T_i$  with  $TS(T_i) < TS(T_j)$  either one of the following condition holds:
  - $finish(T_i) < start(T_j)$
  - $start(T_i) < finish(T_j) < validation(T_j)$  and the set of data items written by  $T_i$  does not intersect with the set of data items read by  $T_j$ .

then validation succeeds and  $T_j$  can be committed. Otherwise, validation fails and  $T_j$  is aborted.

- **Justification:** Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
  - the writes of  $T_j$  do not affect reads of  $T_i$  since they occur after  $T_i$  has finished its reads.
  - the writes of  $T_i$  do not affect reads of  $T_j$  since  $T_j$  does not read any item written by  $T_i$ .

CS425 – Fall 2016 – Boris Glavic 10.43 ©Silberschatz, Korth and Sudarshan

## Schedule Produced by Validation

- Example of schedule produced using validation

$T_{35}$	$T_{36}$
read (B)	read (B) $B := B - 50$ read (A) $A := A + 50$
read (A) (validate) display (A + B)	(validate) write (B) write (A)

CS425 – Fall 2016 – Boris Glavic 10.44 ©Silberschatz, Korth and Sudarshan

## Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful **write** results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a **read(Q)** operation is issued, select an appropriate version of  $Q$  based on the timestamp of the transaction, and return the value of the selected version.
- **reads** never have to wait as an appropriate version is returned immediately.

CS425 – Fall 2016 – Boris Glavic 10.45 ©Silberschatz, Korth and Sudarshan

## Multiversion Timestamp Ordering

- Each data item  $Q$  has a sequence of versions  $\langle Q_1, Q_2, \dots, Q_m \rangle$ . Each version  $Q_k$  contains three data fields:
  - **Content** -- the value of version  $Q_k$ .
  - **W-timestamp** ( $Q_k$ ) -- timestamp of the transaction that created (wrote) version  $Q_k$
  - **R-timestamp** ( $Q_k$ ) -- largest timestamp of a transaction that successfully read version  $Q_k$ .
- when a transaction  $T_i$  creates a new version  $Q_k$  of  $Q$ ,  $Q_k$ 's W-timestamp and R-timestamp are initialized to  $TS(T_i)$ .
- R-timestamp of  $Q_k$  is updated whenever a transaction  $T_j$  reads  $Q_k$ , and  $TS(T_j) > R\text{-timestamp}(Q_k)$ .

CS425 – Fall 2016 – Boris Glavic 10.46 ©Silberschatz, Korth and Sudarshan

## Multiversion Timestamp Ordering (Cont)

- Suppose that transaction  $T_i$  issues a **read(Q)** or **write(Q)** operation. Let  $Q_k$  denote the version of  $Q$  whose write timestamp is the largest write timestamp less than or equal to  $TS(T_i)$ .
  1. If transaction  $T_i$  issues a **read(Q)**, then the value returned is the content of version  $Q_k$ .
  2. If transaction  $T_i$  issues a **write(Q)**
    1. if  $TS(T_i) < R\text{-timestamp}(Q_k)$ , then transaction  $T_i$  is rolled back.
    2. if  $TS(T_i) = W\text{-timestamp}(Q_k)$ , the contents of  $Q_k$  are overwritten
    3. else a new version of  $Q$  is created.
- Observe that
  - Reads always succeed
  - A write by  $T_i$  is rejected if some other transaction  $T_j$  that (in the serialization order defined by the timestamp values) should read  $T_i$ 's write, has already read a version created by a transaction older than  $T_i$ .
- Protocol guarantees serializability

CS425 – Fall 2016 – Boris Glavic 10.47 ©Silberschatz, Korth and Sudarshan

## Multiversion Two-Phase Locking

- Differentiates between read-only transactions and update transactions
- **Update transactions** acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Each successful **write** results in the creation of a new version of the data item written.
  - each version of a data item has a single timestamp whose value is obtained from a counter **ts-counter** that is incremented during commit processing.
- **Read-only transactions** are assigned a timestamp by reading the current value of **ts-counter** before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.

CS425 – Fall 2016 – Boris Glavic 10.48 ©Silberschatz, Korth and Sudarshan



## Multiversion Two-Phase Locking (Cont.)

- When an update transaction wants to read a data item:
  - it obtains a shared lock on it, and reads the latest version.
- When it wants to write an item
  - it obtains X lock on it; it then creates a new version of the item and sets this version's timestamp to  $\infty$ .
- When update transaction  $T_i$  completes, commit processing occurs:
  - $T_i$  sets timestamp on the versions it has created to **ts-counter** + 1
  - $T_i$  increments **ts-counter** by 1
- Read-only transactions that start after  $T_i$  increments **ts-counter** will see the values updated by  $T_i$ .
- Read-only transactions that start before  $T_i$  increments the **ts-counter** will see the value before the updates by  $T_i$ .
- Only serializable schedules are produced.

CS425 – Fall 2016 – Boris Glavic 10.49 ©Silberschatz, Korth and Sudarshan

## MVCC: Implementation Issues

- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - E.g. if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, then Q5 will never be required again

CS425 – Fall 2016 – Boris Glavic 10.50 ©Silberschatz, Korth and Sudarshan

## Snapshot Isolation

- Motivation: Decision support queries that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows
  - Poor performance results
- Solution 1: Give logical “snapshot” of database state to read only transactions, read-write transactions use normal locking
  - Multiversion 2-phase locking
  - Works well, but how does system know a transaction is read only?
- Solution 2: Give snapshot of database state to every transaction, updates alone use 2-phase locking to guard against concurrent updates
  - Problem: variety of anomalies such as lost update can result
  - Partial solution: snapshot isolation level (next slide)
    - Proposed by Berenson et al, SIGMOD 1995
    - Variants implemented in many database systems
      - E.g. Oracle, PostgreSQL, SQL Server 2005

CS425 – Fall 2016 – Boris Glavic 10.51 ©Silberschatz, Korth and Sudarshan

## Snapshot Isolation

- A transaction T1 executing with Snapshot Isolation
  - takes snapshot of committed data at start
  - always reads/modifies data in its own snapshot
  - updates of concurrent transactions are not visible to T1
  - writes of T1 complete when it commits
  - First-committer-wins rule:**
    - Commits only if no other concurrent transaction has already written data that T1 intends to write.

	T1	T2	T3
	W(Y := 1) Commit		
		Start R(X) → 0 R(Y) → 1	
			W(X:=2) W(Z:=3) Commit
		R(Z) → 1 R(Y) → 1 W(X:=3) Commit-Req Abort	

Concurrent updates not visible  
Own updates are visible  
Not first-committer of X  
Serialization error, T2 is rolled back

CS425 – Fall 2016 – Boris Glavic 10.52 ©Silberschatz, Korth and Sudarshan

## Snapshot Read

- Concurrent updates invisible to snapshot read

$X_0 = 100, Y_0 = 0$

<p><math>T_1</math> deposits 50 in Y</p> <p><math>r_1(X_0, 100)</math> <math>r_1(Y_0, 0)</math></p> <p><math>w_1(Y_1, 50)</math> <math>r_1(X_0, 100)</math> (update by <math>T_2</math> not seen) <math>r_1(Y_1, 50)</math> (can see its own updates)</p> <p><math>X_2 = 50, Y_1 = 50</math></p>	<p><math>T_2</math> withdraws 50 from X</p> <p><math>r_2(Y_0, 0)</math> <math>r_2(X_0, 100)</math> <math>w_2(X_2, 50)</math></p> <p><math>r_2(Y_0, 0)</math> (update by <math>T_1</math> not seen)</p>
--	--

CS425 – Fall 2016 – Boris Glavic 10.53 ©Silberschatz, Korth and Sudarshan


## Snapshot Write: First Committer Wins

$X_0 = 100$

<p><math>T_1</math> deposits 50 in X</p> <p><math>r_1(X_0, 100)</math></p> <p><math>w_1(X_1, 150)</math> <math>commit_1</math></p> <p><math>X_1 = 150</math></p>	<p><math>T_2</math> withdraws 50 from X</p> <p><math>r_2(X_0, 100)</math> <math>w_2(X_2, 50)</math></p> <p><math>commit_2</math> (Serialization Error <math>T_2</math> is rolled back)</p>
--	--

- Variant: “**First-updater-wins**”
  - Check for concurrent updates when write occurs by locking item
    - But lock should be held till all concurrent transactions have finished
  - (Oracle uses this plus some extra features)
  - Differs only in when abort occurs, otherwise equivalent


CS425 – Fall 2016 – Boris Glavic 10.54 ©Silberschatz, Korth and Sudarshan



## Benefits of SI

- Reading is *never* blocked,
  - and also doesn't block other txns activities
- Performance similar to Read Committed
- Avoids the usual anomalies
  - No dirty read
  - No lost update
  - No non-repeatable read
  - Predicate based selects are repeatable (no phantoms)
- Problems with SI
  - SI does not always give serializable executions
    - Serializable: among two concurrent txns, one sees the effects of the other
    - In SI: neither sees the effects of the other
  - Result: Integrity constraints can be violated


CS425 – Fall 2016 – Boris Glavic 10.55 ©Silberschatz, Korth and Sudarshan



## Snapshot Isolation

- E.g. of problem with SI
  - T1: x:=y
  - T2: y:=x
  - Initially x = 3 and y = 17
    - Serial execution: x = ??, y = ??
    - if both transactions start at the same time, with snapshot isolation: x = ??, y = ??
- Called **skew write**
- Skew also occurs with inserts
  - E.g.:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1


CS425 – Fall 2016 – Boris Glavic 10.56 ©Silberschatz, Korth and Sudarshan



## Snapshot Isolation Anomalies

- SI breaks serializability when txns modify *different* items, each based on a previous state of the item the other modified
  - Not very common in practice
    - E.g., the TPC-C benchmark runs correctly under SI
    - when txns conflict due to modifying different data, there is usually also a shared item they both modify too (like a total quantity) so SI will abort one of them
  - But does occur
    - Application developers should be careful about write skew
- SI can also cause a read-only transaction anomaly, where read-only transaction may see an inconsistent state even if updaters are serializable
  - We omit details
- Using snapshots to verify primary/foreign key integrity can lead to inconsistency
  - Integrity constraint checking usually done outside of snapshot


CS425 – Fall 2016 – Boris Glavic 10.57 ©Silberschatz, Korth and Sudarshan



## SI In Oracle and PostgreSQL

- **Warning:** SI used when isolation level is set to serializable, by Oracle, and PostgreSQL versions prior to 9.1
  - PostgreSQL's implementation of SI (versions prior to 9.1) described in Section 26.4.1.3
  - Oracle implements "first updater wins" rule (variant of "first committer wins")
    - concurrent writer check is done at time of write, not at commit time
    - Allows transactions to be rolled back earlier
    - Oracle and PostgreSQL < 9.1 do not support true serializable execution
  - PostgreSQL 9.1 introduced new protocol called "Serializable Snapshot Isolation" (SSI)
    - Which guarantees true serializability including handling predicate reads (coming up)


CS425 – Fall 2016 – Boris Glavic 10.58 ©Silberschatz, Korth and Sudarshan



## SI In Oracle and PostgreSQL

- Can sidestep SI for specific queries by using **select .. for update** in Oracle and PostgreSQL
  - E.g.,
    1. **select max(orderno) from orders for update**
    2. read value into local variable maxorder
    3. insert into orders (maxorder+1, ...)
  - Select for update (SFU) treats all data read by the query as if it were also updated, preventing concurrent updates
  - Does not always ensure serializability since phantom phenomena can occur (coming up)
- In PostgreSQL versions < 9.1, SFU locks the data item, but releases locks when the transaction completes, even if other concurrent transactions are active
  - Not quite same as SFU in Oracle, which keeps locks until all concurrent transactions have completed

CS425 – Fall 2016 – Boris Glavic 10.59 ©Silberschatz, Korth and Sudarshan



## Insert and Delete Operations

- If two-phase locking is used :
  - A **delete** operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted.
  - A transaction that inserts a new tuple into the database is given an X-mode lock on the tuple
- Insertions and deletions can lead to the **phantom phenomenon**.
  - A transaction that scans a relation
    - (e.g., find sum of balances of all accounts in Perryridge)
  - and a transaction that inserts a tuple in the relation
    - (e.g., insert a new account at Perryridge)
  - (conceptually) conflict in spite of not accessing any tuple in common.
  - If only tuple locks are used, non-serializable schedules can result
    - E.g. the scan transaction does not see the new account, but reads some other tuple written by the update transaction

CS425 – Fall 2016 – Boris Glavic 10.60 ©Silberschatz, Korth and Sudarshan



## Insert and Delete Operations (Cont.)

- The transaction scanning the relation is reading information that indicates what tuples the relation contains, while a transaction inserting a tuple updates the same information.
  - The conflict should be detected, e.g. by locking the information.
- One solution:
  - Associate a data item with the relation, to represent the information about what tuples the relation contains.
  - Transactions scanning the relation acquire a shared lock in the data item,
  - Transactions inserting or deleting a tuple acquire an exclusive lock on the data item. (Note: locks on the data item do not conflict with locks on individual tuples.)
- Above protocol provides very low concurrency for insertions/deletions.
- Index locking protocols provide higher concurrency while preventing the phantom phenomenon, by requiring locks on certain index buckets.

CS425 – Fall 2016 – Boris Glavic

10.61

©Silberschatz, Korth and Sudarshan



## Index Locking Protocol

- Index locking protocol:
  - Every relation must have at least one index.
  - A transaction can access tuples only after finding them through one or more indices on the relation
  - A transaction  $T_i$  that performs a lookup must lock all the index leaf nodes that it accesses, in S-mode
    - ▶ Even if the leaf node does not contain any tuple satisfying the index lookup (e.g. for a range query, no tuple in a leaf is in the range)
  - A transaction  $T_i$  that inserts, updates or deletes a tuple  $t$  in a relation  $r$ 
    - ▶ must update all indices to  $r$
    - ▶ must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete
  - The rules of the two-phase locking protocol must be observed
- Guarantees that phantom phenomenon won't occur

CS425 – Fall 2016 – Boris Glavic

10.62

©Silberschatz, Korth and Sudarshan



## Next-Key Locking

- Index-locking protocol to prevent phantoms required locking entire leaf
  - Can result in poor concurrency if there are many inserts
- Alternative: for an index lookup
  - Lock all values that satisfy index lookup (match lookup value, or fall in lookup range)
  - Also lock next key value in index
  - Lock mode: S for lookups, X for insert/delete/update
- Ensures that range queries will conflict with inserts/deletes/updates
  - Regardless of which happens first, as long as both are concurrent

CS425 – Fall 2016 – Boris Glavic

10.63

©Silberschatz, Korth and Sudarshan



## Concurrency in Index Structures

- Indices are unlike other database items in that their only job is to help in accessing data.
- Index-structures are typically accessed very often, much more than other database items.
  - Treating index-structures like other database items, e.g. by 2-phase locking of index nodes can lead to low concurrency.
- There are several index concurrency protocols where locks on internal nodes are released early, and not in a two-phase fashion.
  - It is acceptable to have nonserializable concurrent access to an index as long as the accuracy of the index is maintained.
    - ▶ In particular, the exact values read in an internal node of a B\*-tree are irrelevant so long as we land up in the correct leaf node.

CS425 – Fall 2016 – Boris Glavic

10.64

©Silberschatz, Korth and Sudarshan



## Concurrency in Index Structures (Cont.)

- Example of index concurrency protocol:
- Use **crabbing** instead of two-phase locking on the nodes of the B\*-tree, as follows. During search/insertion/deletion:
  - First lock the root node in shared mode.
  - After locking all required children of a node in shared mode, release the lock on the node.
  - During insertion/deletion, upgrade leaf node locks to exclusive mode.
  - When splitting or coalescing requires changes to a parent, lock the parent in exclusive mode.
- Above protocol can cause excessive deadlocks
  - Searches coming down the tree deadlock with updates going up the tree
  - Can abort and restart search, without affecting transaction
- Better protocols are available; see Section 16.9 for one such protocol, the B-link tree protocol
  - Intuition: release lock on parent before acquiring lock on child
    - ▶ And deal with changes that may have happened between lock release and acquire

CS425 – Fall 2016 – Boris Glavic

10.65

©Silberschatz, Korth and Sudarshan



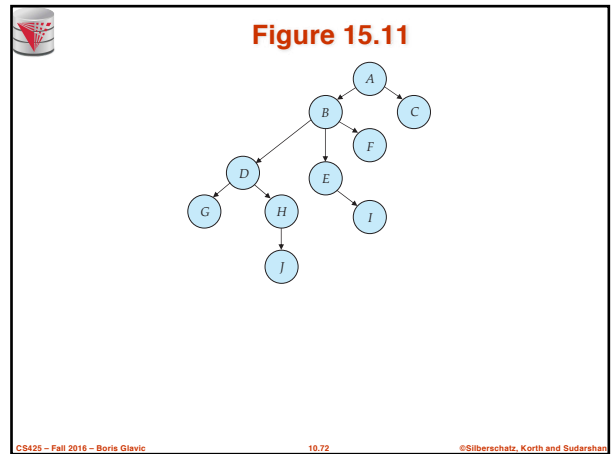
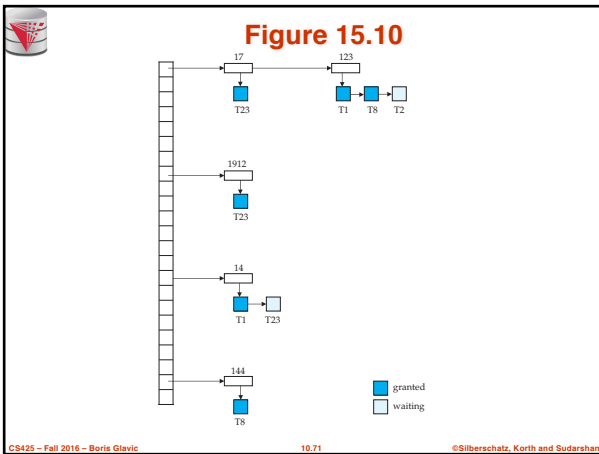
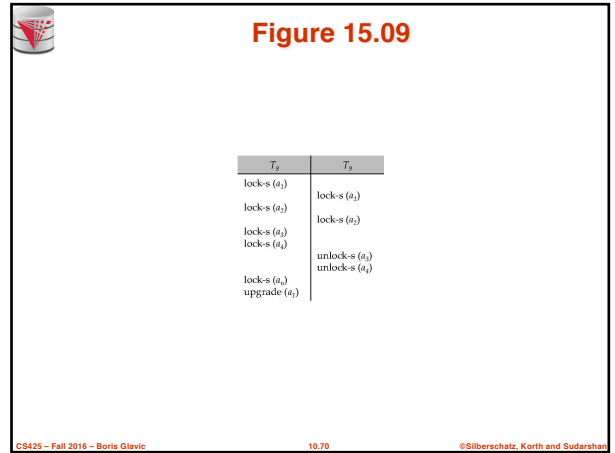
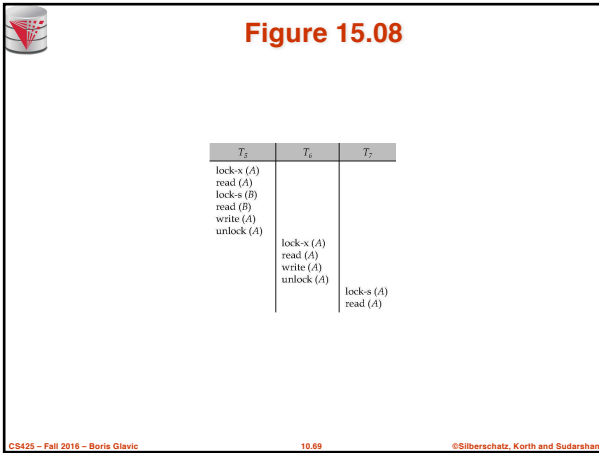
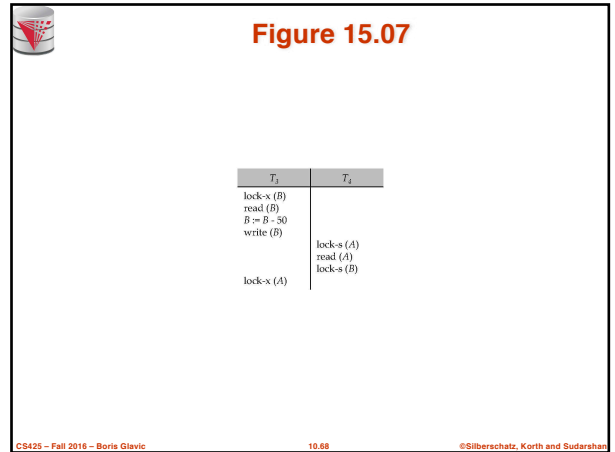
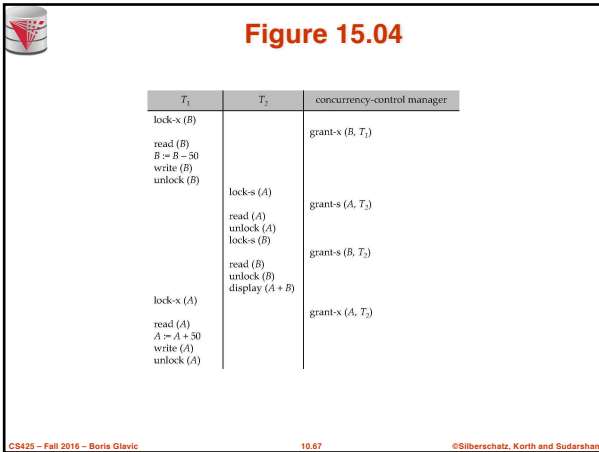
## Figure 15.01

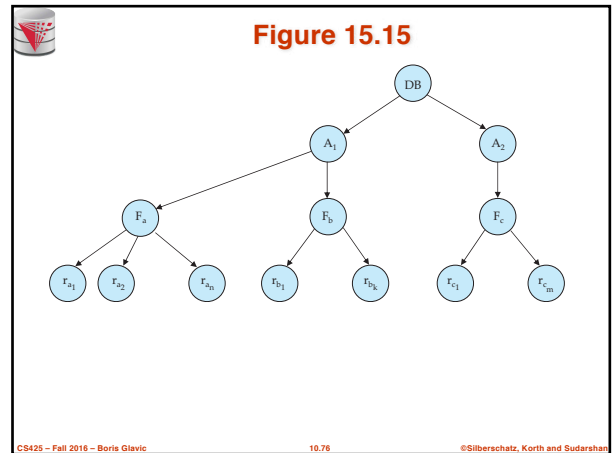
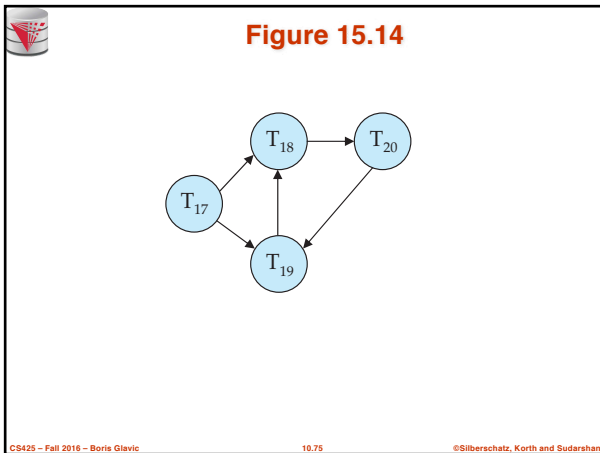
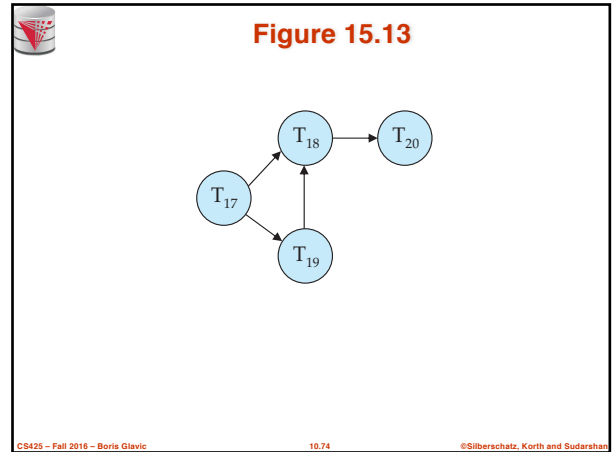
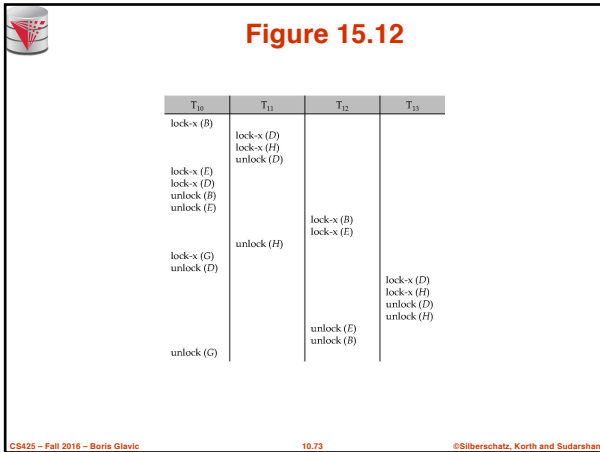
	S	X
S	true	false
X	false	false

CS425 – Fall 2016 – Boris Glavic

10.66

©Silberschatz, Korth and Sudarshan





### Figure 15.16

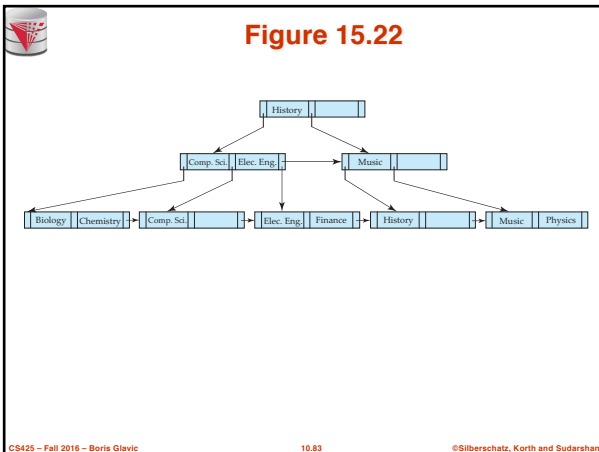
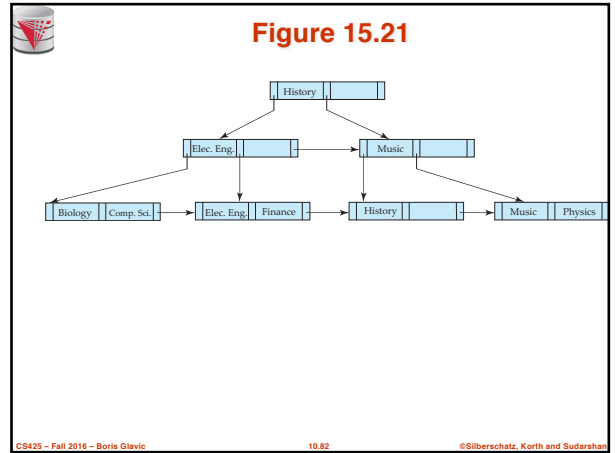
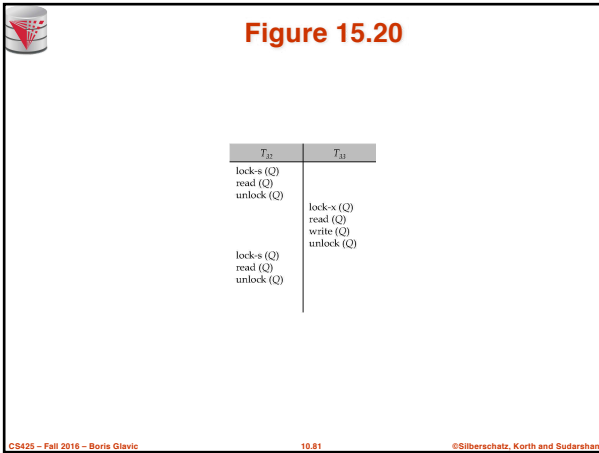
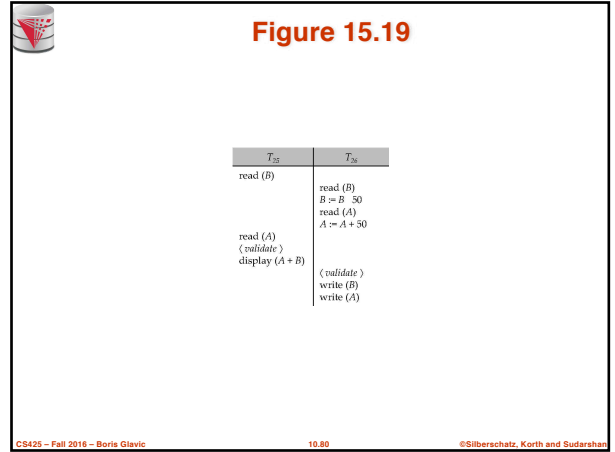
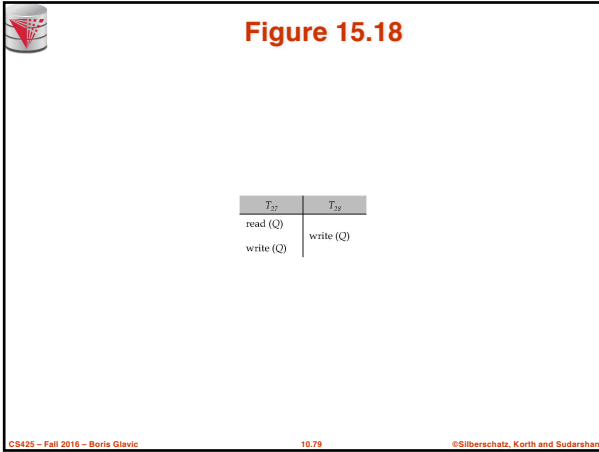
	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

CS425 – Fall 2016 – Boris Glavic 10.77 ©Silberschatz, Korth and Sudarshan

### Figure 15.17

T <sub>25</sub>	T <sub>35</sub>
read (B)	read (B) B = B - 50 write (B)
read (A)	read (A)
display (A + B)	A = A + 50 write (A) display (A + B)

CS425 – Fall 2016 – Boris Glavic 10.78 ©Silberschatz, Korth and Sudarshan



### Figure 15.23

	S	X	I
S	true	false	false
X	false	false	false
I	false	false	true

CS425 – Fall 2016 – Boris Glavic
10.84
©Silberschatz, Korth and Sudarshan



**Figure in-15.1**

$T_{27}$	$T_{28}$	$T_{29}$
read ( $Q$ )	write ( $Q$ )	write ( $Q$ )
write ( $Q$ )		



## Chapter 11: Indexing and Storage

Modified from:  
Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## Chapter 11: Indexing and Storage

- DBMS Storage
  - Memory hierarchy
  - File Organization
  - Buffering
- Indexing
  - Basic Concepts
  - B+-Trees
  - Static Hashing
  - Index Definition in SQL
  - Multiple-Key Access

CS425 – Fall 2013 – Boris Glavic

11.2

©Silberschatz, Korth and Sudarshan



## Memory Hierarchy

Modified from:  
Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## DBMS Storage

- Modern Computers have different types of memory
  - Cache, Main Memory, Harddisk, SSD, ...
- Memory types have different characteristics in terms of
  - Persistent vs. volatile
  - Speed (random vs. sequential access)
  - Size
  - Price – this usually determines size
- Database systems are designed to be use these different memory types effectively
  - Need for persistent storage: the state of the database needs to be written to persistent storage
    - guarantee database content is not lost when the computer is shutdown
  - Moving data between different types of memory
    - Want to use fast memory to speed-up operations
    - Need slower memory for the size

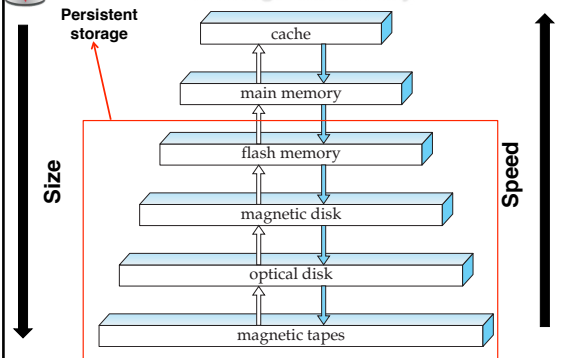
CS425 – Fall 2013 – Boris Glavic

11.4

©Silberschatz, Korth and Sudarshan



## Storage Hierarchy



CS425 – Fall 2013 – Boris Glavic

11.5

©Silberschatz, Korth and Sudarshan



## Main Memory vs. Disk

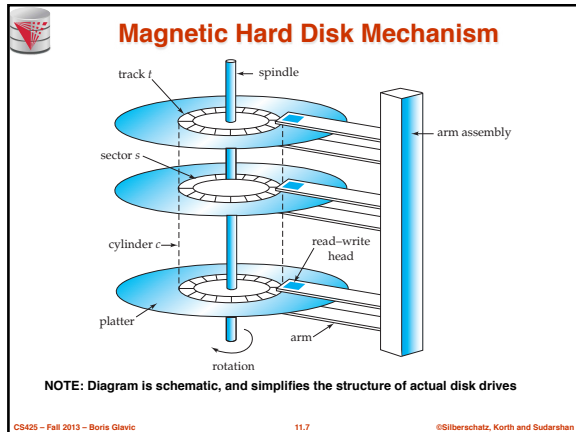
- Why do we not only use main memory
  - What if database does not fit into main memory?
  - Main memory is volatile
- Main memory vs. disk
  - Given available main memory when do we keep which part of the database in main memory
    - **Buffer manager**: Component of DBMS that decides when to move data between disk and main memory
  - How do we ensure transaction property durability
    - Buffer manager needs to make sure data written by committed transactions is written to disk to ensure durability

CS425 – Fall 2013 – Boris Glavic

11.6

©Silberschatz, Korth and Sudarshan





- ### Performance Measures of Disks
- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:
    - **Seek time** – time it takes to reposition the arm over the correct track.
      - ▶ Average seek time is 1/2 the worst case seek time.
      - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
      - ▶ 4 to 10 milliseconds on typical disks
    - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
      - ▶ Average latency is 1/2 of the worst case latency.
      - ▶ 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
  - **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
    - 25 to 100 MB per second max rate, lower for inner tracks
    - Multiple disks may share a controller, so rate that controller can handle is also important
      - ▶ E.g. SATA: 150 MB/sec, SATA-II 3Gb (300 MB/sec)
      - ▶ Ultra 320 SCSI: 320 MB/s, SAS (3 to 6 Gb/sec)
      - ▶ Fiber Channel (FC2Gb or 4Gb): 256 to 512 MB/s
- CS425 – Fall 2013 – Boris Glavic 11.8 ©Silberschatz, Korth and Sudarshan

- ### Random vs. Sequential Access
- Transfer of data from disk has a minimal size = 1 block
    - Reading 1 byte is as fast as reading one block (e.g., 4KB)
  - **Random Access**
    - Read data from anywhere on the disk
    - Need to get to the right track (**seek time**)
    - Need to wait until the right sector is under the arm (on avg 1/2 time for one rotation) (**rotational delay**)
    - Then can transfer data at ~ **transfer rate**
  - **Sequential Access**
    - Read data that is on the current track + sector
    - can transfer data at ~ **transfer rate**
  - Reading large number of small pieces of data randomly is very slow compared to sequential access
    - Thus, try layout data on disk in a way that enables sequential access
- CS425 – Fall 2013 – Boris Glavic 11.9 ©Silberschatz, Korth and Sudarshan

### File Organization

Modified from:  
Database System Concepts, 6th Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use

- ### File Organization
- The database is stored as a collection of **files**. Each file stores **records** (tuples from a table). A record is a sequence of **fields** (the attributes of a tuple).
  - Reading one record of a time from disk would be very slow (random access)
    - Organize our database files in pages (size of block or larger)
    - Read/write data in units of pages
    - One page will usually contain several records
  - One approach:
    - assume record size is fixed
    - each file has records of one particular type only
    - different files are used for different relations

This case is easiest to implement; will consider variable length records later.
- CS425 – Fall 2013 – Boris Glavic 11.11 ©Silberschatz, Korth and Sudarshan

- ### Fixed-Length Records
- Simple approach:
    - Store record  $i$  starting from byte  $n * (i - 1)$ , where  $n$  is the size of each record. Put maximal  $P / n$  records on each page.
    - Record access is simple but records may cross blocks
      - ▶ Modification: do not allow records to cross block boundaries
  - Deletion of record  $i$ : alternatives:
    - move records  $i + 1, \dots, n$  to  $i, \dots, n - 1$
    - move record  $n$  to  $i$
    - do not move records, but link all free records on a *free list*
- |           |       |            |            |       |
|-----------|-------|------------|------------|-------|
| record 0  | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1  | 12121 | Wu         | Finance    | 90000 |
| record 2  | 15151 | Mozart     | Music      | 40000 |
| record 3  | 22222 | Einstein   | Physics    | 95000 |
| record 4  | 32343 | El Said    | History    | 60000 |
| record 5  | 33456 | Gold       | Physics    | 87000 |
| record 6  | 43565 | Katz       | Comp. Sci. | 75000 |
| record 7  | 58583 | Califeri   | History    | 62000 |
| record 8  | 76543 | Singh      | Finance    | 80000 |
| record 9  | 76766 | Crick      | Biology    | 72000 |
| record 10 | 83821 | Brandt     | Comp. Sci. | 92000 |
| record 11 | 98345 | Kim        | Elec. Eng. | 80000 |
- CS425 – Fall 2013 – Boris Glavic 11.12 ©Silberschatz, Korth and Sudarshan

## Free Lists

- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as **pointers** since they “point” to the location of a record.

header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

CS425 – Fall 2013 – Boris Glavic 11.13 ©Silberschatz, Korth and Sudarshan

## Variable-Length Records

- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields such as strings (**varchar**)
  - Record types that allow repeating fields (used in some older data models).
- Attributes are stored in order
- Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- Null values represented by null-value bitmap

Null bitmap (stored in 1 byte)

Bytes 0 4 8 12 16 20 24 28 32 36 40 44 45

CS425 – Fall 2013 – Boris Glavic 11.14 ©Silberschatz, Korth and Sudarshan

## Variable-Length Records: Slotted Page Structure

- Slotted page** header contains:
  - number of record entries
  - end of free space in the block
  - location and size of each record
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
- Pointers should not point directly to record – instead they should point to the entry for the record in header.

CS425 – Fall 2013 – Boris Glavic 11.15 ©Silberschatz, Korth and Sudarshan

## Organization of Records in Files

- Heap** – a record can be placed anywhere in the file where there is space
  - Deletion efficient
  - Insertion efficient
  - Search is expensive
    - Example: Get instructor with name Glavic
      - Have to search through all instructors
- Sequential** – store records in sequential order, based on the value of some search key of each record
  - Deletion expensive and/or waste of space
  - Insertion expensive and/or waste of space
  - Search is efficient (e.g., binary search)
    - As long as the search is on the search key we are ordering on

CS425 – Fall 2013 – Boris Glavic 11.16 ©Silberschatz, Korth and Sudarshan

## Buffering


Modified from:  
**Database System Concepts, 6<sup>th</sup> Ed.**  
 ©Silberschatz, Korth and Sudarshan  
 See [www.db-book.com](http://www.db-book.com) for conditions on re-use

CS425 – Fall 2013 – Boris Glavic 11.18 ©Silberschatz, Korth and Sudarshan

## Buffer Manager

- Buffer Manager**
  - Responsible for loading pages from disk and writing modified pages back to disk
- Handling blocks**
  - If the block is already in the buffer, the buffer manager returns the address of the block in main memory
  - If the block is not in the buffer, the buffer manager
    - Allocates space in the buffer for the block
      - Replacing (throwing out) some other block, if required, to make space for the new block.
      - Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
    - Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester.


CS425 – Fall 2013 – Boris Glavic 11.18 ©Silberschatz, Korth and Sudarshan



## Buffer-Replacement Policies

- Most operating systems replace the block **least recently used** (LRU strategy)
- Idea behind LRU – use past pattern of block references as a predictor of future references
- Queries have well-defined access patterns (such as sequential scans), and a database system can use the information in a user’s query to predict future references
  - LRU can be a bad strategy for certain access patterns involving repeated scans of data
    - For example: when computing the join of 2 relations  $r$  and  $s$  by a nested loops
      - for each tuple  $tr$  of  $r$  do
      - for each tuple  $ts$  of  $s$  do
      - if the tuples  $tr$  and  $ts$  match ...
  - Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable


CS425 – Fall 2013 – Boris Glavic 11.19 ©Silberschatz, Korth and Sudarshan



## Buffer-Replacement Policies (Cont.)

- **Pinned block** – memory block that is not allowed to be written back to disk. E.g., an operation still needs this block.
- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
- **Most recently used (MRU) strategy** – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.
- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
  - E.g., the data dictionary is frequently accessed. Heuristic: keep data-dictionary blocks in main memory buffer
- Buffer managers also support **forced output** of blocks for the purpose of recovery (more in Chapter 16 in the textbook)


CS425 – Fall 2013 – Boris Glavic 11.20 ©Silberschatz, Korth and Sudarshan



## Indexing and Hashing

Modified from:  
Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use

CS425 – Fall 2013 – Boris Glavic 11.21 ©Silberschatz, Korth and Sudarshan




## Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute or set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form
 

search-key	pointer
------------	---------
- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in some sorted order
  - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.


CS425 – Fall 2013 – Boris Glavic 11.22 ©Silberschatz, Korth and Sudarshan



## Index Evaluation Metrics

- Access types supported efficiently. E.g.,
  - records with a specified value in the attribute
  - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

CS425 – Fall 2013 – Boris Glavic 11.23 ©Silberschatz, Korth and Sudarshan



## Ordered Indices

- In an **ordered index**, index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index:** an index whose search key specifies an order different from the sequential order of the file. Also called **non-clustering index**.
- **Index-sequential file:** ordered sequential file with a primary index.

CS425 – Fall 2013 – Boris Glavic 11.24 ©Silberschatz, Korth and Sudarshan

### Secondary Indices Example

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

Secondary index on salary field of instructor

- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

CS425 – Fall 2013 – Boris Glavic 11.25 ©Silberschatz, Korth and Sudarshan

### Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.
- BUT:** Updating indices imposes overhead on database modification --when a file is modified, every index on the file must be updated,
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access

CS425 – Fall 2013 – Boris Glavic 11.26 ©Silberschatz, Korth and Sudarshan

### Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

CS425 – Fall 2013 – Boris Glavic 11.27 ©Silberschatz, Korth and Sudarshan

### Multilevel Index (Cont.)

CS425 – Fall 2013 – Boris Glavic 11.28 ©Silberschatz, Korth and Sudarshan

### Index Update: Deletion

10101	10101	Srinivasan	Comp. Sci.	65000
32343	12121	Wu	Finance	90000
76766	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

- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
- Single-level index entry deletion:**
  - Dense indices** – deletion of search-key is similar to file record deletion.
  - Sparse indices** –
    - if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).
    - If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

CS425 – Fall 2013 – Boris Glavic 11.29 ©Silberschatz, Korth and Sudarshan

### Index Update: Insertion

- Single-level index insertion:**
  - Perform a lookup using the search-key value appearing in the record to be inserted.
  - Dense indices** – if the search-key value does not appear in the index, insert it.
  - Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index 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.
- Multilevel insertion and deletion:** algorithms are simple extensions of the single-level algorithms

CS425 – Fall 2013 – Boris Glavic 11.30 ©Silberschatz, Korth and Sudarshan

## Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
  - Example 1: In the *instructor* relation stored sequentially by ID, we may want to find all instructors in a particular department
  - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values
- We can have a secondary index with an index record for each search-key value

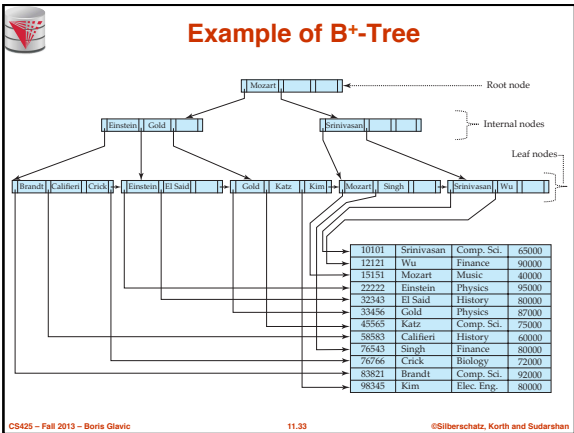
CS425 – Fall 2013 – Boris Glavic 11.31 ©Silberschatz, Korth and Sudarshan

## B+-Tree Index

B+-tree indices are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files
  - performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.
- Advantage of B+-tree index files:
  - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B+-trees:
  - extra insertion and deletion overhead, space overhead.
- Advantages of B+-trees outweigh disadvantages
  - B+-trees are used extensively

CS425 – Fall 2013 – Boris Glavic 11.32 ©Silberschatz, Korth and Sudarshan



## B+-Tree Index Files (Cont.)

A B+-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between  $\lceil n/2 \rceil$  and  $n$  children.
- A leaf node has between  $\lceil (n-1)/2 \rceil$  and  $n-1$  values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and  $(n-1)$  values.

CS425 – Fall 2013 – Boris Glavic 11.34 ©Silberschatz, Korth and Sudarshan

## B+-Tree Node Structure

■ Typical node

$P_1$	$K_1$	$P_2$	...	$P_{n-1}$	$K_{n-1}$	$P_n$
-------	-------	-------	-----	-----------	-----------	-------

- $K_i$  are the search-key values
- $P_i$  are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered
 
$$K_1 < K_2 < K_3 < \dots < K_{n-1}$$
 (Initially assume no duplicate keys, address duplicates later)

CS425 – Fall 2013 – Boris Glavic 11.35 ©Silberschatz, Korth and Sudarshan

## Leaf Nodes in B+-Trees

Properties of a leaf node:

- For  $i = 1, 2, \dots, n-1$ , pointer  $P_i$  points to a file record with search-key value  $K_i$ .
- If  $L_i, L_j$  are leaf nodes and  $i < j$ ,  $L_i$ 's search-key values are less than or equal to  $L_j$ 's search-key values
- $P_n$  points to next leaf node in search-key order

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

CS425 – Fall 2013 – Boris Glavic 11.36 ©Silberschatz, Korth and Sudarshan

## Non-Leaf Nodes in B<sup>+</sup>-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with  $m$  pointers:
  - All the search-keys in the subtree to which  $P_i$  points are less than  $K_i$
  - For  $2 \leq i \leq n-1$ , all the search-keys in the subtree to which  $P_i$  points have values greater than or equal to  $K_{i-1}$  and less than  $K_i$
  - All the search-keys in the subtree to which  $P_n$  points have values greater than or equal to  $K_{n-1}$

$P_1$	$K_1$	$P_2$	...	$P_{n-1}$	$K_{n-1}$	$P_n$
-------	-------	-------	-----	-----------	-----------	-------

CS425 – Fall 2013 – Boris Glavic
11.37
©Silberschatz, Korth and Sudarshan

## Example of B<sup>+</sup>-tree

B<sup>+</sup>-tree for instructor file ( $n = 6$ )

- Leaf nodes must have between 3 and 5 values ( $\lceil (n-1)/2 \rceil$  and  $n-1$ , with  $n = 6$ ).
- Non-leaf nodes other than root must have between 3 and 6 children ( $\lceil n/2 \rceil$  and  $n$  with  $n = 6$ ).
- Root must have at least 2 children.

CS425 – Fall 2013 – Boris Glavic
11.38
©Silberschatz, Korth and Sudarshan

## Observations about B<sup>+</sup>-trees

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B<sup>+</sup>-tree form a hierarchy of sparse indices.
- The B<sup>+</sup>-tree contains a relatively small number of levels
  - ▶ Level below root has at least  $2^* \lceil n/2 \rceil$  values
  - ▶ Next level has at least  $2^* \lceil n/2 \rceil^* \lceil n/2 \rceil$  values
  - ▶ ... etc.
- If there are  $K$  search-key values in the file, the tree height is no more than  $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
- thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

CS425 – Fall 2013 – Boris Glavic
11.39
©Silberschatz, Korth and Sudarshan

## Queries on B<sup>+</sup>-Trees

- Find record with search-key value  $V$ .
  1.  $C = \text{root}$
  2. While  $C$  is not a leaf node {
    1. Let  $i$  be least value s.t.  $V \leq K_i$ .
    2. If no such exists, set  $C = \text{last non-null pointer in } C$
    3. Else { if ( $V = K_i$ ) Set  $C = P_{i+1}$  else set  $C = P_i$  }
  3. Let  $i$  be least value s.t.  $K_i = V$
  4. If there is such a value  $i$ , follow pointer  $P_i$  to the desired record.
  5. Else no record with search-key value  $k$  exists.

CS425 – Fall 2013 – Boris Glavic
11.40
©Silberschatz, Korth and Sudarshan

## Handling Duplicates

- With duplicate search keys
  - In both leaf and internal nodes,
    - ▶ we cannot guarantee that  $K_1 < K_2 < K_3 < \dots < K_{n-1}$
    - ▶ but can guarantee  $K_1 \leq K_2 \leq K_3 \leq \dots \leq K_{n-1}$
  - Search-keys in the subtree to which  $P_i$  points
    - ▶ are  $\leq K_i$ , but not necessarily  $< K_i$
    - ▶ To see why, suppose same search key value  $V$  is present in two leaf node  $L_i$  and  $L_{i+1}$ . Then in parent node  $K_i$  must be equal to  $V$

CS425 – Fall 2013 – Boris Glavic
11.41
©Silberschatz, Korth and Sudarshan

## Handling Duplicates

- We modify find procedure as follows
  - traverse  $P_i$  even if  $V = K_i$
  - As soon as we reach a leaf node  $C$  check if  $C$  has only search key values less than  $V$ 
    - ▶ if so set  $C = \text{right sibling of } C$  before checking whether  $C$  contains  $V$
- Procedure printAll
  - uses modified find procedure to find first occurrence of  $V$
  - Traverse through consecutive leaves to find all occurrences of  $V$

\*\* Errata note: modified find procedure missing in first printing of 6<sup>th</sup> edition

CS425 – Fall 2013 – Boris Glavic
11.42
©Silberschatz, Korth and Sudarshan

### Queries on B+ Trees (Cont.)

- 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 as a disk block, typically 4 kilobytes
  - and  $n$  is typically around 100 (40 bytes per index entry).
- With 1 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 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

CS425 – Fall 2013 – Boris Glavic 11.43 ©Silberschatz, Korth and Sudarshan

### Updates on B+ Trees: Insertion

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
  1. Add record to the file
  2. If necessary add a pointer to the bucket.
3. If the search-key value is not present, then
  1. add the record to the main file (and create a bucket if necessary)
  2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
  3. Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide.

CS425 – Fall 2013 – Boris Glavic 11.44 ©Silberschatz, Korth and Sudarshan

### Updates on B+ Trees: Insertion (Cont.)

- Splitting a leaf node:
  - take the  $n$  (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first  $\lceil n/2 \rceil$  in the original node, and the rest in a new node.
  - let the new node be  $p$ , and let  $k$  be the least key value in  $p$ . Insert  $(k, p)$  in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.
- Splitting of nodes proceeds upwards till a node that is not full is found.
  - In the worst case the root node may be split increasing the height of the tree by 1.

Result of splitting node containing Brandt, Califieri and Crick on inserting Adams  
Next step: insert entry with (Califieri, pointer-to-new-node) into parent

CS425 – Fall 2013 – Boris Glavic 11.45 ©Silberschatz, Korth and Sudarshan

### B+ Tree Insertion

B+ Tree before and after insertion of "Adams"

CS425 – Fall 2013 – Boris Glavic 11.46 ©Silberschatz, Korth and Sudarshan

### B+ Tree Insertion

B+ Tree before and after insertion of "Lampert"

CS425 – Fall 2013 – Boris Glavic 11.47 ©Silberschatz, Korth and Sudarshan

### Insertion in B+ Trees (Cont.)

- Splitting a non-leaf node: when inserting  $(k, p)$  into an already full internal node  $N$ 
  - Copy  $N$  to an in-memory area  $M$  with space for  $n+1$  pointers and  $n$  keys
  - Insert  $(k, p)$  into  $M$
  - Copy  $P_1, K_1, \dots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$  from  $M$  back into node  $N$
  - Copy  $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \dots, K_n, P_{n+1}$  from  $M$  into newly allocated node  $N'$
  - Insert  $(K_{\lceil n/2 \rceil}, N')$  into parent  $N$
- Read pseudocode in book!

CS425 – Fall 2013 – Boris Glavic 11.48 ©Silberschatz, Korth and Sudarshan

### Examples of B+-Tree Deletion

Before and after deleting "Srinivasan"

- Deleting "Srinivasan" causes merging of under-full leaves

CS425 – Fall 2013 – Boris Glavic 11.49 ©Silberschatz, Korth and Sudarshan

### Examples of B+-Tree Deletion (Cont.)

Deletion of "Singh" and "Wu" from result of previous example

- Leaf containing Singh and Wu became underfull, and borrowed a value Kim from its left sibling
- Search-key value in the parent changes as a result

CS425 – Fall 2013 – Boris Glavic 11.50 ©Silberschatz, Korth and Sudarshan

### Example of B+-tree Deletion (Cont.)

Before and after deletion of "Gold" from earlier example

- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
  - Value separating two nodes (at the parent) is pulled down when merging
- Root node then has only one child, and is deleted

CS425 – Fall 2013 – Boris Glavic 11.51 ©Silberschatz, Korth and Sudarshan

### Updates on B+-Trees: Deletion

- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then *merge siblings*:
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair  $(K_{i+1}, P_i)$ , where  $P_i$  is the pointer to the deleted node, from its parent, recursively using the above procedure.

CS425 – Fall 2013 – Boris Glavic 11.52 ©Silberschatz, Korth and Sudarshan

### Updates on B+-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then *redistribute pointers*:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has  $\lceil n/2 \rceil$  or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

CS425 – Fall 2013 – Boris Glavic 11.53 ©Silberschatz, Korth and Sudarshan

### Non-Unique Search Keys

- Alternatives to scheme described earlier
  - Buckets on separate block (bad idea)
  - List of tuple pointers with each key
    - Extra code to handle long lists
    - Deletion of a tuple can be expensive if there are many duplicates on search key (why?)
    - Low space overhead, no extra cost for queries
  - Make search key unique by adding a record-identifier
    - Extra storage overhead for keys
    - Simpler code for insertion/deletion
    - Widely used

CS425 – Fall 2013 – Boris Glavic 11.54 ©Silberschatz, Korth and Sudarshan



## B+-Tree File Organization

- Index file degradation problem is solved by using B+-Tree indices.
- Data file degradation problem is solved by using B+-Tree File Organization.
- The leaf nodes in a B+-tree file organization store records, instead of pointers.
- Leaf nodes are still required to be half full
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B+-tree index.

CS425 – Fall 2013 – Boris Glavic 11.55 ©Silberschatz, Korth and Sudarshan

## B+-Tree File Organization (Cont.)

Example of B+-tree File Organization

- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least  $\lfloor 2n/3 \rfloor$  entries

CS425 – Fall 2013 – Boris Glavic 11.56 ©Silberschatz, Korth and Sudarshan

## Hashing

Modified from:  
Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use

CS425 – Fall 2013 – Boris Glavic 11.58 ©Silberschatz, Korth and Sudarshan

## Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function**.
- Hash function  $h$  is a function from the set of all search-key values  $K$  to the set of all bucket addresses  $B$ .
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

CS425 – Fall 2013 – Boris Glavic 11.58 ©Silberschatz, Korth and Sudarshan

## Example of Hash File Organization

Hash file organization of *instructor* file, using *dept\_name* as key  
(See figure in next slide.)

- There are 10 buckets.
- The binary representation of the  $i$ th character is assumed to be the integer  $i$ .
- The hash function returns the sum of the binary representations of the characters modulo 10
  - E.g.  $h(\text{Music}) = 1$      $h(\text{History}) = 2$   
 $h(\text{Physics}) = 3$      $h(\text{Elec. Eng.}) = 3$

CS425 – Fall 2013 – Boris Glavic 11.59 ©Silberschatz, Korth and Sudarshan

## Example of Hash File Organization

bucket 0				bucket 4			
				12121	Wu	Finance	90000
				76543	Singh	Finance	80000
bucket 1				bucket 5			
15151	Mozart	Music	40000	76766	Crick	Biology	72000
bucket 2				bucket 6			
32343	El Said	History	80000	10101	Srinivasan	Comp. Sci.	65000
58583	Califeri	History	60000	45565	Katz	Comp. Sci.	75000
				83821	Brandt	Comp. Sci.	92000
bucket 3				bucket 7			
22222	Einstein	Physics	95000				
33456	Gold	Physics	87000				
98345	Kim	Elec. Eng.	80000				

Hash file organization of *instructor* file, using *dept\_name* as key  
(see previous slide for details).

CS425 – Fall 2013 – Boris Glavic 11.60 ©Silberschatz, Korth and Sudarshan

## Hash Functions

- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**, 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 sum modulo the number of buckets could be returned. .

CS425 – Fall 2013 – Boris Glavic 11.61 ©Silberschatz, Korth and Sudarshan

## Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using **overflow buckets**.

CS425 – Fall 2013 – Boris Glavic 11.62 ©Silberschatz, Korth and Sudarshan

## Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed hashing**.
  - An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.

CS425 – Fall 2013 – Boris Glavic 11.63 ©Silberschatz, Korth and Sudarshan

## Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary indices
  - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - However, we use the term hash index to refer to both secondary index structures and hash organized files.

CS425 – Fall 2013 – Boris Glavic 11.64 ©Silberschatz, Korth and Sudarshan

## Example of Hash Index

76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45665	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
88583	Callifieri	History	62000
15151	Mozart	Music	40000
33456	Einstein	Physics	95000
83821	Gold	Physics	87000

hash index on *instructor*, on attribute *ID*

CS425 – Fall 2013 – Boris Glavic 11.65 ©Silberschatz, Korth and Sudarshan

## Deficiencies of Static Hashing

- In static hashing, function  $h$  maps search-key values to a fixed set of  $B$  of bucket addresses. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
  - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

CS425 – Fall 2013 – Boris Glavic 11.66 ©Silberschatz, Korth and Sudarshan

## Index Definition in SQL

- Create an index
  - create index** <index-name> **on** <relation-name> (<attribute-list>)
  - E.g.: **create index** *b-index* **on** *branch*(*branch\_name*)
- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index
  - drop index** <index-name>
- Most database systems allow specification of type of index, and clustering.

CS425 – Fall 2013 – Boris Glavic 11.67 ©Silberschatz, Korth and Sudarshan

## End of Chapter

Modified from:  
Database System Concepts, 6<sup>th</sup> Ed.  
©Silberschatz, Korth and Sudarshan  
See [www.db-book.com](http://www.db-book.com) for conditions on re-use

### Figure 11.01

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

CS425 – Fall 2013 – Boris Glavic 11.69 ©Silberschatz, Korth and Sudarshan

### Figure 11.15

CS425 – Fall 2013 – Boris Glavic 11.70 ©Silberschatz, Korth and Sudarshan

## Partitioned Hashing

- Hash values are split into segments that depend on each attribute of the search-key.
  - $(A_1, A_2, \dots, A_n)$  for  $n$  attribute search-key
- Example:  $n = 2$ , for *customer*, search-key being (*customer-street*, *customer-city*)
 

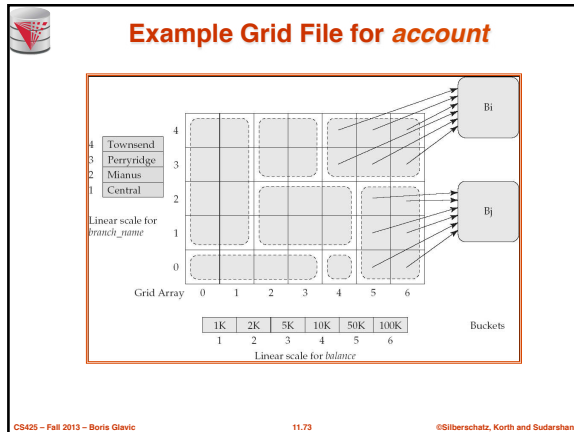
search-key value	hash value
(Main, Harrison)	101 111
(Main, Brooklyn)	101 001
(Park, Palo Alto)	010 010
(Spring, Brooklyn)	001 001
(Alma, Palo Alto)	110 010
- To answer equality query on single attribute, need to look up multiple buckets. Similar in effect to grid files.

CS425 – Fall 2013 – Boris Glavic 11.71 ©Silberschatz, Korth and Sudarshan

## Grid Files

- Structure used to speed the processing of general multiple search-key queries involving one or more comparison operators.
- The **grid file** has a single grid array and one linear scale for each search-key attribute. The grid array has number of dimensions equal to number of search-key attributes.
- Multiple cells of grid array can point to same bucket
- To find the bucket for a search-key value, locate the row and column of its cell using the linear scales and follow pointer

CS425 – Fall 2013 – Boris Glavic 11.72 ©Silberschatz, Korth and Sudarshan



### Queries on a Grid File

- A grid file on two attributes *A* and *B* can handle queries of all following forms with reasonable efficiency
  - $(a_1 \leq A \leq a_2)$
  - $(b_1 \leq B \leq b_2)$
  - $(a_1 \leq A \leq a_2 \wedge b_1 \leq B \leq b_2)$ .
- E.g., to answer  $(a_1 \leq A \leq a_2 \wedge b_1 \leq B \leq b_2)$ , use linear scales to find corresponding candidate grid array cells, and look up all the buckets pointed to from those cells.

CS425 – Fall 2013 – Boris Glavic 11.74 ©Silberschatz, Korth and Sudarshan

### Grid Files (Cont.)

- During insertion, if a bucket becomes full, new bucket can be created if more than one cell points to it.
  - Idea similar to extendable hashing, but on multiple dimensions
  - If only one cell points to it, either an overflow bucket must be created or the grid size must be increased
- Linear scales must be chosen to uniformly distribute records across cells.
  - Otherwise there will be too many overflow buckets.
- Periodic re-organization to increase grid size will help.
  - But reorganization can be very expensive.
- Space overhead of grid array can be high.
- R-trees (Chapter 23) are an alternative

CS425 – Fall 2013 – Boris Glavic 11.75 ©Silberschatz, Korth and Sudarshan