

# Introduction to Database Systems

## IDBS - Fall 2024

### Lecture 1 - Introduction

Course Introduction

DBMS Introduction

Relational Data Model

SQL DDL

---

Readings: PDBM 1, 6.1, 7.1-7.2

Omar Shahbaz Khan

# Course Responsible

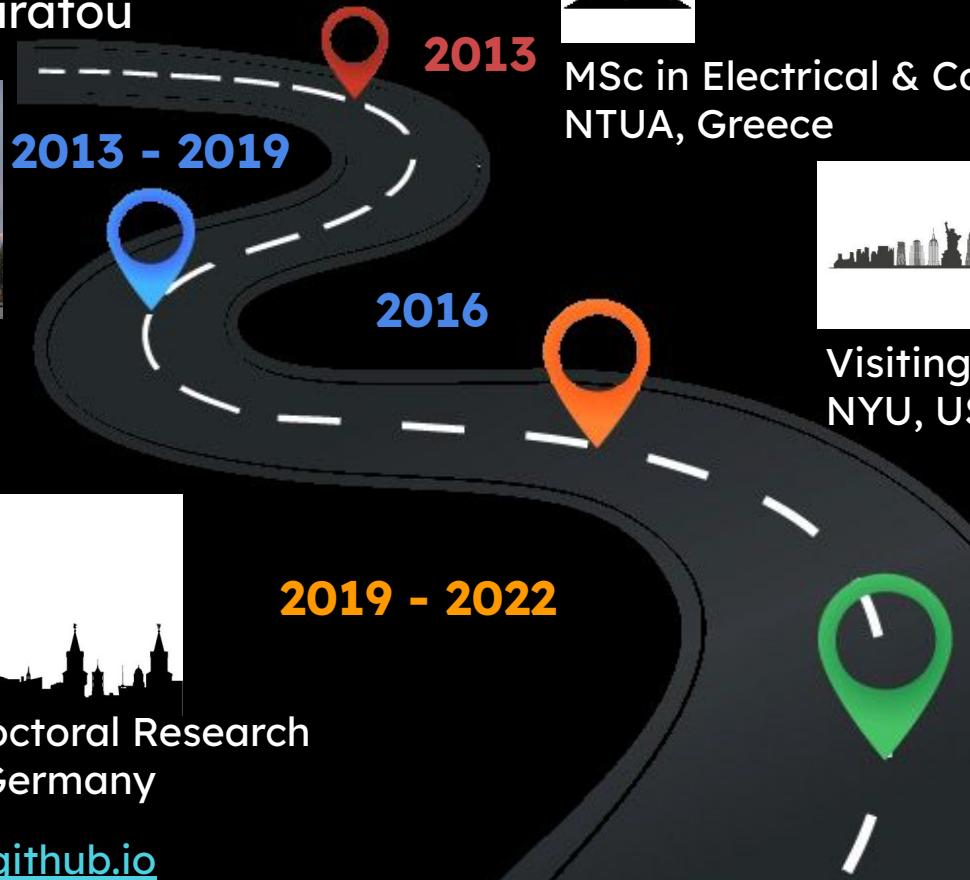
Eleni Tzirita Zacharatou



Ph.D. in CS  
EPFL, Switzerland



Postdoctoral Research  
TUB, Germany



MSc in Electrical & Computer Engineering  
NTUA, Greece



Visiting Researcher  
NYU, USA

2022 - Now



Assistant Professor  
ITU, Denmark

# Lecturer

Omar Shahbaz Khan



**2013 - 2022**

BSc (SWU), MSc (CS), PhD in CS  
ITU

Website: [oskhan.com](http://oskhan.com)

IT UNIVERSITY OF COPENHAGEN



**2022 - Now**

Postdoctoral Researcher,  
Reykjavik University,  
ITU (20%)

# Guest Lecturer

Martin Hentschel

- BSc, MSc, PhD in CS from ETH Zurich, Switzerland
- Microsoft
- Snowflake
- ITU
- Research Interests: database systems, open data and metadata formats, and data management in general
- <https://dasya.itu.dk/people/martinhentschel/>

# Teaching Assistants

**Anders Arvesen**

Study program: MSc. Computer Science

**Adam Hadou Temsamani**

Study program: MSc. Computer Science

**Anne-Marie Rommerdahl**

Study program: BSc. Software Development

**Cristina Avram**

Study program: BSc. Data Science

**Erling Amundsen**

Study program: MSc. Data Science

**Marcus Henrik Simonsen**

Study program: MSc. Computer Science

**Oliver Flyckt Wilhjelm**

Study program: MSc. Games (Technology)

**Philip Kristian Møller Flyvholm**

Study program: MSc. Computer Science

# Research Group



[www.dasya.itu.dk](http://www.dasya.itu.dk)

Lab for Research and Education:  
5th floor (5A56)

15+ People

Entire Data Lifecycle:

- Collection
- Transfer
- Storage
- Curation
- Processing
- Analytics

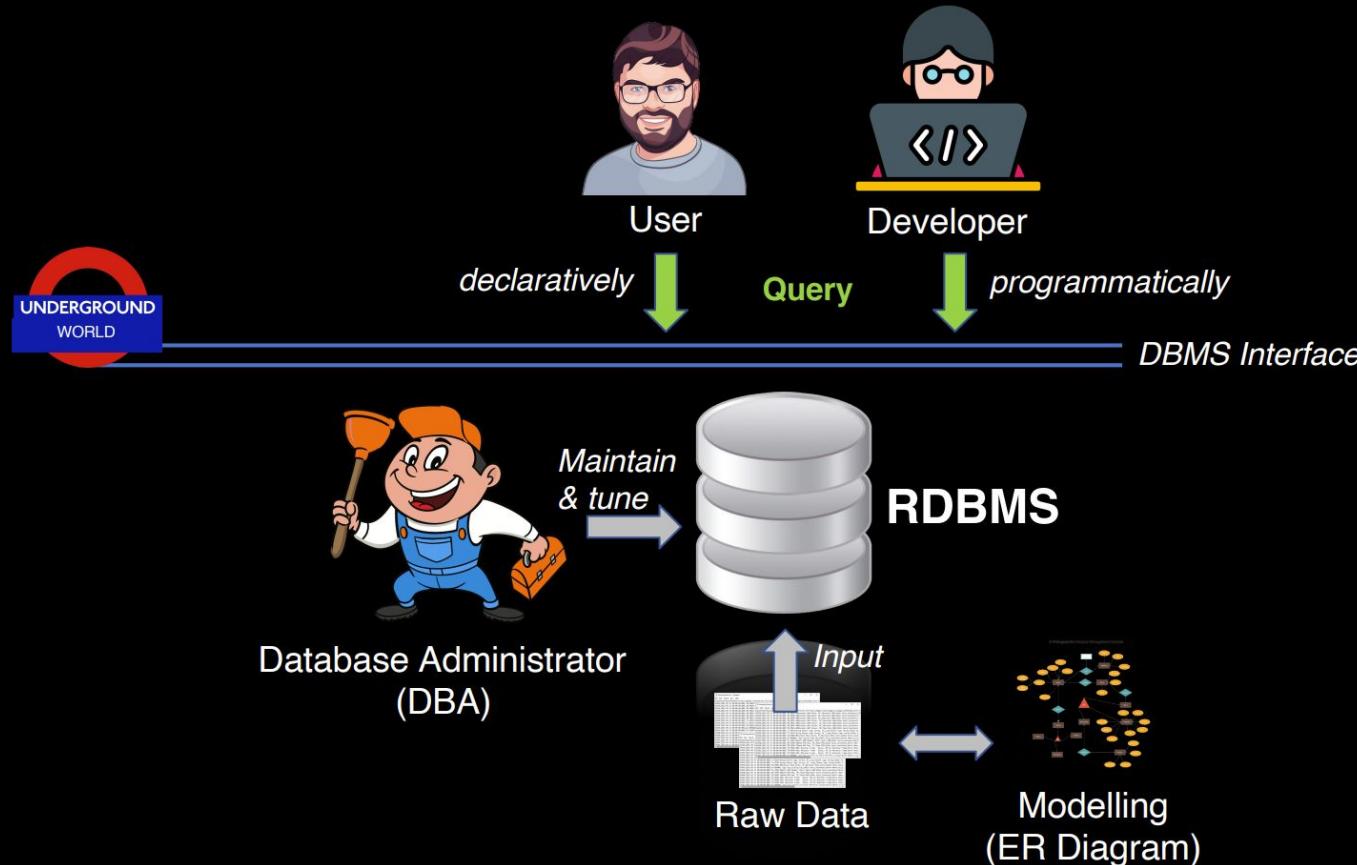
Infrastructure for Data Science

# But this about you!

- ~250 Students
  - Mostly from BSc. in Software Development
- Too many for a round table
- What do you expect from Introduction to Database Systems?
  - Mentimeter: [www.menti.com](http://www.menti.com) | Code: 2244 4401



# What you will learn



# Intended Learning Outcomes

- Write SQL queries: multiple relations; compound conditions; grouping; aggregation; and subqueries.
- Use relational DBMSs from a conventional programming language in a secure manner
- Suggest a database design in the ER model and convert to a relational database schema in a suitable normal form
- Analyze/predict/improve query processing efficiency of the designed database using indices
- Reflect upon the evolution of the hardware and storage hierarchy and its impact on data management system design.
- Discuss the pros and cons of different classes of data systems for modern analytics and data science applications.

# Intended Learning Outcomes

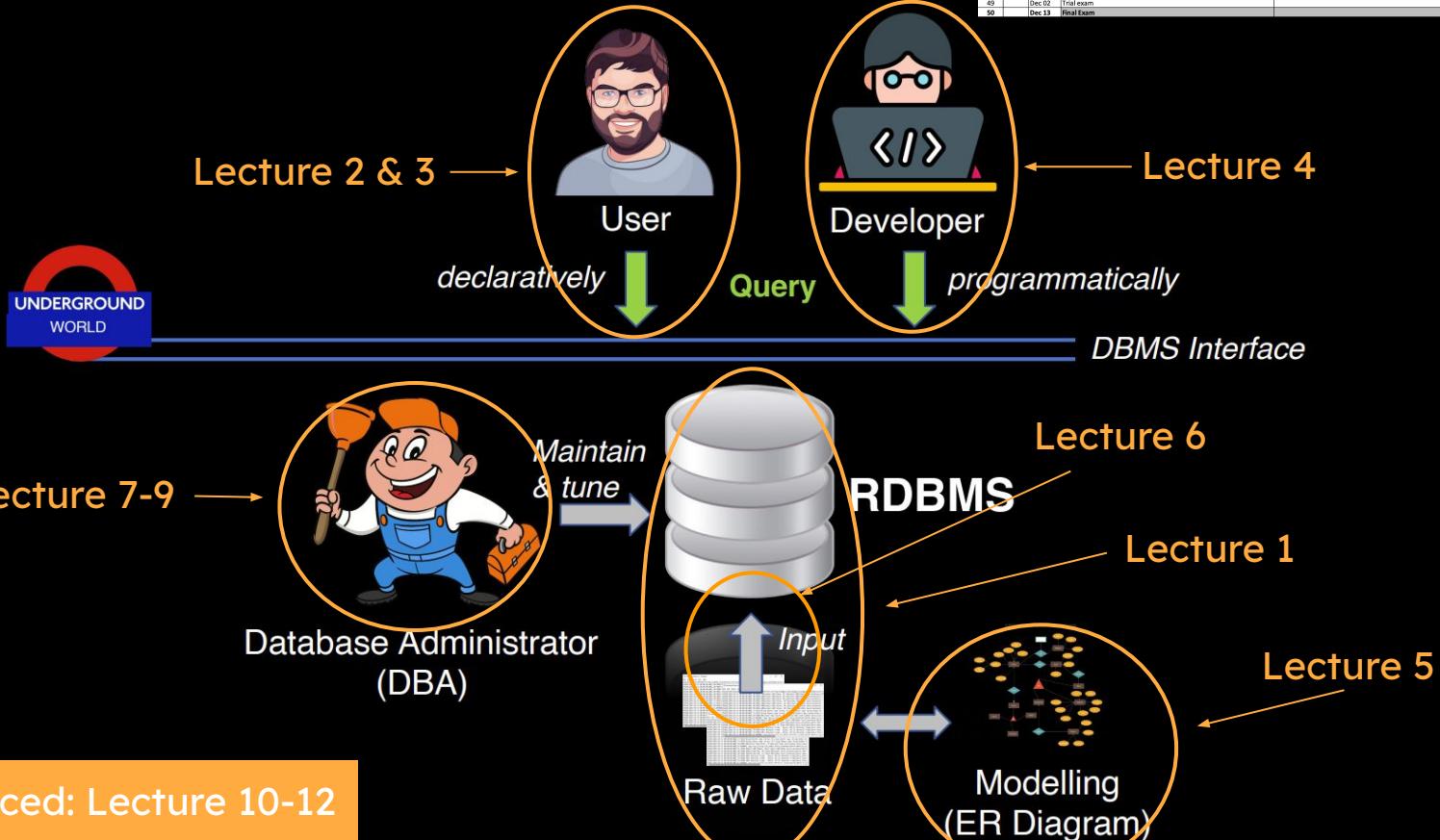
- Write SQL queries: multiple relations; compound conditions; grouping; aggregation; and subqueries.
  - Use relational algebra and SQL to manipulate data.
  - Suggest a suitable schema for a given database application.
  - Analyse the performance of a database using various indicators.
  - Reflect on the impact of data management system design.
  - Discuss the pros and cons of different classes of data systems for modern analytics and data science applications.
1. Getting into Database Systems
  2. Getting data using SQL and from your apps
  3. Design a database
  4. Tune a database
  5. Advanced databases (internals and big data)

# Course Schedule

Week	Lecture	Date	Lecture Topic	Readings	Exercises / Homeworks	Notes
36	1	Sep 2	Introduction; Course overview; Relational model	PDBM: 1, 6.1, 7.1-7.2	Exercise L1	Lecturer: Omar Shahbaz Khan
37	2	Sep 09	SQL DDL/DML; Basic SQL queries; Joins; Aggregations	PDBM: 7.3	Exercise L2	Lecturer: Eleni Tzirita Zacharou
38	3	Sep 16	Complex SQL queries; Subqueries; Views	PDBM: 7.3 - 7.4	Homework 1 (due on 23.09.2024)	Lecturer: Eleni Tzirita Zacharou
39	4	Sep 23	Triggers; Transactions; Using SQL from Python	PDBM: 9.2, 14.1, 14.2.1, 14.5 + more (details below)	Exercise L4	Lecturer: Omar Shahbaz Khan
40	5	Sep 30	ER modeling	PDBM: 3.0-3.3, 6.3-6.4	Exercise L5	Lecturer: Omar Shahbaz Khan
41	6	Oct 07	Normalization	PDBM: 6.2-6.4	Homework 2 (due on 14.10.2024)	Lecturer: Eleni Tzirita Zacharou
42		Oct 14	<b>Semester Break</b>			
43	7	Oct 21	Storage hierarchy; Physical database design; Indexing	PDBM: 12.1, 12.2, 12.3.1 - 12.3.7 + more (details below)	Exercise L7	Lecturer: Eleni Tzirita Zacharou
44	8	Oct 28	B-trees; Performance tuning; Access methods; Join implementation	PDBM: 12.3.8, 13.1	Exercise L8	Lecturer: Omar Shahbaz Khan
45	9	Nov 04	Storage Models; Architecture of a DBMS; Main memory DBMSs	PDBM: 2 + more (details below)	Homework 3 (due on 11.11.2024)	Lecturer: Martin Hentschel
46	10	Nov 11	Transaction Management; Transactions in main memory DBMSs	PDBM: 14 + more (details below)	Old exercises/exams/homeworks	Lecturer: Omar Shahbaz Khan
47	11	Nov 18	Scaling-out; NoSQL; Eventual consistency; CAP theorem	PDBM: 11 (+ optional papers)	Old exercises/exams/homeworks	Lecturer: Omar Shahbaz Khan
48	12	Nov 25	Big data analytics; Distributed computing frameworks	PDBM: 19.1-19.2, 19.4, 20.1-20.3	Homework 4 (due on 02.12.2024)	Lecturer: Martin Hentschel
49		Dec 02	Trial exam			Online
50		Dec 13	<b>Final Exam</b>			

- BSc. vs MSc.
  - Pretty much the same schedule
  - One different question (5%) on the exam

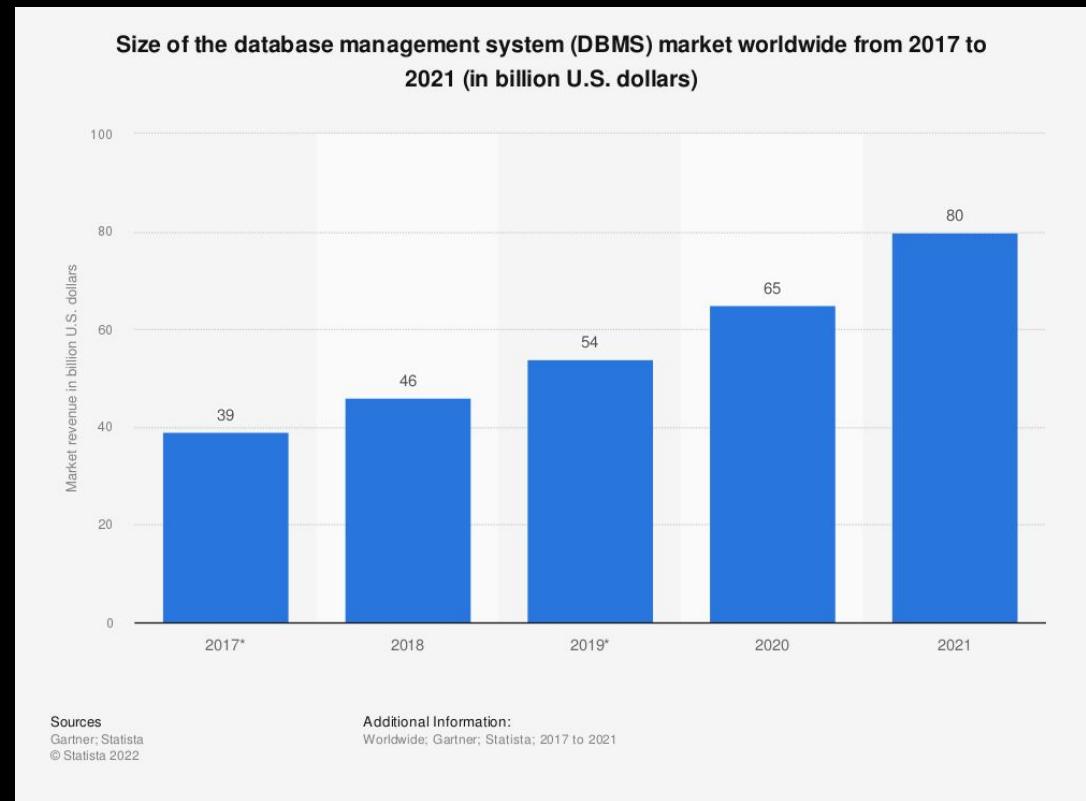
# Course Schedule (Illustrated)



Week	Lecture Date	Lecture Topic	Reading	Exercises / Homeworks	Note
36	1 Sep 08	Introduction to Database Systems, Entity-relationship model	PODM: 1, 6.1, 7.1-7.2	Exercise 1	Lecturer: Omar Shahbaz Khan
37	2 Sep 09	SQL DDL/DML: Basic SQL queries; joins; Aggregations	PODM: 7.3	Exercise 2	Lecturer: Eleni Tsirita Zacharatos
38	3 Sep 16	Complex SQL queries; Subqueries; Views	PODM: 7.3 - 7.4	Homework 1 (due on 23.09.2024)	
39	4 Sep 23	Triggers; Transactions; Using SQL from Python	PODM: 9.2, 14, 14.2.1, 14.5 + more (details below)	Exercise 3	Lecturer: Eleni Tsirita Zacharatos
40	5 Sep 30	Performance Tuning; Indexes	PODM: 12.3, 12.3.1, 12.3.4-6, 12.4, 12.5, 12.6, 12.7	Homework 2 (due on 14.10.2024)	Lecturer: Omar Shahbaz Khan
41	6 Oct 07	Normalization	PODM: 6.2-6.4	Homework 3 (due on 20.10.2024)	Lecturer: Eleni Tsirita Zacharatos
42	7 Oct 14	Semester Break	PODM: 6.2-6.4	Homework 4 (due on 05.11.2024)	Lecturer: Eleni Tsirita Zacharatos
43	8 Oct 21	Relational theory; Physical database design; Indexing	PODM: 12.1-12.2, 12.3.1 - 12.3.7 + more (details below)	Exercise 4	Lecturer: Eleni Tsirita Zacharatos
44	9 Oct 28	B-trees; Performance tuning; Access methods; Join implementation	PODM: 12.3.8, 13.1	Exercise 5	Lecturer: Omar Shahbaz Khan
45	9 Nov 04	Storage Models; Architecture of a DBMS; Main memory DBMs	PODM: 2 + more (details below)	Homework 3 (due on 11.11.2024)	Lecturer: Martin Hentschel
46	10 Nov 11	Transaction Management; Transactions in main memory DBMs	PODM: 14 + more (details below)	Old exercises/exams/homeworks	Lecturer: Omar Shahbaz Khan
47	11 Nov 18	Distributed DBMS; Logical consistency; CAP theorem	PODM: 15 + more (details below)	Old exercises/exams/homeworks	Lecturer: Omar Shahbaz Khan
48	12 Nov 25	Huge data analytics; Distributed computing frameworks	PODM: 19.1-19.2, 19.4, 20.1-20.3	Homework 4 (due on 05.12.2024)	Lecturer: Martin Hentschel
49	Dec 02	Final exam			Online
50	Dec 13	Final Exam			

# Why is it important?

- Crucial to effectively manage and utilize data
- Help maintain data integrity and security
- Ease app development



<https://www.statista.com/statistics/724611/worldwide-database-market/>

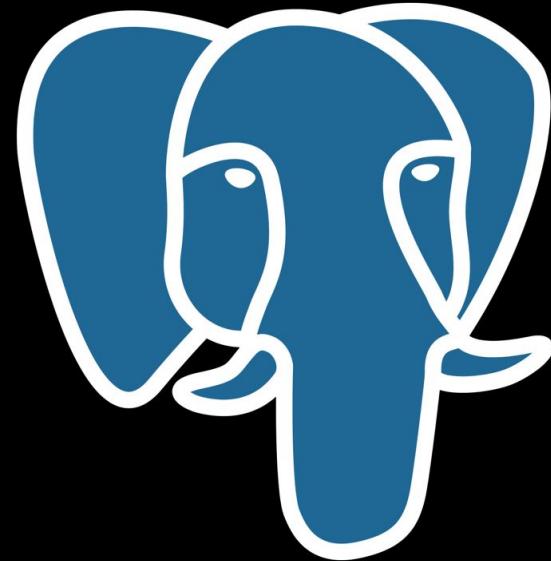
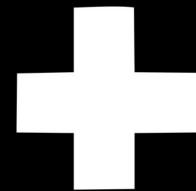
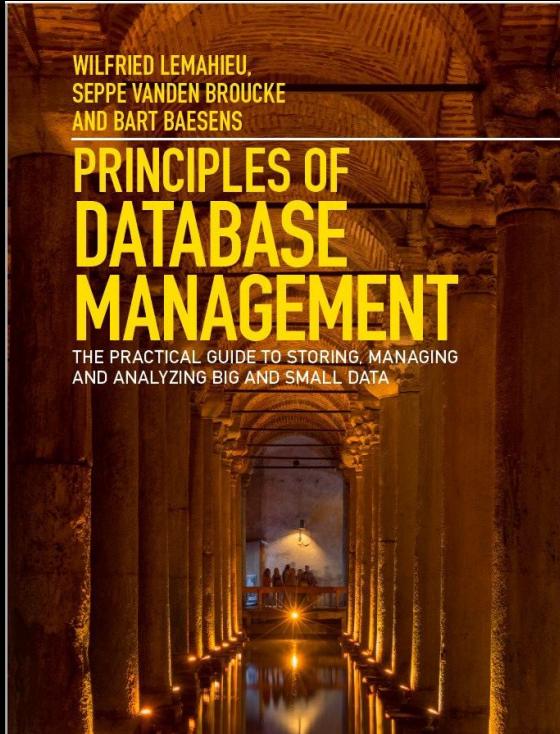
# Course Structure

- Lectures
  - Mondays 12:15 - 14:00 in Aud 1
  - *Preparation required: reading material, watching videos*
- Exercises
  - Mondays 14:15 - 16:00 in 3A54, 3A52, and 4A14-16.
  - If all rooms are full we also have 3A18.
  - *Preparation not required (related to previous lecture)*
- Homeworks
  - 4 Homeworks (deadlines published on learnIT)
  - Mandatory (3 out of 4), yet easy to get accepted!
  - *Feedback in the following weeks (if submitted on time)*
- LearnIT: course outline, materials, announcements, ...
- Piazza: Q&A, messages, and updates
  - *Ask consistently throughout the semester*
  - *Help your peers!*

# Course Methodology

- You learn: We are here to help you!
- You need to read the book beforehand
  - Yes, we often assume you have done so
  - All readings are in the schedule on learnIT
  - In some weeks, there can be some video recording for you to watch *before* the lecture
- We work in a pull model fashion: ask questions!

# Book and Database System



PostgreSQL  
(pgAdmin + psql)

# How will we assess your learning?

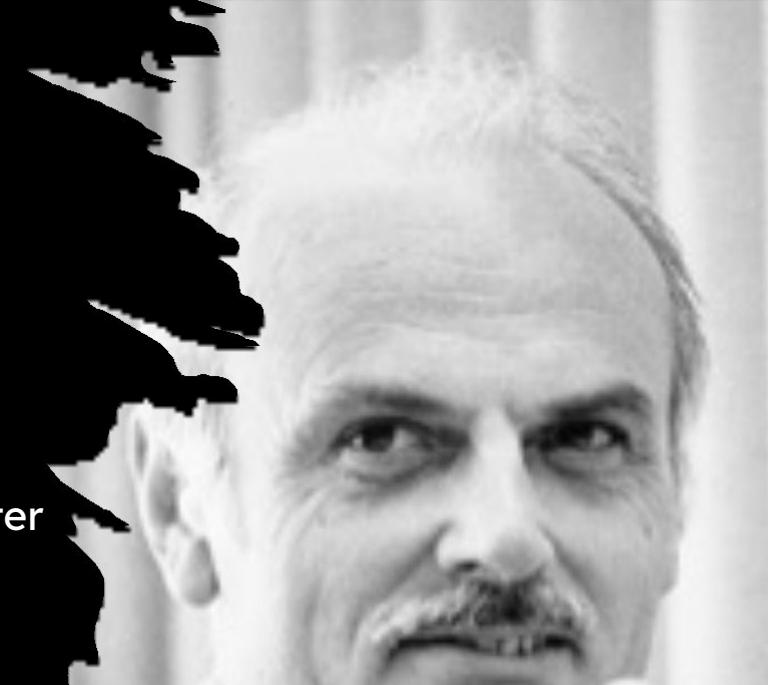
- 100% Exam (Quiz on LearnIT)
  - Restricted: no Internet access!
  - All course materials allowed offline
  - Communication is not allowed
- Exercises and homeworks will prepare you!

# Profile of the Week

# Edgar F. Codd

Father of Databases (*Relational Model*)

- 1923: Born 23/8, Isle of Portland, England
- 1965: PhD in CS from University of Michigan
- 1967: Moved to IBM Almaden Research Center
- 1969: Invented the relational model
- 1976: IBM Fellow
- 1981: Turing Award
- 1994: ACM Fellow



*Information Retrieval*

P. BAXENDALE, Editor

---

## A Relational Model of Data for Large Shared Data Banks

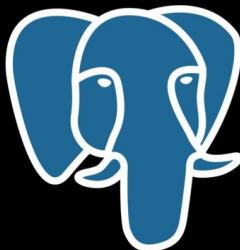
E. F. CODD

*IBM Research Laboratory, San Jose, California*

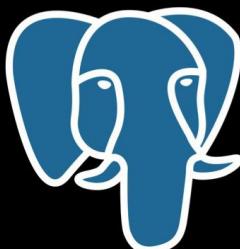
The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] presently in vogue for non-inferential systems. It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representa-

# DBMS Brief Introduction

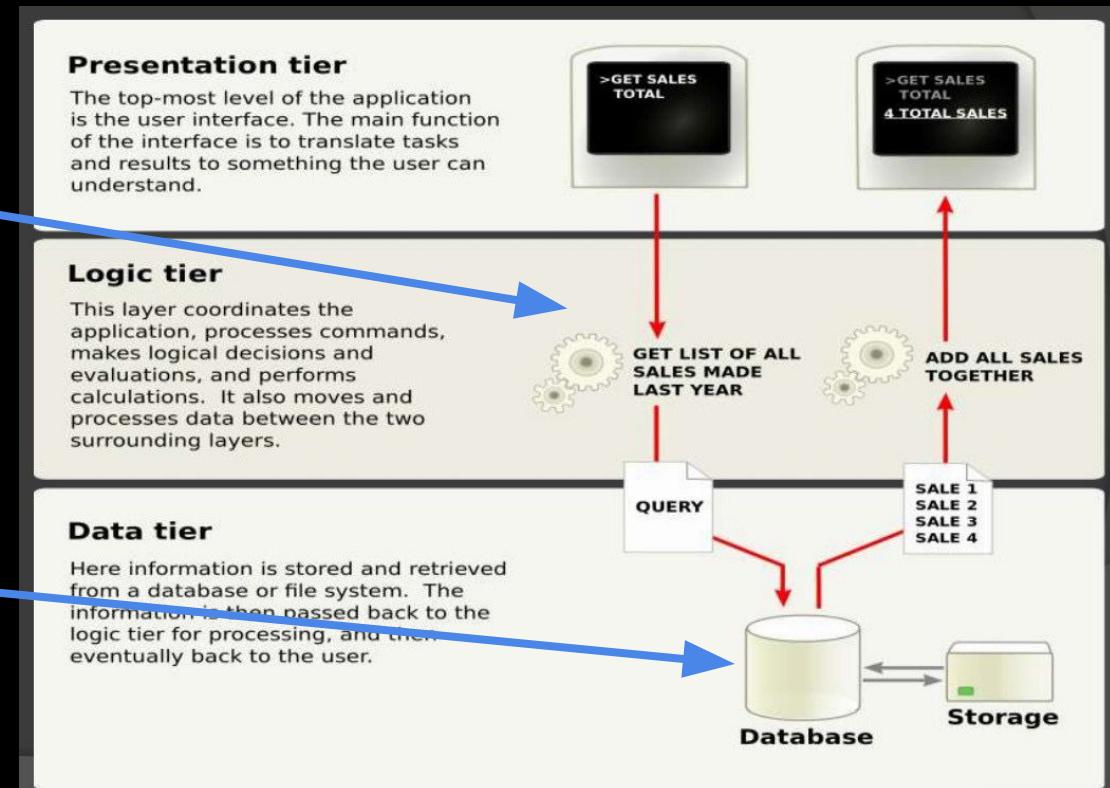
# Three-Layer Architecture



pgAdmin + psql



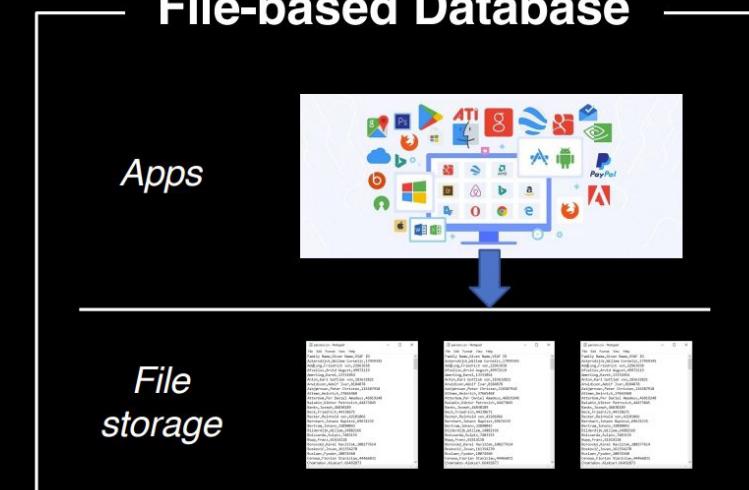
PostgreSQL



# Database Definition

- A database is a collection of related data items within a specific business process or problem setting
  - A database system provides a way to systematically organize, store, retrieve, and query a database
- 
- **Cons:**
    - Redundant data
    - Inconsistent data
    - **Data dependency**
    - Limited concurrency
    - Expensive maintenance
    - Poor performance

## File-based Database

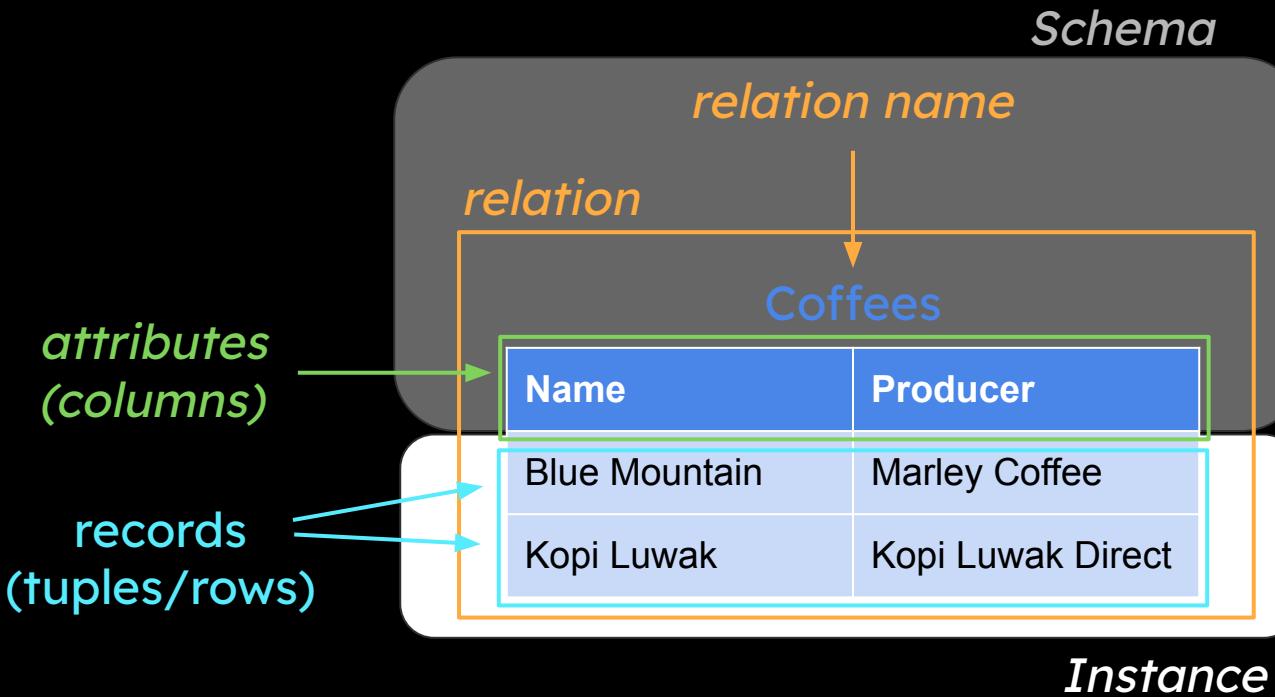


# Relational Database Definition

- A relational database is a type of database that is based on the relational model
  - stores data in a set of tables with rows and columns (a.k.a. relations)
  - uses relationships between these tables to manage the data.
- A relational database system (RDBMS) implements and manages relational databases.
- Pros:
  - Unique data entities
  - Data integrity
  - **Data independence**
  - High concurrency
  - Cheap maintenance
  - High performance

# Relational Model

# Basic Concepts



# Schema vs Instance vs Database

- Relation
  - Schema = name + list of attributes
    - Optional: attribute types
    - **Coffees** (Name, Producer)
    - **Coffees** (Name : STRING, Producer : STRING)
  - Instance
    - Records in a relation
      - E.g., (Blue Mountain, Marley Coffee)
- A Database is a collection of relations
  - Database schema = set of all relations names in the database
  - Database instance = set of all relations instances in the database

# Example of a Database Schema

- **Students** (SID : INT, Name : STRING, Email : STRING, Semester : INT)
- **Faculty** (FID : INT, Name : STRING, DID : INT)
- **Courses** (CID : STRING, Name : STRING, DID : INT)
- **Departments** (DID : INT, Name : STRING)
- **Transcripts** (CID : STRING, SID : INT, Grade : STRING, Comment : STRING)

# Why Relations?

## Pros

- Very simple model
- How we typically think about structured data
- Conceptual model behind SQL, which is the most important query language today

## Cons

- Too simple for some things:
  - hierarchical multi-valued data
- Much of today's data is NOT structured
- It could be complex to implement well

# Identifiers

- **Students** (SID : INT, Name : STRING, Email : STRING, Semester : INT)
- **Faculty** (FID : INT, Name : STRING, DID : INT)
- **Courses** (CID : STRING, Name : STRING, DID : INT)
- **Departments** (DID : INT, Name : STRING)
- **Transcripts** (CID : STRING, SID : INT, Grade : STRING, Comment : STRING)

# Keys and Superkeys

- What is a key?
  - Defines unique records (instances)
  - Helps in setting relationships between relations
  - Ensures the mathematical definition of a relation (*set* of records)
- Superkeys
  - Is a set of attributes that uniquely identifies records: *Uniqueness* property
  - The entire set of attributes of a relation is a superkey
  - Minimal superkey: *Minimality* property
    - No attributes can be removed from a superkey without violating the uniqueness property

Students				
	key	SID	Name	Email
		01789	Bob Brown	bobr@itu.dk
	superkey	01883	Lucas White	luwh@itu.dk

# Candidate Keys

- Attributes that satisfy the uniqueness and minimality properties
  - Minimal superkey = (candidate) key
  - Superkeys contain at least one (candidate) key
  - A relation can have many (candidate) keys

Students

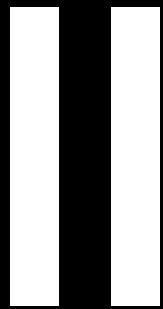
key	SID	Name	Email	Semester
superkey	01789	Bob Brown	bobr@itu.dk	2
	01883	Lucas White	luwh@itu.dk	5

# Primary Keys

- A key to identify records in a relation
  - Important to define indexes and for storage purposes (later in the course)
  - Cannot be null
  - Also used to establish relationships with other relations
  - From all candidate keys only one can be the Primary Key
    - The remaining ones are known as *Alternative Keys*

Students

key	SID	Name	Email	Semester
superkey	01789	Bob Brown	bobr@itu.dk	2
	01883	Lucas White	luwh@itu.dk	5



# Live Exercise

- What are superkeys and (candidate) keys?

- (SID)
- (Email)
- (SID, Name)
- (Semester)
- (Email, Semester)
- (Name)
- (Name, Semester)

- What is the best key for being the primary key?

- Which of these keys does not make sense in practice?

SID	Name	Email	Semester
01789	Bob Brown	bobr@itu.dk	2
01883	Lucas White	luwh@itu.dk	5
01234	Olga Marx	olma@itu.dk	6
04321	Jens Schuh	jesc@itu.dk	1
01439	Olga Marx	olmr@itu.dk	2
01832	Peter Pitt	pepi@itu.dk	1
02378	Line Dunn	lidu@itu.dk	3

# Live Exercise

- What are superkeys and (candidate) keys?
  - (SID)
  - (Email)
  - (SID, Name)
  - (Semester)
  - (Email, Semester)
  - (Name)
  - (Name, Semester)
  - (Candidate) Key
  - (Candidate) Key
  - Superkey
  - None
  - Superkey
  - None
  - (Candidate) Key
- What is the best key for being the primary key?
- Which of these keys does not make sense in practice?

SID	Name	Email	Semester
01789	Bob Brown	bobr@itu.dk	2
01883	Lucas White	luwh@itu.dk	5
01234	Olga Marx	olma@itu.dk	6
04321	Jens Schuh	jesc@itu.dk	1
01439	Olga Marx	olmr@itu.dk	2
01832	Peter Pitt	pepi@itu.dk	1
02378	Line Dunn	lidu@itu.dk	3



# Live Exercise

- What are superkeys and (candidate) keys?
  - (SID) -- (Candidate) Key
  - (Email) -- (Candidate) Key
  - (SID, Name) -- Superkey
  - (Semester) -- None
  - (Email, Semester) -- Superkey
  - (Name) -- None
  - (Name, Semester) -- (Candidate) Key
- What is the best key for being the primary key?
  - (SID) is ideal for being the Primary Key
- Which of these keys does not make sense in practice?
  - (Name, Semester)

SID	Name	Email	Semester
01789	Bob Brown	bobr@itu.dk	2
01883	Lucas White	luwh@itu.dk	5
01234	Olga Marx	olma@itu.dk	6
04321	Jens Schuh	jesc@itu.dk	1
01439	Olga Marx	olmr@itu.dk	2
01832	Peter Pitt	pepi@itu.dk	1
02378	Line Dunn	lidu@itu.dk	3



# Relationships

- **Students** (SID : INT, Name : STRING, Email : STRING, Semester : INT)
  - **Faculty** (FID : INT, Name : STRING, DID : INT)
  - **Courses** (CID : STRING, Name : STRING, DID : INT)
  - **Departments** (DID : INT, Name : STRING)
  - **Transcripts** (CID : STRING, SID : INT, Grade : STRING, Comment : STRING)
- 
- The diagram illustrates the relationships between the Faculty, Courses, and Departments relations. It shows three green ovals, each containing the key attribute DID from one of the relations. Three green arrows originate from these ovals and point to the DID attribute in the Courses relation, indicating that Faculty, Courses, and Departments are all related to Courses through their common key attribute.

# Foreign Keys

- Defines the relationship between relations
- A key FK in a relation R is a foreign key iff:
  - The attributes in FK matches a primary key PK of a relation S and they are of the same type
  - Any record  $i$  in R has a value in FK that either
    - occurs as a value of PK for some record  $j$  in S, or
    - is null
  - I.e., FK = PK (domain and values)
- A relation can have several foreign keys

# Foreign Keys: Example

**Departments**

PK	DID	Name
1		Business
2		Computer Science

**Courses**

PK	CID	Name	DID	FK
MATH		Bob Brown	2	
I2DBS		Intro. To DB Syst.	2	
SWE		Soft. Engineering	4	

**Students**

PK	SID	Name	Email	Semester
01789	Bob Brown	bobr@itu.dk	2	
01883	Lucas White	luwh@itu.dk	5	
01234	Olga Marx	olma@itu.dk	6	
...	...	...	...	...

**Transcripts**

PK	CID	SID	Grade	Comment	Foreign Relation
I2DBS	01789	7	Student didn't...		
MATH	01234	12	Clear display of...		

# Foreign Keys: Example

**Departments**

PK DID

DID	Name
1	Business
2	Computer Science

**Courses**

PK CID

FK DID

CID	Name	DID
MATH	Bob Brown	2
I2DBS	Intro. To DB Syst.	2
SWE	Soft. Engineering	4

**Students**

PK SID

SID	Name	Email	Semester
01789	Bob Brown	bobr@itu.dk	2
01883	Lucas White	luwh@itu.dk	5
01234	Olga Marx	olma@itu.dk	6
...	...	...	...

**Transcripts**

FK CID

FK SID

Foreign Relation

CID	SID	Grade	Comment
I2DBS	01789	7	Student didn't...
MATH	01234	12	Clear display of...

# All keys in a Relation

## Superkeys

### Minimal Superkeys / (Candidate) Keys

Primary Key

Foreign Keys

Alternative Keys

**NOTE:**  
Keys are part of the schema

# So far, all together

*Primary Key = (Candidate) Key = Minimal Superkey*

The diagram illustrates the **Students** relation. It is divided into two main sections: **Schema** and **Instance**. The **Schema** section shows the relation name **Students** above a table with four columns: **SID**, **Name**, **Email**, and **Semester**. The **Instance** section shows actual data rows. A green bracket labeled "Attributes" points to the columns. A purple bracket labeled "Key Relation" points to the entire table. A red bracket labeled "Primary Key = (Candidate) Key = Minimal Superkey" points to the **SID** column, which is highlighted with a green oval.

SID	Name	Email	Semester
01789	Bob Brown	bobr@itu.dk	2
01883	Lucas White	luwh@itu.dk	5
01234	Olga Marx	olma@itu.dk	6
...	...	...	...

The diagram illustrates the **Transcripts** relation. It is divided into several sections: **Relation name** (labeled **Transcripts**), **Superkey** (highlighted in cyan), **Foreign Keys** (highlighted in green), **Foreign Relation** (highlighted in purple), and **Records** (the data rows). The relation has four columns: **CID**, **SID**, **Grade**, and **Comment**. The **CID** and **SID** columns are highlighted with green ovals. The **SID** column is also highlighted with a cyan oval, indicating it is a foreign key. The **Grade** and **Comment** columns are highlighted with cyan outlines. A blue arrow labeled "Relation" points to the first row. A green arrow labeled "Records" points to the second row.

CID	SID	Grade	Comment
I2DBS	01789	7	Student didn't...
MATH	01234	12	Clear display of...

# Integrity Constraints

- An integrity constraint (IC) is a limitation of the allowed content (or development) of a database
  - Ensures that the data is always correct and consistent
  - There exist various ICs
- It is the RDBMS that takes care of ensuring the ICs in a database
- ICs already seen so far
  - Domain constraint
  - Key constraint
  - Entity constraint (PK)
    - attribute type and format (e.g., DATE)
    - uniqueness & minimality
    - NOT NULL
    - PK = FK
  - Referential constraint (FK)
- More advanced ICs
  - Functional dependencies
  - Temporal constraint...

# SQL

# Structured Query Language (SQL)

- SEQUEL if you worked for IBM in the 80s
- SQL is primarily a query language, for getting information from a database (DML)
  - also includes a data-definition component for describing database schemas (DDL)
- Invented in the 70s by IBM
- The three most common commands in SQL queries
  - SELECT, FROM, WHERE

```
SELECT * FROM Students WHERE Name = 'Lucas White';
```

# SQL

- **Data Definition Language (DDL)**

- Used by the database administrator (DBA) to define the database's data model
- Three common commands:
  - CREATE TABLE, ALTER TABLE, and DROP TABLE

*Today's Focus*

- **Data Manipulation Language (DML)**

- Used by applications and users to retrieve, insert, modify, and delete records
- Four statements:
  - SELECT, INSERT, UPDATE, and DELETE

# First Normal Form (1NF)

- Each attribute in a relation has:
  - a primitive type (atomic values), and;
  - a unique name
- The main goal is to eliminate redundant data definition
- Benefits:
  - Data integrity
  - Data consistency
  - Easy data manipulation
  - Better data organization

# Data Types

Type	Description
CHAR(n)	Fixed-length string of size <i>n</i>
VARCHAR(n)	Variable-length string of maximum size <i>n</i>
SMALLINT	Small integer (-32,768 and 32,767)
INT	Integer (-2,147,483,648 and 2,147,483,647)
FLOATAT(n, d)	Small number with a floating decimal point: n = max digits and d = max decimals
DOUBLE(n, d)	Large number with a floating decimal point: n = max digits and d = max decimals
DATE	Date in format YYYY-MM-DD
DATETIME	Date and time in format YYYY-MM-DD HH:MI:SS
TIME	Time in format HH:MI:SS
BOOLEAN	True or false
BLOB	Binary large object (typically unstructured)

**NOTE:**

Check PostgreSQL types here:

<https://www.postgresql.org/docs/current/datatype.html>

# Create Relation Students

Students (SID : INT, Name : STRING, Email : STRING, Semester : INT)

```
CREATE TABLE Students (
    SID INT,
    Name VARCHAR(255),
    Email CHAR(11),
    Semester INT
);
```

# Define a Primary Key

Students (SID : INT, Name : STRING, Email : STRING, Semester : INT)

```
ALTER TABLE Students ADD PRIMARY KEY (SID);
```

```
CREATE TABLE Students (
    SID INT PRIMARY KEY,
    Name VARCHAR(255),
    Email CHAR(11),
    Semester INT
);
```

**Courses** (CID : STRING, Name : STRING, DID : INT)  
**Departments** (DID : INT, Name : STRING)

# Define a Foreign Key

```
CREATE TABLE Departments (
    DID INT PRIMARY KEY,
    Name VARCHAR(255)
);
```

```
CREATE TABLE Courses (
    CID VARCHAR(25) PRIMARY KEY,
    Name VARCHAR(255),
    DID INT,
    FOREIGN KEY (DID) REFERENCES Departments (DID)
);
```

## ALTERNATIVE

```
CREATE TABLE Courses (
    CID VARCHAR(25) PRIMARY KEY,
    Name VARCHAR(255),
    DID INT REFERENCES Departments (DID)
);
```

# Multiple-Attribute Primary Keys

Students (SID : INT, Name : STRING, Email : STRING, Semester : INT)

Courses (CID : STRING, Name : STRING, DID : INT)

Transcripts (CID : STRING, SID : INT, Grade : STRING, Comment : STRING)

```
CREATE TABLE Transcripts (
    CID VARCHAR(25),
    SID INT,
    Grade INT,
    Comment VARCHAR(255),
    PRIMARY KEY (CID, SID),
    FOREIGN KEY (CID) REFERENCES Courses (CID),
    FOREIGN KEY (SID) REFERENCES Students (SID)
);
```

# NULL

- What if a value is missing?
  - Does not exist?
  - Unknown?
  - Secret?
- SQL solution: NULL
  - NULL = no value
  - More next week...
- By default, attributes can be NULL
  - Except: PRIMARY KEY attributes
  - Except: NOT NULL attributes
- Allowing NULL values is a design decision!

# Transcripts Relation with NOT NULL

Students (SID : INT, Name : STRING, Email : STRING, Semester : INT)

Courses (CID : STRING, Name : STRING, DID : INT)

Transcripts (CID : STRING, SID : INT, Grade : STRING, Comment : STRING)

```
CREATE TABLE Transcripts (
    CID VARCHAR(25) NOT NULL,
    SID INT NOT NULL,
    Grade INT NOT NULL,
    Comment VARCHAR(255),
    PRIMARY KEY (CID, SID),
    FOREIGN KEY (CID) REFERENCES Courses (CID),
    FOREIGN KEY (SID) REFERENCES Students (SID)
);
```

# Drop All Created Relations

```
DROP TABLE Transcripts;  
DROP TABLE Courses;  
DROP TABLE Departments;  
DROP TABLE Students;
```

**ORDER MATTERS!!!**

Drop Foreign Relations before Key Relations

# Takeaways

Can't wait for the SQL!

## Relational Model

- Relations, attributes, keys, primary & foreign keys...

## SQL DDL = Data Definition Language

- CREATE TABLE, DROP TABLE, ALTER TABLE, ...
- Allows to create complex schemas and maintain them

## SQL DML = Data Manipulation Language

- INSERT, DELETE, UPDATE, SELECT
- Simple set of commands for complicated selections
- *(we will dive into it next week)*

# Next Time in IDBS...

# Introduction to Database Systems

## IDBS - Fall 2024

### Lecture 2 - SQL

---

SQL DML

Basic SQL queries

Joins

Aggregations

---

Readings: PDBM 7.3

Eleni Tzirita Zacharatou

# Introduction to Database Systems

## IDBS – Fall 2024

- Week 2:
- SQL Select
- SQL Joins
- SQL Aggregations

---

Eleni Tzirita Zacharatou

Readings:

PDBM 7.3

- **Data Definition Language (DDL)**
  - Used by the database administrator (DBA) to define the database's data model
  - Three common commands:
    - CREATE TABLE, ALTER TABLE, and DROP TABLE
  
- **Data Manipulation Language (DML)**
  - Used by applications and users to retrieve, insert, modify, and delete records
  - Four statements:
    - SELECT, INSERT, UPDATE, and DELETE

*Today's focus*

# Database Schema

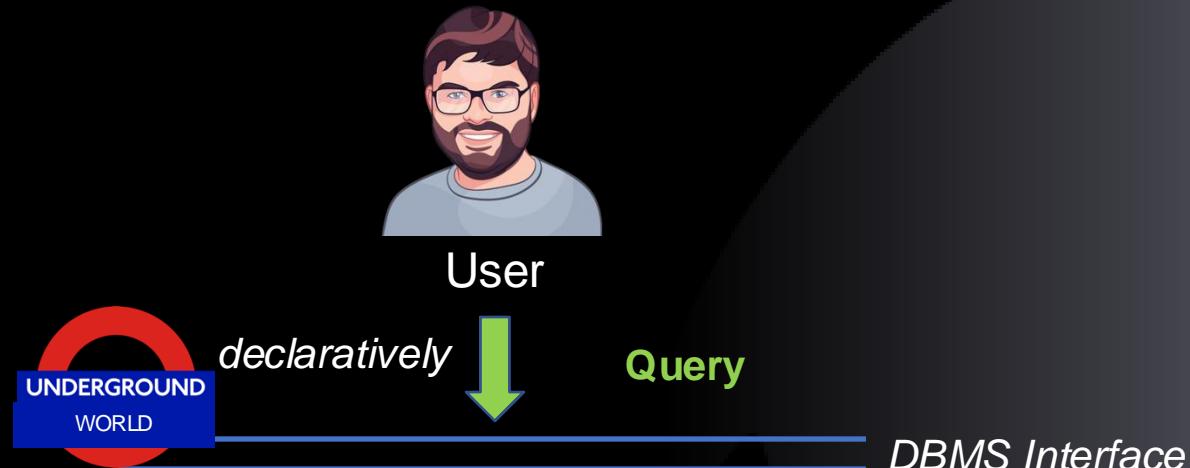
---

## Running Example

- Coffees(name, manufacturer)
- Coffeehouses(name, address, license)
- Drinkers(name, address, phone)
- Likes(drinker, coffee)
- Sells(coffeehouse, coffee, price)
- Frequents(drinker, coffeehouse)

### Note

Keys are underlined



# SQL DML -- Select

Readings:  
PDBM 1

# Rationale & Example

- **SELECT** -- desired attributes  
**FROM** -- one or more relations  
**WHERE** -- condition about records of the relations
- Which coffees are made by Ottolina?

Coffees	
name	manufacturer

```
sql
SELECT name FROM Coffees
WHERE manufacturer = 'Ottolina';
```



name
Fortissima
Maracaibo
Buongiorno

# The Same but Graphically

Coffees	
name	manufacturer
...	...
...	...
Maracaibo	Ottolina
...	...

```
sql  
SELECT name FROM Coffees  
WHERE manufacturer = 'Ottolina';
```

*iterate*

*filter*  
*Check if Ottolina*  
*If so, project name*

name
Fortissima
Maracaibo

Note

SQL uses bag semantics

Allows for duplicates

# Star in SELECT

- What if the **SELECT** clause has \* instead?

Coffees	
name	manufacturer

```
sql
SELECT * FROM Coffees
WHERE manufacturer = 'Ottolina';
```

name	manufacturer
Fortissima	Ottolina
Maracaibo	Ottolina
Buongiorno	Ottolina

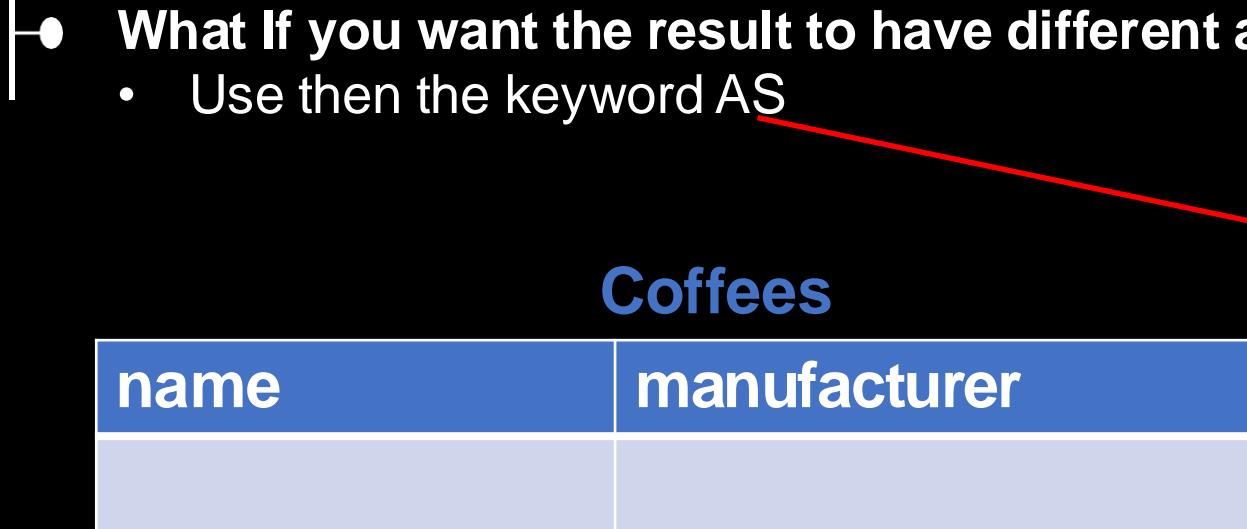
- This selects all attributes from the Coffees relation

# Attribute Renaming

- What If you want the result to have different attribute name?
  - Use then the keyword AS

Coffees	
name	manufacturer

```
sql
SELECT name AS coffee, manufacturer FROM Coffees
WHERE manufacturer = 'Ottolina';
```



coffee	manufacturer
Fortissima	Ottolina
Maracaibo	Ottolina
Buongiorno	Ottolina

# Expressions in SELECT

- Any expression that makes sense can appear as an element of a SELECT clause.

```
SELECT coffeehouse, coffee, price * 0.965 AS priceInYen FROM Sells;
```

Sells

coffeehouse	coffee	price

coffeehouse	coffee	priceInYen
Sue's	Fortissima	342
Joe's	Maracaibo	285
Bob's	Buongiorno	149

# Constants as Expressions in SELECT

- Any constant as expression (example with the Likes relation)

Likes		sql	
drinker	coffee	SELECT drinker, 'likes Fortissima' AS whoLikesFort FROM Likes WHERE coffee = 'Fortissima';	
Sally			
Fred			

A red oval highlights the string 'likes Fortissima' in the SQL query. An orange arrow points from this highlighted text down to the resulting table, indicating its mapping to the 'whoLikesFort' column.

drinker	whoLikesFort
Sally	Likes Fortissima
Fred	Likes Fortissima

# Live Exercise

- What should the following queries return?
  - Assume the relation Coffees has two records only

coffee	manufacturer
Fortissima	Ottolina
Maracaibo	Ottolina

coffee	manufacturer
Fortissima	Ottolina
Maracaibo	Ottolina

sql

```
SELECT *  
FROM Coffees;
```

Query
42
42

sql

```
SELECT 42 AS Query  
FROM Coffees
```

Query
42

sql

```
SELECT 42 AS Query;
```

# Practice

---

- Analog has a 50% discount on a second coffee of the same type. For each coffee sold there, show the price of two coffees ('priceOfTwo').
- A coffeehouse owner by the name of “Eleni” has passed out with caffeine shock. Write a query to find her home phone number.

# Conditions in WHERE

- Comparisons **=, <>, <, >, <=, >=.**
  - Many other operators that produce boolean-valued results.
- Boolean operators **AND, OR, NOT.**
- Find the price Joe's coffeehouse charges for Maracaibo in Sells

Sells

coffeehouse	coffee	price

sql

```
SELECT price
FROM Sells
WHERE coffeehouse = 'Joe''s' AND coffee = 'Maracaibo';
```

# Patterns

---

- A condition can compare a string to a pattern by:
  - <Attribute> LIKE <pattern>
  - <Attribute> NOT LIKE <pattern>
- **Pattern is a quoted string with**
  - % = “any string”
  - \_ = “any character”
- **Escaping characters**
  - “\” default escape character
  - ESCAPE modifier
- **PostgreSQL supports regular expressions**
  - These are much more expressive!

sql

```
SELECT * FROM Coffeehouses WHERE address LIKE '%1';
```

sql

```
SELECT * FROM Coffeehouses WHERE address LIKE '%!_1' ESCAPE '!';
```

# Practice

---

- A coffeehouse patron by the last name of “Sivertsen” has passed out, again from caffeine shock. Show a query to find his home phone.
- Someone has recommended a coffeehouse that sells a coffee called “Blue”-something, that costs more than 100. Unfortunately, she doesn’t remember the name of the coffeehouse. Find where that coffee is sold at such a high price.

# Sorting

---

- **SQL assumes bags semantics**
  - No order assumption
- **ORDER BY sorts the results**
  - Ascending (ASC) or descending (DESC)

```
sql
SELECT coffeehouse, price
FROM Sells
ORDER BY coffeehouse, price DESC;
```

- **When should we use ORDER BY?**

QUESTION



User



*declaratively*

Query

*DBMS Interface*

# SQL DML -- Joins

Readings:

PDBM 1

# Multi-Relation Queries

---

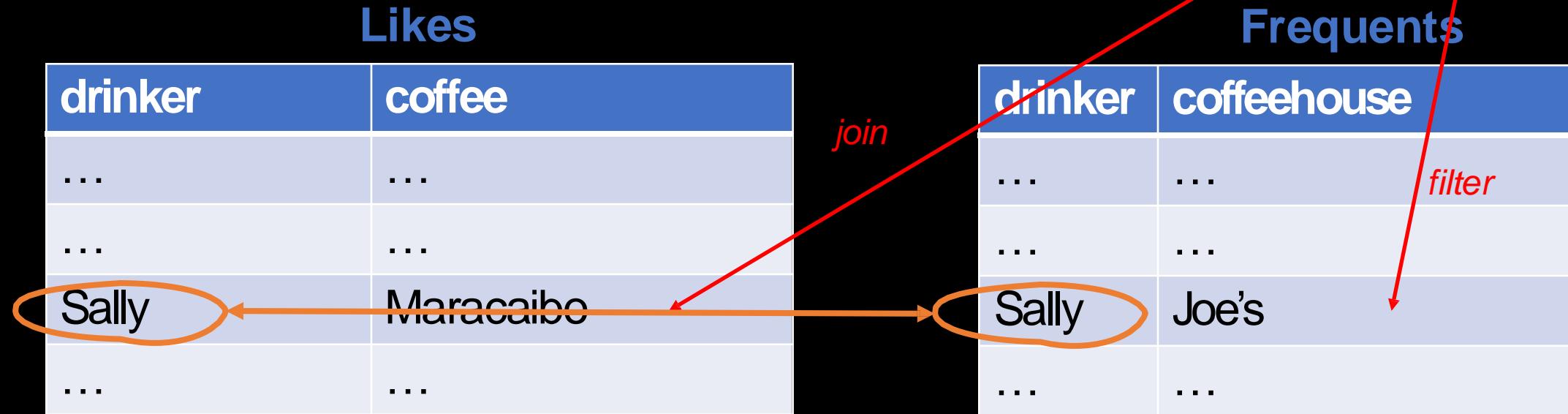
- Most typically, queries combine data from more than one relation.
- Several relations in the FROM clause.
- Distinguish attributes of the same name by “<relation>.<attribute>”
- Find the coffees liked by at least one person who frequents Joe’s.

sql

```
SELECT coffee
FROM Likes
JOIN Frequent
ON Likes.drinker = Frequent.drinker
WHERE Frequent.coffeehouse = 'Joe''s';
```

# The Same but Graphically

```
sql  
  
SELECT coffee  
FROM Likes  
JOIN Frequent  
ON Likes.drinker = Frequent.drinker  
WHERE Frequent.coffeehouse = 'Joe''s';
```



# Join Semantics

- Start with the product of all relations in the **FROM clause**
  - Imagine one tuple-variable for each relation in the FROM clause.
  - Think of nested for-loops
- Apply the selection conditions from the **JOIN clauses**
  - Usually F.foreign\_key = K.key
- Apply the selection condition from the **WHERE clause**
- Apply the selection Project onto the list of attributes and expressions in the **SELECT clause**.

sql

```
SELECT coffee
  FROM Likes
  JOIN Frequent
    ON Likes.drinker = Frequent.drinker
   WHERE Frequent.coffeehouse = 'Joe''s';
```

Likes

drinker	coffee
...	...
Sally	Maracaibo

Frequent

drinker	coffeehouse
...	...
Sally	Joe's

# Renaming Relations

sql

```
SELECT coffee
FROM Likes
JOIN Frequent
ON Likes.drinker = Frequent.drinker
WHERE Frequent.coffeehouse = 'Joe''s';
```

sql

```
SELECT L.coffee
FROM Likes L
JOIN Frequent F
ON L.drinker = F.drinker
WHERE F.coffeehouse = 'Joe''s';
```



# Practice

---

- For each person that “frequents” some coffeehouse, show the name of the person and the address of the coffeehouse.
- For each person that “frequents” some coffeehouse, show the address of the person and the address of the coffeehouse.
- For each person that “frequents” some coffeehouse, show the name of the person and the address of the coffeehouse.
  - Why don’t we need the Drinkers relation?
  - What if we want only the names of drinkers and coffeehouses?
  - What if we used IDs in our design, and wanted the names?
  - What if we want all drinkers, including those that don’t frequent any coffeehouse?
  - What if we only want the name of drinkers?

# Select + Join

---

- **SELECT** -- desired attributes  
**FROM** -- one or more relations  
**WHERE** -- condition about records of the relations  
**JOIN** -- connect records between relations

# Duplicate Elimination

- Force the result to be a set with **SELECT DISTINCT ...**
  - Can also do this with GROUP BY – **but please don't!**
- From **Sells(coffeehouse, coffee, price)**
  - find all the different prices charged for coffees:

## Note

Sorting =  $O(n \log n)$

sql

```
SELECT DISTINCT price  
FROM Sells;
```

## Note

Without DISTINCT, each price would be listed as many times as there were coffeehouse/coffee pairs at that price

SQL



User



*declaratively*

Query

*DBMS Interface*

# SQL DML -- Aggregations

Readings:

PDBM 1

# Relation-based Aggregations

---

- **SUM, AVG, COUNT, MIN, and MAX can be applied to a column in a SELECT clause**
  - Produces that aggregation on the column
- **COUNT(\*) counts the number of tuples.**
- **Find the average price of Maracaibo**

sql

```
SELECT AVG(price)
FROM Sells
WHERE coffee = 'Maracaibo';
```

# Duplicate Elimination

---

- Use DISTINCT inside an aggregation.
- Find the number of different prices charged for Maracaibo

sql

```
SELECT COUNT(DISTINCT price)
FROM Sells
WHERE coffee = 'Maracaibo';
```

# Practice

---

- Show the highest price of any coffee

# Nulls in Aggregations

Sells	coffeehouse	coffee	price

- **NULL never contributes to a sum, average, or count, and can never be the minimum or maximum of a column.**
- **But if there are only NULL values in a column**
  - The result of the aggregation is NULL
- **Exception: COUNT of an empty set is 0.**

## Note

The number of coffeehouses  
at a known price!

sql

```
SELECT COUNT(coffeehouse)
FROM Sells
WHERE coffee = 'Maracaibo';
```

sql

```
SELECT COUNT(price)
FROM Sells
WHERE coffee = 'Maracaibo';
```

# Grouping-based Aggregations

- **SELECT-FROM-WHERE expression followed by GROUP BY**
- **The relation that results from the SELECT-FROM-WHERE is:**
  - grouped according to the values of the attributes in the GROUP BY clause
  - any aggregation is applied only within each group
- **How does the system perform GROUP BY?**
  - sort the relation OR hash it!
- **Find the average price for each coffee**

sql

```
SELECT coffee, AVG(price)  
FROM Sells  
GROUP BY coffee;
```

coffee	AVG(price)
Maracaibo	2.33
Fortissima	3.51

# Grouping-based Aggregations

- Find for each drinker the average price of Maracaibo at the coffeehouses they frequent

```
sql
SELECT drinker, AVG(price)
FROM Frequent F
JOIN Sells S ON F.coffeehouse = S.coffeehouse
WHERE coffee = 'Maracaibo'
GROUP BY drinker;
```

Frequent

drinker	coffeehouse

Sells

coffeehouse	coffee	price

# SELECT Attributes with Aggregations

---

- If any aggregation is used, then each element of the SELECT list must be either:
  - An attribute on the GROUP BY list
  - Aggregation – COUNT, AVG, MAX, ...
- All GROUP BY attributes must be in SELECT
  - Caveat: SQL standard vs. Implementation
    - Some systems are more flexible!
    - PostgreSQL allows omitting functionally dependent attributes (see week 6!)

# Example Query

---

Sells

coffeehouse	coffee	price

- For each coffeehouse, show the average price of all coffees in that coffeehouse.

sql

```
SELECT coffeehouse, AVG(price) AS average_price  
FROM Sells  
GROUP BY coffeehouse;
```



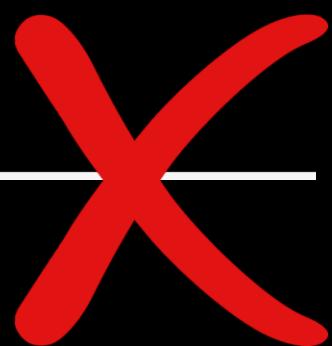
coffeehouse	average_price

# What About this One?

- Show the price of the most expensive coffee (from any coffeehouse) and the name of the coffeehouse that sells it

sql

```
SELECT coffeehouse, MAX(price)  
FROM Sells  
GROUP BY coffeehouse;
```



sql

```
SELECT coffeehouse, MAX(price)  
FROM Sells
```



sql

```
SELECT coffeehouse, price  
FROM Sells  
WHERE price = (SELECT MAX(price) FROM Sells);
```

# Practice

---

- For each coffeehouse, show the number of coffees sold in that coffeehouse and the average price.
- For each coffeehouse, show the price of the most expensive coffee sold in that coffeehouse.
- Show the name and price of the least expensive coffee (from any coffeehouse).

# HAVING Clause

- HAVING <condition> may follow a GROUP BY clause.
  - If so, the condition applies to each group, and groups not satisfying the condition are eliminated.
- Like WHERE but for groups!
- Find the average price of those coffees that are served in at least two coffeehouses

Sells

coffeehouse	coffee	price

```
sql
SELECT coffee, AVG(price)
FROM Sells
GROUP BY coffee
HAVING COUNT(coffeehouse) > 1
```

# HAVING Conditions

---

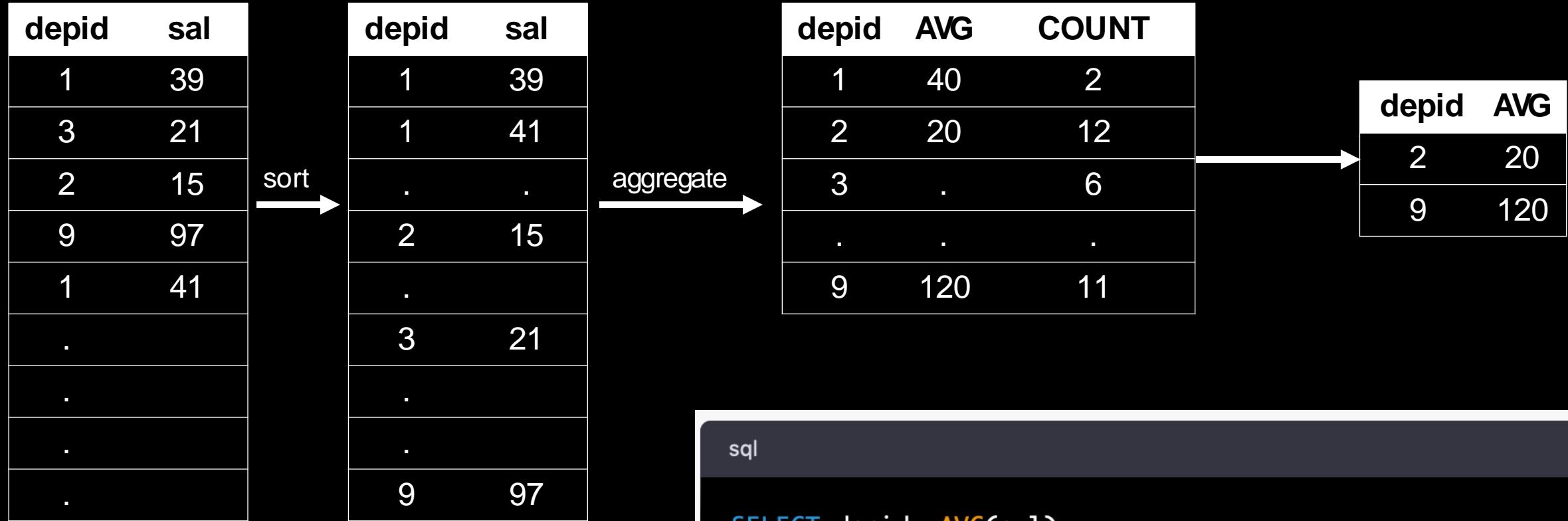
- Anything goes in a sub-query – later...
- Outside sub-queries, they may refer to attributes only if they are either
  - A grouping attribute, or
  - Aggregated(same condition as for SELECT clauses with aggregation)

# Practice

---

- For each coffeehouse that sells more than two coffees, show the number of coffees sold in that coffeehouse and their average price.
- For each coffeehouse, show the number of drinkers that frequent the coffeehouse.
- For each drinker that frequents more than one coffeehouse, show the number of coffeehouses that he/she frequents.

# GROUP BY + HAVING Implementation



sql

```
SELECT depid, AVG(sal)
FROM Emp
GROUP BY depid
HAVING COUNT(*) > 10
```



# Takeaways

- **SQL universal relational query language**
  - Result set (bag) is a table
- **1 Block**
  - select, from, where, group by, having, sort by clauses
- **Joins**
  - Combine data from many tables
- **Aggregations**
  - Group data
- **SQL is code**
  - Treat it accordingly



# What is next?

- **Next week: Advanced SQL**
- **Exercise 2 is out (SQL)**
  - Remember: No submission – but very important to do it!
  - It has some queries that you did not learn how to do yet – just think about potential solutions for now, we will cover them next week.
- **Homework 1 description is out**
  - SQL queries, some being more complex
  - PDF and SQL script are already available
  - Quiz on LearnIT will open at the end of the week
  - **For help:** Talk to TAs in exercise class
  - **Due Sep 23 at 23:59** – individual submissions!
  - **Remember:** We only accept and give feedback to HWs submitted before the deadline.
  - **Do the exercises before the homework!**

# Introduction to Database Systems

## IDBS - Fall 2024

### Week 3 - Advanced SQL

---

Division

JOINS and NULL

Set Operations

Subqueries

Views

---

Readings: PDBM 7.3-7.4

Eleni Tzirita Zacharatou

# Running Example 1

Same schema as in week 2

Script on LearnIT: [03-coffees-schema/queries.sql](#)

Coffees (name, manufacturer)

Coffeehouses (name, address, license)

Drinkers (name, address, phone)

Likes (drinker, coffee)

Sells (coffeehouses, coffee, price)

Frequents (drinker, coffeehouse)

# Running Example 2

Sports Database same schema as in week 2 exercises

Script on LearnIT: [03-sports-schema/queries.sql](#)

People (ID, name, gender, height)

Gender (gender, description)

Sports (ID, name, record)

Competitions (ID, place, held)

Results (peopleID, competitionID, sportID, result)

Last Time in IDBS...

# Last Time...

- SELECT - FROM - WHERE
- JOIN
- ORDER BY, DISTINCT, Aggregations
- GROUP BY + HAVING

# Last Time...

- SELECT - FROM - WHERE

```
SELECT name  
FROM Coffees;
```

- JOIN

```
SELECT name AS coffee  
FROM Coffees  
WHERE manufacturer LIKE 'Kopi%';
```

- ORDER BY, DISTINCT, Aggregations

- GROUP BY + HAVING

# Last Time...

- SELECT - FROM - WHERE

```
SELECT f.drinker,  
       f.coffeehouse,  
       ch.address  
  FROM Frequent f  
 JOIN Coffeehouses ch ON  
       f.coffeehouse = ch.name;
```

- JOIN

- ORDER BY, DISTINCT, Aggregations

- GROUP BY + HAVING

# Last Time...

- SELECT - FROM - WHERE

```
SELECT *
  FROM Sells
 ORDER BY price ASC;
```

- JOIN

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

- ORDER BY, DISTINCT, Aggregations

```
SELECT COUNT(DISTINCT price)
  FROM Sells;
```

- GROUP BY + HAVING

# Last Time...

- SELECT - FROM - WHERE

```
SELECT coffee, MAX(price)  
      FROM Sells  
    GROUP BY coffee;
```

- JOIN

```
SELECT coffee, AVG(price)  
      FROM Sells  
    GROUP BY coffee  
 HAVING COUNT(coffeehouse) > 1;
```

- ORDER BY, DISTINCT, Aggregations

- GROUP BY + HAVING

# Wake Up Task!

## Sports DB

People (ID, name, gender, height)

Gender (gender, description)

Sports (ID, name, record)

Competitions (ID, place, held)

Results (peopleID, competitionID, sportID, result)

How many sports have the word “Jump” in them?

How many people have participated in more than 25 competitions?

This Time...

# -- TODO

- Division
- JOINS & NULL
  - Natural Joins, Cross Joins, Self-Joins
  - NULL
  - Three-valued logic
  - Operators, Outer Joins
- Set Operations
  - UNION, INTERSECT, EXCEPT
- Subqueries (Nested Queries)
  - =, IN, EXISTS, ALL, ANY
- Views: Queries as subroutines

/

# What is Division?

- $R_1 / R_2 = \text{tuples of } R_1 \text{ associated with all tuples of } R_2$
- Find the students who have taken all courses in a program
- Find the airlines who land at all airports in a country/continent/the world
- Find the coffeehouses that sell all existing coffees

# Division with Counting

- We can write *division* using GROUP BY, HAVING and a COUNT subquery
- Find the coffeehouses that sell all existing coffees

Coffees (name, manufacturer)

Coffeehouses (name, address, license)

Drinkers (name, address, phone)

Likes (drinker, coffee)

Sells (coffeehouses, coffee, price)

Frequents (drinker, coffeehouse)

```
SELECT coffeehouse  
      FROM Sells  
      GROUP BY coffeehouse  
      HAVING COUNT(coffee) = (
```

2. For each coffeehouse,  
return it only if  
it sells that  
many

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

1. Count the number of coffees

# Practice

- Names of drinkers who frequent all coffeehouses
- 

```
SELECT drinker
      FROM Frequents
    GROUP BY drinker
HAVING COUNT(coffeehouse) = (
    SELECT COUNT(*)
      FROM Coffeehouses);
```

---

Coffees (name, manufacturer)

Coffeehouses (name, address, license)

Drinkers (name, address, phone)

Likes (drinker, coffee)

Sells (coffeehouses, coffee, price)

Frequents (drinker, coffeehouse)

# More Complete/Complex Example

- Exercise 2, #20

Show the ID, name, record and worst results of sports that have at least one result from every place where a competition has ever been held

```
SELECT s.id, s.name, s.record, MIN(r.result)
  FROM Sports s
  JOIN Results r ON s.id = r.sportId
  JOIN Competitions c ON
    r.competitionId = c.id
 GROUP BY s.id, s.name, s.record
 HAVING COUNT(DISTINCT c.place) = (
    SELECT COUNT(DISTINCT c.place)
      FROM Competitions c);
```

People (ID, name, gender, height)

Gender (gender, description)

Sports (ID, name, record)

Competitions (ID, place, held)

Results (peopleID, competitionID, sportID, result)

# -- TODO

## ✓ Division

## ● JOINS & NULL

- Natural Joins, Cross Joins, Self-Joins
- NULL
- Three-valued logic
- Operators, Outer Joins

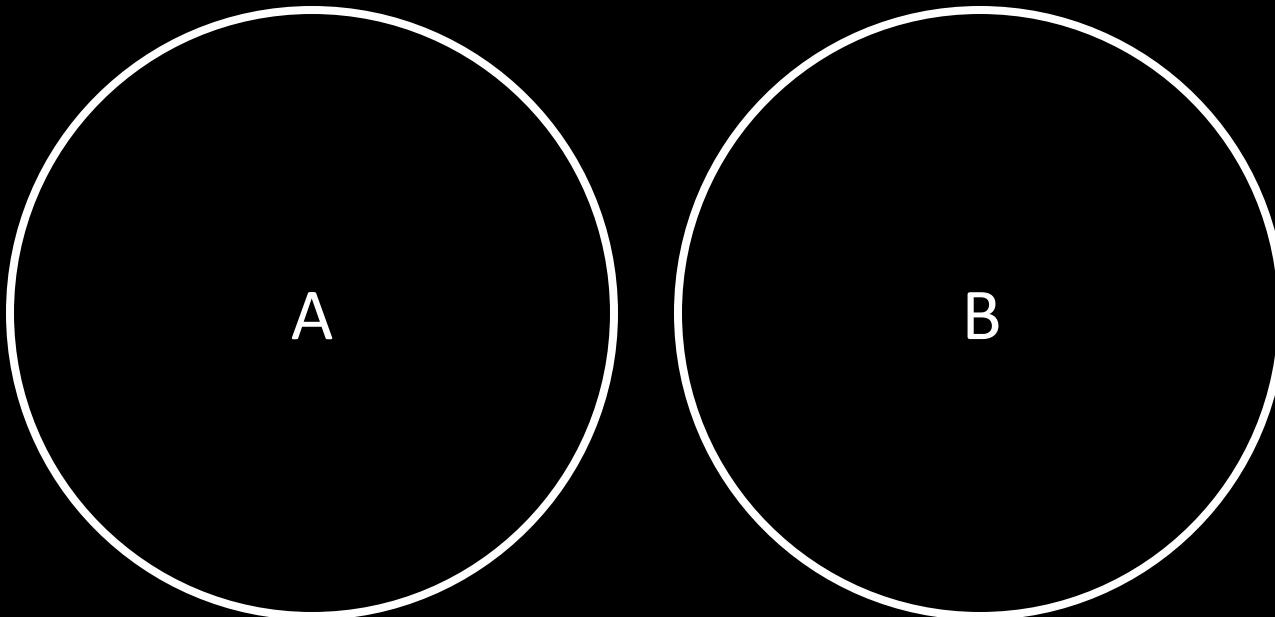
## ● Set Operations

- UNION, INTERSECT, EXCEPT

## ● Subqueries (Nested Queries)

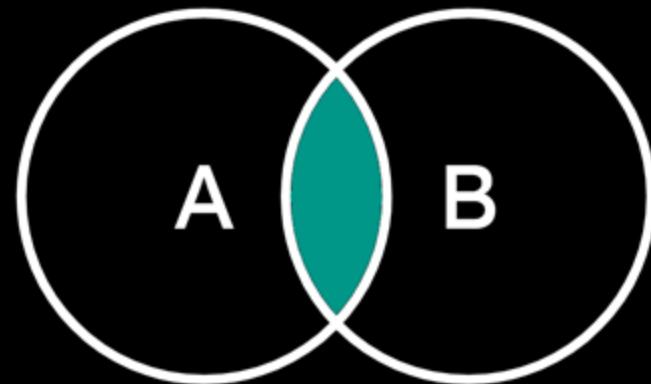
- =, IN, EXISTS, ALL, ANY

## ● Views: Queries as subroutines



# INNER JOIN

- Most joins are inner joins
- Combines the data of two relations based on a common column
- This is what we did last week
  - Following are some variants of INNER Joins



```
SELECT *
  FROM Drinkers d
  JOIN Frequents f ON
    d.name = f.drinker;
```

# Natural Joins and Cross Joins (Product)

## A NATURAL JOIN B:

- Assumes “=” on columns of same name
  - Removes duplicates
- Can be dangerous, not recommended!

```
SELECT *  
FROM Frequent  
NATURAL JOIN Likes;
```

## A CROSS JOIN B:

- Removes no columns, has no join condition
  - Can simulate with an always true condition (ON 1 = 1)
  - Rarely (but not never) the right thing to do!

```
SELECT *  
FROM Likes l  
CROSS JOIN Drinkers d;
```

# Self-Join

- From `Coffees(name, manufacturer)`, find all pairs of coffees by the same manufacturer
  - Need a double loop to compare two coffee records
  - WHERE clause ensures no pairs with the same coffees, and alphabetical order

```
SELECT c1.name, c2.name,  
       c1.manufacturer  
  FROM Coffees c1  
 JOIN Coffees c2 ON  
       c1.manufacturer =  
       c2.manufacturer  
 WHERE c1.name < c2.name;
```

# Self-Join: Example

- Show all coffees that are more expensive than some other coffee sold at the same coffeehouse

1. Get all pairs:

```
SELECT *  
FROM Sells s1  
JOIN Sells s2  
ON s1.coffeehouse =  
s2.coffeehouse;
```

2. Remove unwanted pairs:

```
SELECT *  
FROM Sells s1  
JOIN Sells s2  
ON s1.coffeehouse =  
s2.coffeehouse  
WHERE s1.price > s2.price;
```

Coffees (name, manufacturer)  
Coffeehouses (name, address, license)  
Drinkers (name, address, phone)  
Likes (drinker, coffee)  
Sells (coffeehouses, coffee, price)  
Frequents (drinker, coffeehouse)

3. Remove duplicates  
and show only coffees:

```
SELECT DISTINCT s1.coffee  
FROM Sells s1  
JOIN Sells s2  
ON s1.coffeehouse =  
s2.coffeehouse  
WHERE s1.price > s2.price;
```

# NULL Values

- Tuples in SQL relations can have NULL as a value for one or more components
- Meaning depends on context. Two Common cases:
  - **Missing Value** - e.g., we know Joe's Bar has some address, but we don't know what it is
    - "secret", "figure not available", "TBA", "impossible to calculate", "partly unknown", "uncertain", "pending"
  - **Inapplicable** - e.g., the value attribute spouse for an unmarried person
    - "undefined", "moot", "irrelevant", "none", "N/A"

# Comparing NULL Values

- The logic conditions in SQL is 3-valued
  - TRUE, FALSE, UNKNOWN
- Comparing any value (including NULL itself) with NULL yields UNKNOWN
  - $\text{NULL} \neq \text{NULL}$
  - Must use **IS NULL** or **IS NOT NULL**
- A tuple is in a query answer iff the **WHERE** clause is TRUE (not FALSE or UNKNOWN)

# Example

- Show the coffeehouses that sell a coffee at an unknown price
- Show the name of all coffees that are sold at a known price

Coffees (name, manufacturer)

Coffeehouses (name, address, license)

Drinkers (name, address, phone)

Likes (drinker, coffee)

Sells (coffeehouses, coffee, price)

Frequents (drinker, coffeehouse)

```
SELECT DISTINCT coffeehouse  
  FROM Sells  
 WHERE price IS NULL;
```

```
SELECT DISTINCT coffee  
  FROM Sells  
 WHERE price IS NOT NULL;
```

# Three-Valued Truth Tables

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

=	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	TRUE	NULL
NULL	NULL	NULL	NULL

# NULL vs Aggregation

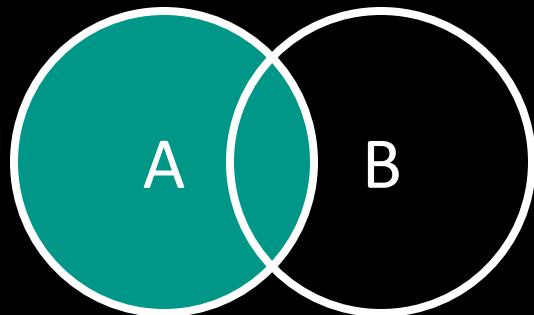
- NULL never contributes to a sum, average, or count, and can never be the minimum or maximum of a column
- But if there are no non-NULL values in a column then the result of the aggregation is NULL
  - Exception: COUNT of an empty set is 0

# Outer Joins

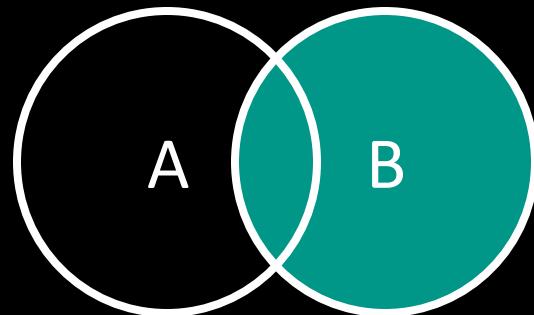
- Sometimes a coffee is sold nowhere, but we want it in our results
- $A \text{ OUTER JOIN } B$  is the core of an outer join expression
- It is modified by LEFT, RIGHT, or FULL before OUTER
  - LEFT = pad dangling tuples of A with NULL
  - RIGHT = pad dangling tuples of B with NULL
  - FULL = pad both;

# JOINS in a nutshell

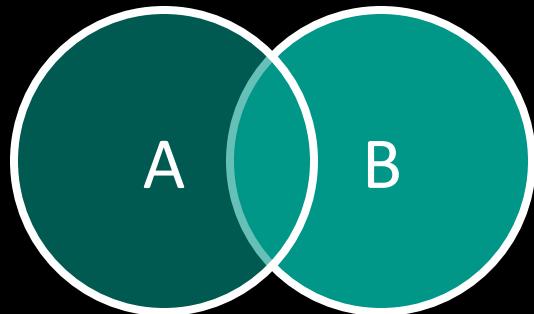
LEFT OUTER JOIN



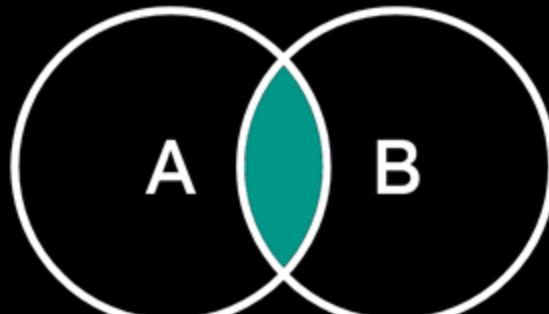
RIGHT OUTER JOIN



FULL OUTER JOIN



INNER JOIN



# Example of an Outer Join

Suppose we add a new coffee:

```
INSERT INTO Coffees VALUES  
( 'Bragakaffi' , 'Ó.J. & Kaaber' );
```

- Show coffeehouses along with the manufacturers of the coffees they are selling.  
Also include manufacturers that do not sell any coffees.

Coffees (name, manufacturer)  
Coffeehouses (name, address, license)  
Drinkers (name, address, phone)  
Likes (drinker, coffee)  
Sells (coffeehouses, coffee, price)  
Frequents (drinker, coffeehouse)

```
SELECT s.coffeehouse ,  
       c.manufacturer  
  FROM Sells s  
RIGHT OUTER JOIN Coffees c  
    ON s.coffee = c.name;
```

# Practice

- Show all drinkers and the coffees they like, but include drinkers that do not like any coffees
- Can you use OUTER JOIN to show ONLY drinkers who do not like any coffees?

Coffees (name, manufacturer)

Coffeehouses (name, address, license)

Drinkers (name, address, phone)

Likes (drinker, coffee)

Sells (coffeehouses, coffee, price)

Frequents (drinker, coffeehouse)

# Practice

- Show all drinkers and the coffees they like, but include drinkers that do not like any coffees
- Can you use OUTER JOIN to show ONLY drinkers who do not like any coffees?

Coffees (name, manufacturer)  
Coffeehouses (name, address, license)  
Drinkers (name, address, phone)  
Likes (drinker, coffee)  
Sells (coffeehouses, coffee, price)  
Frequents (drinker, coffeehouse)

```
SELECT *  
FROM Likes l  
JOIN Drinkers d ON d.name = l.drinker;
```

WRONG!

```
SELECT *  
FROM Likes l  
RIGHT OUTER JOIN Drinkers d  
ON d.name = l.drinker;
```

Correct!

```
SELECT *  
FROM Drinkers d  
LEFT OUTER JOIN Likes l  
ON l.drinker = d.name;
```

Correct!

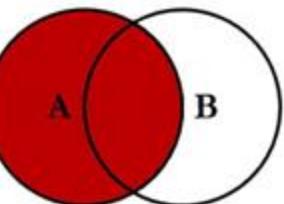
# Practice

- Show all drinkers and the coffees they like, but include drinkers that do not like any coffees
- Can you use OUTER JOIN to show ONLY drinkers who do not like any coffees?

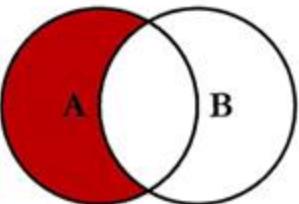
```
SELECT d.name  
FROM Drinkers d  
LEFT JOIN Likes l  
ON d.name = l.drinker  
WHERE l.drinker IS NULL;
```

Coffees (name, manufacturer)  
Coffeehouses (name, address, license)  
Drinkers (name, address, phone)  
Likes (drinker, coffee)  
Sells (coffeehouses, coffee, price)  
Frequents (drinker, coffeehouse)

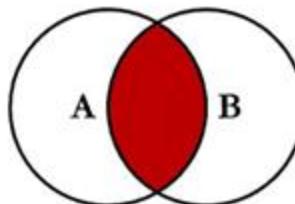
# SQL JOINS



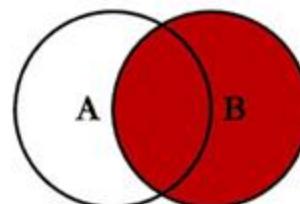
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



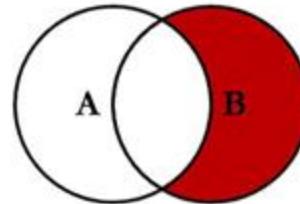
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



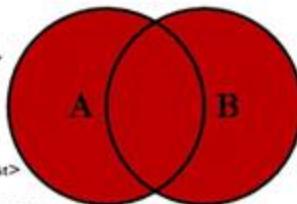
```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```

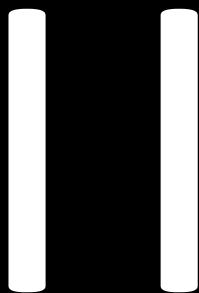


```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL,
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```

© C.L. Moffat, 2008



# -- TODO

✓ Division

✓ JOIN & NULL

✓ Natural Joins, Cross Joins, Self-Joins

✓ NULL

✓ Three-valued logic

✓ Operators, Outer Joins

- Set Operations
  - UNION, INTERSECT, EXCEPT
- Subqueries (Nested Queries)
  - =, IN, EXISTS, ALL, ANY
- Views: Queries as subroutines

# Set Queries

- Syntax:

<Query 1>

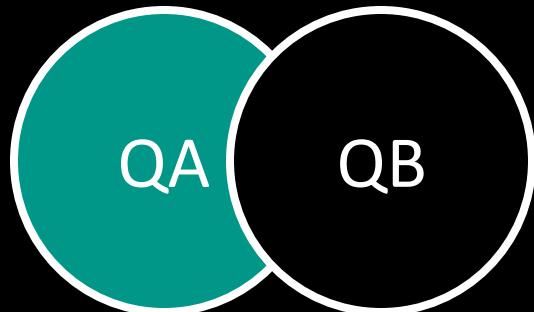
UNION/INTERSECT/EXCEPT

<Query 2>

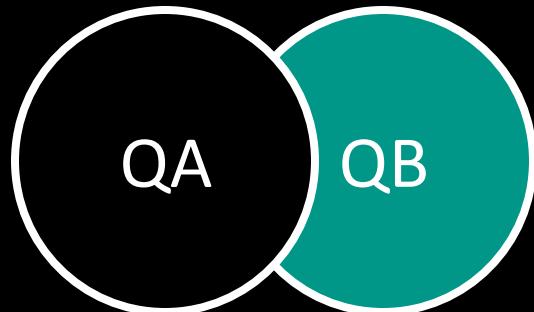
- Queries must be “union compatible” = results have matching schema:
  - Same number of attributes
  - Attributes  $i$  of both tables have same (matching type)

# Set Operators in a nutshell

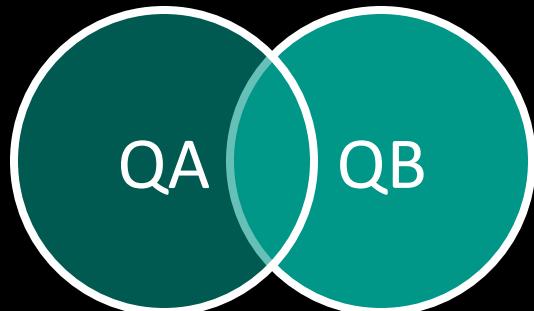
QA EXCEPT QB



QB EXCEPT QA



QA UNION QB  
QB UNION QA



QA INTERSECT QB  
QB INTERSECT QA



# Operators

- UNION

Show all drinkers that like “Kopi luwak” or live in “Amager”

- INTERSECT

Show all coffees that are manufactured by “Marley Coffee” and sold at an unknown price

- EXCEPT

Show all drinkers that frequents “Analog” but not those that like the coffee “Kopi luwak”.

# Operators

- UNION

Show all drinkers that like “Kopi luwak” or live in “Amager”

- INTERSECT

Show all coffees that are manufactured by “Marley Coffee” and sold at an unknown price

- EXCEPT

Show all drinkers that frequents “Analog” but not those that like the coffee “Kopi luwak”.

```
SELECT l.drinker
  FROM Likes l
 WHERE l.coffee = 'Kopi luwak'
UNION
SELECT d.name
  FROM Drinkers d
 WHERE d.address = 'Amager';
```

# Operators

- UNION

Show all drinkers that like “Kopi luwak” or live in “Amager”

- INTERSECT

Show all coffees that are manufactured by “Marley Coffee” and sold at an unknown price

- EXCEPT

Show all drinkers that frequents “Analog” but not those that like the coffee “Kopi luwak”.

```
SELECT c.name  
  FROM Coffees c  
 WHERE c.manufacturer = 'Marley Coffee'  
INTERSECT  
SELECT s.coffee  
  FROM Sells s  
 WHERE s.price IS NULL;
```

# Operators

- UNION

Show all drinkers that like “Kopi luwak” or live in “Amager”

- INTERSECT

Show all coffees that are manufactured by “Marley Coffee” and sold at an unknown price

- EXCEPT

Show all drinkers that frequents “Analog” but not those that like the coffee “Kopi luwak”.

```
SELECT f.drinker  
FROM Frequent f  
WHERE f.coffeehouse = 'Analog'  
EXCEPT  
SELECT l.drinker  
FROM Likes l  
WHERE l.coffee = 'Kopi luwak';
```

# Duplicates

- Set operators remove duplicates
- To keep duplicates use:
  - $R \text{ UNION ALL } S$  - (0, 1, 2x5, 3x3, 4, 5)
  - $R \text{ INTERSECT ALL } S$  - (2x2, 1x3)
  - $R \text{ EXCEPT ALL } S$  - (1, 0x2, 1x3, 4)
- Most practical use:  
 $\text{UNION ALL}$  when  $R$  and  $S$  are known to be disjoint!

<u>R</u>	<u>S</u>
1	0
2	2
2	2
3	2
3	3
4	5

# -- TODO

✓ Division

✓ JOIN & NULL

- ✓ Natural Joins, Cross Joins, Self-Joins

- ✓ NULL

- ✓ Three-valued logic

- ✓ Operators, Outer Joins

✓ Set Operations

- ✓ UNION, INTERSECT, EXCEPT

- Subqueries (Nested Queries)

- =, IN, EXISTS, ALL, ANY

- Views: Queries as subroutines

QUERY  
QUERY  
QUERY  
QUERY  
QUERY  
QUERY  
QUERY  
QUERY

# Subqueries

In SQL, a parenthesized SELECT-FROM-WHERE statement can be used as a value in many places, including WHERE and FROM clauses

```
SELECT *  
      FROM R  
     WHERE R.a <op>  
           (SELECT S.b  
                 FROM S);
```

```
SELECT T.a  
      FROM (SELECT R.a, S.c  
            FROM R  
           JOIN S  
          ON R.a = S.b)  
      AS T;
```

```
SELECT R.a  
      FROM R  
     JOIN (SELECT S.b, S.c  
            FROM S) AS T  
       ON R.a = T.b;
```

# Single-Tuple (=)

- Subqueries return relations, but...  
If a subquery is guaranteed to produce one tuple, then it can be used as a value
  - Usually, the tuple has only one attribute
  - Implicit type cast
  - Run-time Error when >1 tuples or no tuples are returned
- Practical for:
  - Finding records with highest or lowest values
  - Division queries

```
SELECT coffee
      FROM Sells
     WHERE price =
           (SELECT MIN(price)
              FROM Sells);
```

```
SELECT s.coffeehouse
      FROM Sells s
 GROUP BY s.coffeehouse
 HAVING COUNT(s.coffee) =
           (SELECT COUNT(*)
              FROM Coffees);
```

# Practice

- Find the coffeehouses that serve some coffee for the same price Mocha charges for Blue Mountain

Coffees (name, manufacturer)

Coffeehouses (name, address, license)

Drinkers (name, address, phone)

Likes (drinker, coffee)

Sells (coffeehouses, coffee, price)

Frequents (drinker, coffeehouse)

# Practice

- Find the coffeehouses that serve some coffee for the same price Mocha charges for Blue Mountain
- Using the `Sells(coffeehouse, coffee, price)` relation, this can be done in two queries:
  - Find the price Mocha charges for Blue Mountain
  - Find the coffeehouses that serve a coffee at that price
- Can it be done in a single query?

```
SELECT price  
FROM Sells  
WHERE coffeehouse = 'Mocha'  
AND coffee = 'Blue Mountain';
```

```
SELECT coffeehouse  
FROM Sells  
WHERE price = 300;
```

# Practice

- Find the coffeehouses that serve some coffee for the same price Mocha charges for Blue Mountain
- Using the `Sells(coffeehouse, coffee, price)` relation, this can be done in two queries:
  - Find the price Mocha charges for Blue Mountain
  - Find the coffeehouses that serve a coffee at that price
- Can it be done in a single query?

```
SELECT coffeehouse
  FROM Sells
 WHERE price =
    (SELECT price
      FROM Sells
     WHERE coffeehouse = 'Mocha'
       AND coffee = 'Blue Mountain');
```

---

# IN Operator

- Used to check if a tuple is IN a subquery's produced relation
  - The opposite can be achieved using NOT IN
- IN expressions can appear in WHERE clauses
- Example:  
Using **Coffees(name, manufacturer)** and **Likes(drinker, coffee)**, find the name and manufacturer of each coffee that Johan likes

```
SELECT *  
FROM Coffees  
WHERE name IN (  
    SELECT coffee  
    FROM Likes  
    WHERE drinker LIKE 'Johan%');
```

## Example: NOT IN

- Show the name of all drinkers that does not frequent any coffeehouses

```
SELECT d.name  
  FROM Drinkers d  
 WHERE d.name NOT IN (  
   SELECT f.drinker  
     FROM Frequents f);
```

# Set Operators: UNION

- Show all drinkers that like “Kopi luwak” or live in “Amager”

```
SELECT l.drinker  
      FROM Likes l  
     WHERE l.coffee = 'Kopi Luwak'  
UNION  
SELECT d.name  
      FROM Drinkers d  
     WHERE d.address = 'Amager';
```

```
SELECT d.name  
      FROM Drinkers d  
     WHERE d.address = 'Amager'  
          OR d.name IN (  
                  SELECT l.drinker  
                      FROM Likes l  
                     WHERE l.coffee = 'Kopi Luwak');
```

# Set Operators: INTERSECT

- Show all coffees that are manufactured by “Marley Coffee” and sold at an unknown price

```
SELECT c.name  
  FROM Coffees c  
 WHERE c.manufacturer = 'Marley Coffee'  
INTERSECT  
SELECT s.coffee  
  FROM Sells s  
 WHERE s.price IS NULL;
```

```
SELECT c.name  
  FROM Coffees c  
 WHERE c.manufacturer = 'Marley Coffee'  
      AND c.name IN (  
          SELECT s.coffee  
            FROM Sells s  
           WHERE s.price IS NULL);
```

# Set Operators: EXCEPT

- Show all drinkers that frequents “Analog” but not those that like the coffee “Kopi luwak”

```
SELECT f.drinker  
      FROM Frequents f  
     WHERE f.coffeehouse = 'Analog'  
EXCEPT  
SELECT l.drinker  
      FROM Likes l  
     WHERE l.coffee = 'Kopi luwak';
```

```
SELECT f.drinker  
      FROM Frequents f  
     WHERE f.coffeehouse = 'Analog'  
          AND f.drinker NOT IN (  
SELECT l.drinker  
      FROM Likes l  
     WHERE l.coffee = 'Kopi luwak');
```

# EXISTS Operator

- **EXISTS (<subquery>)**
  - Returns true if and only if a query result is not empty

- Example:

Show the names of coffees that are liked by someone

- Then show the opposite

```
SELECT c.name
  FROM Coffees c
 WHERE EXISTS (
    SELECT *
      FROM Likes l
     WHERE l.coffee = c.name);
```

```
SELECT c.name
  FROM Coffees c
 WHERE NOT EXISTS (
    SELECT *
      FROM Likes l
     WHERE l.coffee = c.name);
```

# EXISTS Operator

- Example:

From **Coffees(name, manufacturer)**,  
find coffees of manufacturers that only sell  
a single coffee

```
SELECT c1.name
      FROM Coffees c1
 WHERE NOT EXISTS (
   SELECT *
      FROM Coffees c2
     WHERE c2.manufacturer =
           c1.manufacturer
    AND c2.name <> c1.name);
```

---

Set of coffees with the same  
manufacturer as c1, but not the  
same name

# Practice

Insert the following:

```
INSERT INTO Coffeehouses VALUES  
('Kaffitar', 'Stroget', 'No question');
```

- Using NOT EXISTS, show the name of all coffeehouses that no one frequents

# Practice

Insert the following:

```
INSERT INTO Coffeehouses VALUES  
('Kaffitar', 'Stroget', 'No question');
```

- Using NOT EXISTS, show the name of all coffeehouses that no one frequents

```
SELECT ch.name  
FROM Coffeehouses ch  
WHERE NOT EXISTS (  
    SELECT *  
    FROM Frequent f  
    WHERE f.coffeehouse = ch.name);
```

# ANY Operator

- $x <\text{op}> \text{ANY} (<\text{subquery}>)$ 
  - Returns true iff at least one tuple in the subquery satisfies the comparison operator  $=, >, <, \dots$
- Example:
  - For each coffeehouse, show all coffees that are more expensive than some other coffee sold at that coffeehouse

```
SELECT coffee
      FROM Sells
 WHERE price > ANY (
           SELECT price
             FROM Sells);
```

# ALL Operator

- $x \text{ <op> ALL } (\text{<subquery>})$ 
  - Returns true iff for every tuple in the relation satisfies the comparison operator  $=, <, >, \dots$
- Example:
  - From  $\text{Sells}(\text{coffeehouse}, \text{coffee}, \text{price})$ , find the coffee(s) sold for the highest price

```
SELECT coffee
      FROM Sells
 WHERE price >= ALL (
          SELECT price
              FROM Sells
 WHERE price IS NOT NULL);
```

# FROM

- Subqueries can also be used in FROM clauses
- Must use a tuple-variable to name tuples of the results
- Example:
  - Find the coffees liked by at least one person who frequents Mocha

```
SELECT DISTINCT l.coffee
  FROM Likes l
  JOIN (SELECT drinker
        FROM Frequents
       WHERE coffeehouse =
             'Mocha') md
    ON l.drinker = md.drinker;
```

# -- TODO

✓ Division

✓ JOIN & NULL

- ✓ Natural Joins, Cross Joins, Self-Joins

- ✓ NULL

- ✓ Three-valued logic

- ✓ Operators, Outer Joins

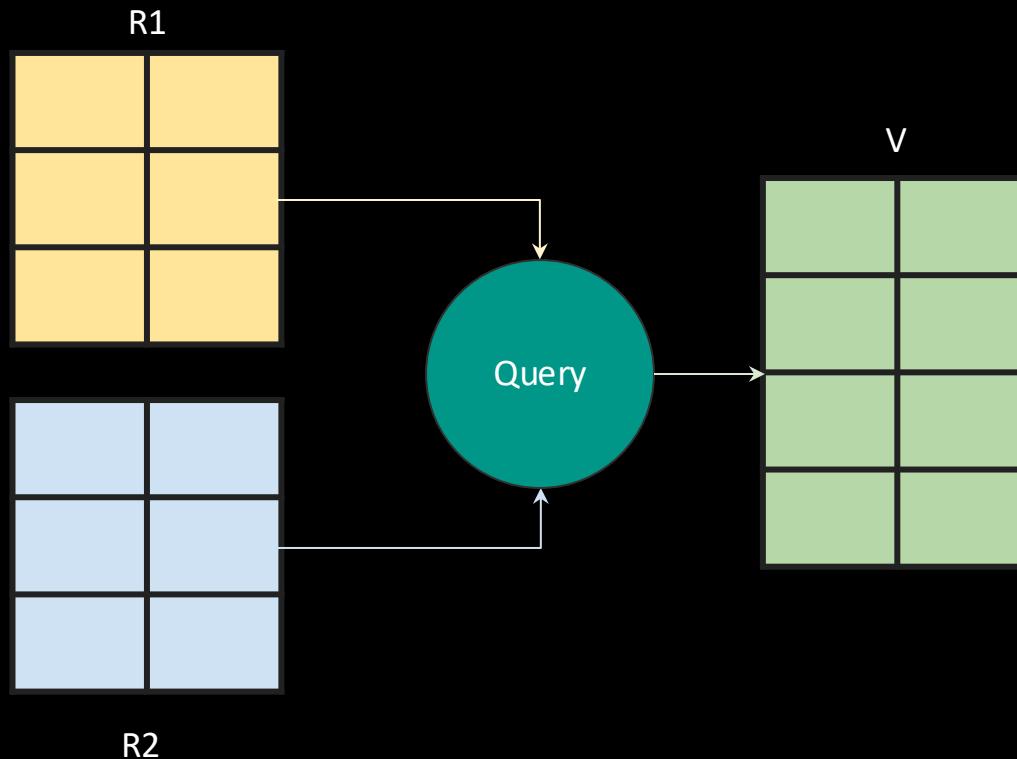
✓ Set Operations

- ✓ UNION, INTERSECT, EXCEPT

✓ Subqueries (Nested Queries)

- ✓ =, IN, EXISTS, ALL, ANY

- Views: Queries as subroutines



# What is a View?

- A virtual table constructed from actual tables on the fly
  - Accessed in queries like any other table
  - Not materialized, constructed when accessed
  - Similar to a subroutine in ordinary programming
  - Is a schema element

```
CREATE VIEW Name (Columns,...)  
AS  
<SQL QUERY>
```

# Example of View (Sports DB)

- For each result, we want:
  - the name of the athlete
  - the name of the sport
  - the percentage of the record achieved by the result  
(a result that is a record should therefore appear as 100; this column should be named “percentage”).

```
SELECT p.name, s.name,  
ROUND(100*CAST((r.result/s.record) AS  
numeric),0) AS percentage  
FROM People p  
JOIN Results r ON p.id = r.peopleid  
JOIN Sports s ON s.id = r.sportsid;
```

People (ID, name, gender, height)

Gender (gender, description)

Sports (ID, name, record)

Competitions (ID, place, held)

Results (peopleID, competitionID, sportID, result)

# Example of View (Sports DB)

- For each result, we want:
  - the name of the athlete
  - the name of the sport
  - the percentage of the record achieved by the result  
(a result that is a record should therefore appear as 100; this column should be named “percentage”).

People (ID, name, gender, height)

Gender (gender, description)

Sports (ID, name, record)

Competitions (ID, place, held)

Results (peopleID, competitionID, sportID, result)

CREATE VIEW

E2Q10 (pid, pname, sid, sname, percentage)

AS

```
SELECT p.id, p.name, s.id, s.name,  
ROUND(100*CAST((r.result/s.record) AS  
numeric),0) AS percentage
```

FROM People p

JOIN Results r ON p.id = r.peopleid

JOIN Sports s ON s.id = r.sportid;

SELECT pname, sname, percentage

FROM E2Q10;

# Practice (Sports DB)

Create views for these queries:

- The ID, name and gender of all athletes who participated in the competition held in Hvide Sande in 2009.
- The name and gender of all people with last names that starts with a “J” and ends with a “sen”

People (ID, name, gender, height)

Gender (gender, description)

Sports (ID, name, record)

Competitions (ID, place, held)

Results (peopleID, competitionID, sportID, result)

```
SELECT DISTINCT p.id, p.name, g.description
  FROM People p
  JOIN Gender g ON p.gender = g.gender
  JOIN Results r ON p.id = r.peopleid
  JOIN Competitions c
    ON c.id = r.competitionid
 WHERE c.place = 'Hvide Sande'
   AND EXTRACT(YEAR FROM c.held) = 2009;
```

```
SELECT p.name, g.description
  FROM People p
  JOIN Gender g ON p.gender = g.gender
 WHERE p.name LIKE '% J%sen';
```

# -- TODO -> DONE

✓ Division

✓ JOIN & NULL

✓ Natural Joins, Cross Joins, Self-Joins

✓ NULL

✓ Three-valued logic

✓ Operators, Outer Joins

✓ Set Operations

✓ UNION, INTERSECT, EXCEPT

✓ Subqueries (Nested Queries)

✓ =, IN, EXISTS, ALL, ANY

✓ Views: Queries as subroutines

# Takeaways

SQL

SEQUEL

SQUEAL

SQUIRREL

## One Block

- SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY

## Many Blocks

- Multi Query
  - Connected by: UNION, INTERSECT, EXCEPT
- Subqueries in FROM and WHERE
  - Blocks connected by: =, IN, EXISTS, ANY, ALL
  - Opposites (usually) with NOT
  - BUT: Prefer to use Joins when possible
- Division is one important type of queries with >1 query block!

Next Time in IDBS...

# Introduction to Database Systems

## IDBS - Fall 2024

### Week 4 - SQL and Python

---

Functions

Triggers

SQL and DBMS in Python

Transactions

---

Readings: DDBM 9.2, 14.1, 14.2.1 and 14.5

Omar Shahbaz Khan

# Introduction to Database Systems

## IDBS - Fall 2024

### Lecture 4 - SQL Programming and Python

Functions

Triggers & Constraints

SQL and DBMS in Python

Transactions

---

Readings: PDBM 9.2, 14.1, 14.2.1 and 14.5

Omar Shahbaz Khan

# Last Time in IDBS...

# -- TODO -> DONE

- ✓ Division
- ✓ JOIN & NULL
  - ✓ Natural Joins, Cross Joins,  
Self-Joins
  - ✓ NULL
  - ✓ Outer Joins (LEFT, RIGHT, FULL)
- ✓ Set Operations
  - ✓ UNION, INTERSECT, EXCEPT
- ✓ Subqueries (Nested Queries)
  - ✓ =, IN, EXISTS, ALL, ANY
- ✓ Views: Queries as subroutines

# Wake Up Task!

## Bills DB

People (PID, pName, pGender, pHeight)

Accounts (AID, PID, aDate, aBalance, aOver)

AccountRecords (RID, AID, rDate, rType, rAmount, rBalance)

Bills (BID, PID, bDueDate, bAmount, bIsPaid)

- How many accounts have never been used before?
- How many accounts have a negative account balance and also have bills due?

# This Time...

# -- TODO

- Database Functions
- Database Triggers & Constraints
- Transactions
- SQL in Python

$$f(x)$$

# Functions in SQL

- What is a function?
- Similar to other programming languages methods/functions
- How is it useful in SQL?

# Create a new Person

- In the Sports database create a new person
  - Insert a row
- Can be cumbersome to write `INSERT INTO ... VALUES ...` if we want to insert many new rows
- How about a function?

```
INSERT INTO  
People (name, gender, height)  
VALUES ('Terry', 'M', 1.77);
```

# Create a new Person

- In the Sports database create a new person
  - Insert a row
- Can be cumbersome to write `INSERT INTO ... VALUES ...` if we want to insert many new rows
- How about a function?

```
DROP FUNCTION IF EXISTS NewPerson;  
  
CREATE FUNCTION NewPerson (  
    Function Parameters  
        IN pname VARCHAR(50),  
        IN pgender CHAR(1),  
        IN pheight FLOAT  
    )  
RETURNS INTEGER  
AS $$  
BEGIN  
    Insert Statement  
        INSERT INTO  
        Person (name, gender, height)  
        VALUES (pname, pgender, pheight);  
        RETURN lastval();  
    END;  
$$ LANGUAGE plpgsql;  
    Return Type  
    Return the last value of  
    the table's SEQUENCE
```

# Using Functions

- From an SQL Script
- Typically used from an ODBC  
(Java, Python, C#, ...)

```
SELECT NewPerson('Terry', 'M', 1.77);
```

```
SELECT * FROM NewPerson('Terry', 'M', 1.77);
```

```
DO $$  
BEGIN  
    PERFORM NewPerson('Terry', 'M', 1.77);  
END  
$$;
```

# Let's make a function (I)

- Assume RecordLog table:

```
RecordLog (
    peopleID INT,
    competitionID INT,
    sportID INT,
    oldrecord FLOAT,
    newrecord FLOAT,
    seton DATE
);
```

- Function: BiggestRecordJump
  - Input: ID of sport
  - Output: The largest increase of that sport

```
RecordLog (
    peopleID INT,
    competitionID INT,
    sportID INT,
    oldrecord FLOAT,
    newrecord FLOAT,
    seton DATE
);
```

# Let's make a function (II)

- Assume RecordLog table:

```
RecordLog (
    peopleID INT,
    competitionID INT,
    sportID INT,
    oldrecord FLOAT,
    newrecord FLOAT,
    seton DATE
);
```

- Function: BiggestRecordJump
  - Input: ID of sport
  - Output: The largest increase of that sport

```
DROP FUNCTION IF EXISTS BiggestRecordJump();

CREATE FUNCTION BiggestRecordJump(
    IN sid INT
)
RETURNS FLOAT
AS $$ 
DECLARE r FLOAT; Variable to store value from query
BEGIN
    SELECT MAX(newrecord - oldrecord) INTO r
        FROM RecordLog
        WHERE sportID = sid
    RETURN r;
END;
$$ LANGUAGE plpgsql;

SELECT BiggestRecordJump(1);
```

# Are Functions Faster?

- May be faster than executing from a client
  - Why?
- Code may be pre-compiled and optimized
  - Do not need to invoke optimizer again
  - May occur with well written queries using plan caching
- The code runs at the server
  - The server may be more efficient
  - No need to move data

# Pros & Cons

- Code shared across ALL applications
- May be used for access control
- May give performance benefits
- Very system-specific
- Code maintenance requires care
  - Versioning is difficult

# IDENTITY Columns

- Run 04-sports-schema.sql
- Try inserting a new person
- Due to the ID column not being an IDENTITY column it will fail inserting the row
- Quickfix: Add parameter pid in the NewPerson Function
- Best to avoid this by using `GENERATED ALWAYS AS IDENTITY` when creating the table
- Fixed tables with ID columns correctly set to `GENERATED ALWAYS AS IDENTITY` (run 04-sports-schema-fixed.sql)
- In case you want to insert a row with a specific ID, you can do so by `OVERRIDING SYSTEM VALUE`. See example in the file.

# -- TODO

## ✓ Database Functions

- Database Triggers & Constraints
- Transactions
- SQL in Python



# Triggers in SQL

- Automatically executed function in response to certain events on a table (or view) in a database
  - Events are typically INSERT, UPDATE, DELETE
  - Triggers can execute BEFORE or AFTER the event
- Useful for maintaining the integrity of the information on the Database.
  - For example, in a company database when a new record (representing a new worker) is added to the employees table, new records could also be created in the tables of taxes, vacations and salaries.
- Logging historical data, for example to keep track of employees' previous salaries.

# Triggers in PostgreSQL

- Multiple triggers per table per event
  - INSERT / UPDATE / DELETE
  - Run in alphabetical order
  - Per row OR Per statement – focus here on per row
- New data is in the **NEW** record
  - For INSERT / UPDATE
  - Same schema as modified relation
  - Can refer to **NEW.ID**, **NEW.name**, ...
- Old data is in the **OLD** record
  - For UPDATE / DELETE
  - Same schema, refer to **OLD.ID**, **OLD.name**, ...
- The variable **TG\_OP** says which operation it is

# Trigger: Checking Values

- Use triggers to check value semantics
- A result in the Sports DB must not be negative

```
CREATE FUNCTION CheckResult()
RETURNS TRIGGER
AS $$ BEGIN
    IF (NEW.result < 0.0) THEN
        RAISE EXCEPTION
        'CheckResult: Result must be a
        positive'
        USING ERRCODE = '45000';
    END IF;
    RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

Create  
Trigger based  
on function

```
CREATE TRIGGER CheckResult
BEFORE INSERT OR UPDATE
ON Results
FOR EACH ROW EXECUTE PROCEDURE CheckResult();
```

# Trigger: Ban Updates/Deletes

- Use triggers to not allow updating or deleting rows

```
CREATE FUNCTION BanChanges()
RETURNS TRIGGER
AS $$
BEGIN
    RAISE EXCEPTION
        'BanChanges: Cannot change
        results!'
    USING ERRCODE = '45000';
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER BanChanges
BEFORE UPDATE OR DELETE
ON Results
FOR EACH ROW EXECUTE PROCEDURE
BanChanges();
```

# Trigger: Update Records

- Use triggers to update the values after an event
- If a result in Sports DB is a new record, the row for that sport in the sports table need to be updated

```
CREATE FUNCTION UpdateRecord()
RETURNS TRIGGER
AS $$ BEGIN
    IF NEW.result > (
        SELECT s.record
        FROM Sports s
        WHERE s.id = NEW.sportID
    ) THEN
        UPDATE Sports
        SET record = NEW.result
        WHERE s.id = NEW.sportID;
    END IF;
    RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

Could also be `RETURN NULL` since result of AFTER trigger is ignored

```
CREATE TRIGGER UpdateRecord
AFTER INSERT OR UPDATE
ON Results
FOR EACH ROW EXECUTE PROCEDURE UpdateRecord();
```

# Trigger: Log Changes

```
CREATE TABLE RecordLog (
    peopleID INT,
    competitionID INT,
    sportID INT,
    oldrecord FLOAT,
    newrecord FLOAT,
    seton DATE
    PRIMARY KEY
    (peopleID, competitionID, sportID)
    FOREIGN KEY
    (peopleID, competitionID, sportID)
    REFERENCES Results
    (peopleID, competitionID, sportID)
);
```

```
CREATE FUNCTION LogRecord()
RETURNS TRIGGER
DECLARE oldRecord FLOAT;
AS $$ BEGIN
    IF NEW.result > (
        SELECT s.record
        FROM Sports s
        WHERE s.id = NEW.sportID
    ) THEN
        SELECT s.record INTO oldRecord
        FROM Sports s
        WHERE s.id = NEW.sportID;
        INSERT INTO RecordLog
        VALUES (NEW.peopleID, NEW.competitionID,
                NEW.sportID, oldRecord,
                NEW.result);
    END IF;
    RETURN NEW;
END; $$ LANGUAGE plpgsql;

CREATE TRIGGER LogRecord
AFTER INSERT ON Results
FOR EACH ROW EXECUTE PROCEDURE LogRecord();
```

# Questions

- Why not create the LogRecord on Sports?
- What about Trigger order?
  - LogRecord vs UpdateRecord
  - Order is alphabetical = good in this case
    - Try with different order
  - May be better to merge similar triggers
- What about BEFORE or AFTER?
  - LogRecord and UpdateRecord are AFTER
  - CheckResult and BanUpdates are BEFORE
  - BEFORE and AFTER what? When does each apply? Why?

# Merged Trigger (BanUpdates + Check Updates)

```
CREATE FUNCTION MergedTrigger()
RETURNS TRIGGER
AS $$ BEGIN
    IF (TG_OP = 'DELETE' OR TG_OP = 'UPDATE')
    THEN
        RAISE EXCEPTION
            'MergedTrigger: Cannot change results!'
        USING ERRCODE = '45000';
    END IF;
    IF (NEW.result < 0.0) THEN
        RAISE EXCEPTION
            'CheckResult: Result must be a positive!'
        USING ERRCODE = '45000';
    END IF;
END; $$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER MergedTrigger
BEFORE INSERT OR UPDATE OR DELETE
ON Results
FOR EACH ROW EXECUTE PROCEDURE
MergedTrigger();

INSERT INTO Results
VALUES (1,1,3,-1.0);
DELETE FROM Results WHERE sportID = 3;
```

# BEFORE vs AFTER

1. Are you only checking the newly inserted/updated entry? -> BEFORE (or AFTER)
  - o Checking happens earlier with BEFORE, so less work
  - o But if other triggers might modify the values, then prefer AFTER
2. Are you inserting a row to another table with a foreign key constraint to the NEW record? -> AFTER
  - o Otherwise, the NEW record is NOT in the database, so your insertion will fail!
3. Are you modifying the NEW record? -> BEFORE
  - o Otherwise, the record is already in the database and will not be changed
4. Are you doing both 2 and 3? -> BEFORE and AFTER
  - o Two different triggers!

# Are Triggers Faster?

- May be faster than executing the code from a client
  - Why?
- Code may be pre-compiled and optimized
  - Do not need to invoke optimizer again
  - But this may happen with well written queries using plan caching
- The code runs at the server
  - The server may be more efficient
  - No need to move data to client

# Pros & Cons

- Code often runs faster
    - No context switch
    - Compiled code
    - No data transfers
  - Useful for security
    - Wrap data and functionality
    - Same access from all clients
  - Same code for all applications
- 
- Code is “hidden”
    - Only visible via system tables
    - Easily forgotten
    - Versioning is hard!
    - Schema may be edited using a GUI = no-no!
  - Generally not portable

# Exercises on SQL Programming

- Exercise 4 are out: Views, functions, triggers
  - Use test script to verify your work
  - Can run parts in pgAdmin – best via command prompt
  - You can extend the script with your own tests!
  - Also has a very nice database cleaning script

```
# once
psql -q Bills < bills-schema.sql

#repeatedly
psql Bills < universal-cleanup.sql
psql Bills < your-solution.sql
psql Bills < test-script.sql > output.txt 2>&1
less output.txt (linux / mac)
cat output.txt (windows) or open in an editor
```

NOTE: Windows uses '-f' instead of '<' and requires the DB name at the end of the command

NOTE 2: You may also be required to add user, -U <username>

# Exercises on SQL Programming (Catalog)

- All information about the database is stored in tables!
  - This is the catalog!

<https://www.postgresql.org/docs/current/catalogs.html>

- See also: System information functions

<https://www.postgresql.org/docs/current/functions-info.html>

```
SELECT *  
FROM information_schema.tables  
WHERE table_schema = 'public';
```

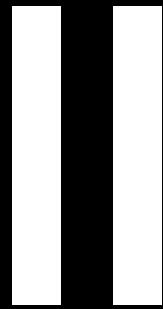
```
SELECT current_database();
```

```
SELECT current_user;
```

```
SELECT lastval();
```

# -- TODO

- ✓ Database Functions
- ✓ Database Triggers & Constraints
- Transactions
- SQL in Python





# Transfer 50 DKK from Bob to Alice

Step 1: Get balance of Bob

```
SELECT Balance INTO x  
FROM accounts  
WHERE AccountName = 'Bob'
```

Account Name	Balance
Bob	200
Alice	0

Account Name	Balance
Bob	150
Alice	0

Step 2: Reduce balance of Bob

```
UPDATE accounts  
SET Balance = x-50  
WHERE AccountName = 'Bob'
```

## WHAT IF SOMETHING CRASHES?

Step 3: Get balance of Alice

```
SELECT Balance INTO y  
FROM accounts  
WHERE AccountName = 'Alice'
```

Account Name	Balance
Bob	150
Alice	0

Account Name	Balance
Bob	150
Alice	50

Step 4: Increase balance of Alice

```
UPDATE accounts  
SET Balance = y+50  
WHERE AccountName = 'Alice'
```

# What is a Transaction?

- A group of related operations to the database
- Wish for “all or nothing” execution
- Wish for isolation from other transactions

## Basic SQL Syntax:

```
BEGIN;  
COMMIT;  
ROLLBACK;
```

## Savepoints:

```
SAVEPOINT <name>;  
ROLLBACK TO SAVEPOINT <name>;
```

# Transactions in PostgreSQL

- By default, every statement is a transaction
  - To override this behaviour:
  - `BEGIN; ... COMMIT; / ROLLBACK;`
  - Some (DDL) statements implicitly COMMIT transactions
- Calling a function starts a transaction
  - Can assume that the function has transactional properties!
  - Errors abort the transaction, erase all previous operations!
- Errors inside functions:
  - Cannot simply say: `ROLLBACK`
  - Need to raise and handle exceptions!
  - See example in exercise code

# Transactions and Testing

- Transactions are useful in test scripts
  - Multiple examples in Exercise 4
- Simple pattern to test changing the database
  - ... without actually changing the database

```
BEGIN  
  -- Make changes  
  -- Run test queries  
ROLLBACK
```

# -- TODO

- ✓ Database Functions
- ✓ Database Triggers & Constraints
- ✓ Transactions
- SQL in Python

# Problem: Vendor Lock-in

- They're all compatible with standard SQL...
- ...and mutually incompatible due to vendor-specific functionality!



# Open Database Connectivity (ODBC)

- In-between translation layer
- Vendors not so happy as lock-in is profitable!



# Python and PostgreSQL

- pyodbc is a package that implements ODBC and can connect to any databases
- The psycopg package is a database adapter specifically for PostgreSQL
  - Reliable, documented, references
  - To install: `python -m pip install psycopg` or `pip install psycopg`
    - import psycopg
- A simple database API - Similar to ODBC
  - `connect()`: Establish connection to specified database
  - `cursor()`: Object to manage context of an SQL operation
  - `execute()`: Execute SQL queries
  - `fetchone()`, `fetchmany()`, `fetchall()`: Get results from queries
  - Much more, but these are the most used functions

# Connecting to a Database

```
import psycopg as pg  
from psycopg.rows import dict_row ← Optional: Rows are returned as dict, better readability
```

```
conn_string = "host=localhost dbname=Sports user=postgres password=*****"
```

```
# Connection option one  
conn = pg.connect(conn_string, autocommit=True, row_factory=dict_row)  
# Execute Queries using Cursors  
conn.close()
```

```
# Connection option two  
with pg.connect(conn_string, autocommit=True, row_factory=dict_row) as conn:  
    # Execute Queries using Cursors  
# Closes connection
```

# Tips

```
try:  
    # Connection option one  
    conn = pg.connect(conn_string, autocommit=True, row_factory=dict_row)  
    # Execute Queries using Cursors  
except Exception as error:  
    print(error)  
finally:  
    conn.close()  
  
try:  
    # Connection option two  
    with pg.connect(conn_string, autocommit=True, row_factory=dict_row) as conn:  
        # Execute Queries using Cursors  
        # Closes connection  
    except Exception as error:  
        print(error)
```



This block always run

# Querying the DB

```
# Get all athletes  
athletes: list[dict] = conn.execute("SELECT * FROM Athletes").fetchall()
```

Creates a cursor object and  
executes the query

```
# Get five athletes  
athletes: list[dict] = conn.execute("SELECT * FROM Athletes").fetchmany(5)
```

What if I want more items later  
from the same query?

```
# Get all athletes  
athletes: list[dict] = conn.execute("SELECT * FROM Athletes").fetchone()
```

# Cursors

```
# Get query cursor
athletes_cursor = conn.execute("SELECT * FROM Athletes")

# Get five athletes
athletes: list[dict] = athletes_cursor.fetchmany(5)

# Get sixth athlete
athlete_six = athletes_cursor.fetchone()

# Get the next 10 athletes
athlete_next_ten = athletes_cursor.fetchmany(10)
```

# Cursors

```
# Get query cursor
athletes_cursor = conn.execute("SELECT * FROM Athletes")

# Get five athletes
athletes: list[dict] = athletes_cursor.fetchmany(5)

sports_cursor = conn.execute("SELECT * FROM Sports")

# Get sixth athlete
athlete_six = athletes_cursor.fetchone()

sports: list[dict] = sports_cursor.fetchall()

# Get the next 10 athletes
athlete_next_ten = athletes_cursor.fetchmany(10)

athletes_cursor.close()
sports_cursor.close()
```

Advantage: Multiple cursors active at the same time, which can fetch or execute queries

Remember to close the cursors (context block, finally block)

# Creating a Table

```
with pg.connect(conn_string, autocommit=True, row_factory=dict_row) as conn:  
    conn.execute("DROP TABLE IF EXISTS Quotes")  
    conn.execute(  
        """  
        CREATE TABLE Quotes (  
            x INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,  
            y VARCHAR(250) NOT NULL,  
        )  
        """  
    )
```

Better than SERIAL and follows  
SQL Standard.  
Still a sequence under the hood.  
[PostgreSQL Identity Columns](#)

# INSERT

```
conn = pg.connect(conn_string, autocommit=True, row_factory=dict_row)
with conn:
    conn.execute("INSERT INTO Quotes (y) VALUES ('I am Vengeance')")

nope = conn.execute("SELECT 42 as universe").fetchone()
```

Won't run since the connection is closed

# INSERT

```
conn = pg.connect(conn_string, autocommit=True, row_factory=dict_row)
with conn:
    conn.execute("INSERT INTO Quotes (y) VALUES ('I am Vengeance')")
```

```
nope = conn.execute("SELECT 42 as universe").fetchone()
```

Won't run since the connection is closed

```
with conn.cursor() as cur:
```

```
    cur.execute("INSERT INTO Quotes (y) VALUES ('I am the Night')")
```

```
yes = conn.execute("SELECT 42 as universe").fetchone()
```

Will run since only the cursor closed

```
conn.close()
```

# Querying with variables

```
in_text = 'I am Batman'  
sel_text = "'%Batman%'"
```

```
with pg.connect(conn_string, autocommit=True, row_factory=dict_row) as conn:  
    conn.execute("INSERT INTO Quotes (y) VALUES (%s)", [in_text])  
    conn.execute("SELECT * FROM Quotes WHERE y LIKE %s" % sel_text)
```

Question: Is this good or bad?

**SELECT Query is Prone to SQL Injections!**

# SQL Injection Example

```
in_text = 'I am Batman'  
sel_text = "'%Batman%'; DELETE FROM Quotes WHERE 1=1; INSERT INTO Quotes (y)  
VALUES 'Joker was here';--"  
  
with pg.connect(conn_string, autocommit=True, row_factory=dict_row) as conn:  
    conn.execute("INSERT INTO Quotes (y) VALUES (%s)", [in_text])  
    conn.execute("SELECT * FROM Quotes WHERE y LIKE %s" % sel_text)
```

Even relevant in the future:



Star Trek Discovery Season 2  
Episode 8

Year 2255+

# Prepared Statements

- Separate the query string and variables
- Prepared Statements
  - Pre-compiled SQL statement at DBMS
  - Client supplied data is treated as content of a parameter and not an SQL statement

```
PREPARE my_query AS  
SELECT *  
FROM Quotes  
WHERE y = "$1"
```

```
EXECUTE my_query ("Probe used SQL  
Injection... it's not very effective")
```

# Prepared Statements in Python

```
in_text = 'I am Batman'  
sel_text = "'%Batman%'; DELETE FROM Quotes WHERE 1=1; INSERT INTO Quotes (y)  
VALUES 'Joker was here';--"
```

```
with pg.connect(conn_string, autocommit=True, row_factory=dict_row) as conn:  
    conn.execute("INSERT INTO Quotes (y) VALUES (%s)", [text])  
    conn.execute("SELECT * FROM Quotes WHERE y LIKE %s", [sel_text])
```



# Not safe against all cases!

- Some elements can't be prepared
- Most prominent is the ORDER BY clause
- Check the input in python before using the variable

```
sort_key = "name"  
  
if sort_key in columns:  
  
    cur = conn.execute(  
        f"""  
        SELECT *  
        FROM Athletes  
        ORDER BY {sort_key}  
        """)  
  
    athletes = cur.fetchall()
```

# Autocommit

- If not specified in `connect()` autocommit is OFF by default
- Using `with` connection context commits at the end of the block

```
with pg.connect(conn_string, row_factory=dict_row) as conn:  
    conn.execute("INSERT INTO Quotes (y) VALUES ('I am the one who knocks')")  
    # BEGIN;  
    # INSERT...;  
    # COMMIT; (default) / ROLLBACK; (if error is raised)
```

- Good for a small number of queries
- But closes connection after, so this may not always be desired

# Autocommit (Cursors)

```
conn = pg.connect(conn_string, row_factory=dict_row)

with conn.cursor() as cur:

    tests = cur.execute("SELECT * FROM Quotes")

    # BEGIN;

    # SELECT...;

# Not committed, manually call conn.commit() inside or after the block
conn.commit()
```

- Even a `SELECT` statement begins a transaction
  - Easy to forget, making it more dangerous to not set autocommit on

# Transactions with Autocommit ON

```
conn = pg.connect(conn_string, autocommit=True, row_factory=dict_row)

conn.execute("INSERT INTO Quotes (y) VALUES ('I am inevitable')")

# BEGIN; SELECT...; COMMIT/ROLLBACK;

with conn.transaction():

    # BEGIN;

    with conn.cursor() as cur:

        cur.execute("INSERT INTO Quotes (y) VALUES ('I am Groot')")

        cur.execute("INSERT INTO Quotes (y) VALUES ('I am Iron Man!')")

# COMMIT; (default) / ROLLBACK; (if error is raised)
```

# -- TODO -> DONE

- ✓ Database Functions
- ✓ Database Triggers & Constraints
- ✓ Transactions
- ✓ SQL in Python

# Takeaways

## Functions

Set of database operations, performance benefits

## Triggers

Execute a function in response to a database event

**ODBC (Python psycopg, Java JDBC)**  
API for database operations

## Transactions

Atomicity, Consistency, Isolation, Durability



**Next Time in IDBS...**

# Introduction to Database Systems

## IDBS - Fall 2024

### Lecture 5 - Designing Databases

ER Diagrams

Translation to SQL DDL

---

Readings: PDBM 3.0-3.3, 6.3-6.4

Omar Shahbaz Khan

# Extra bits

# Roles in Databases

- Multiple users interacting with the database
  - Should everyone use the postgres role?
  - Roles are the same as Users
- Access Control
  - We can define roles with different privileges
  - Avoid accidents or fatal errors (database related and business related)

# Roles in SQL

- CREATE

```
CREATE ROLE <role> WITH LOGIN ENCRYPTED PASSWORD <password>;  
CREATE ROLE viewer WITH LOGIN ENCRYPTED PASSWORD '1337';
```

- GRANT

```
GRANT <privilege> ON <schema_elements> TO <role>;  
GRANT SELECT ON Sports, Results, Competitions, Athletes,  
Gender TO viewer;
```

- REVOKE

```
REVOKE SELECT ON Sports, Results, Competitions, Athletes,  
Gender FROM viewer;
```

- DROP OWNED BY

```
DROP OWNED BY viewer;
```

- DROP

```
DROP ROLE viewer;
```

# Using Roles

- To use roles in Python and in general we use the SQL statement SET ROLE
- SET ROLE <role>
- SET LOCAL ROLE <role>
- Best to execute in a transaction so we do not accidentally allow others to execute as the set role
- Check the add functions in the exercise

# Roles and Functions I

- Creating a function in PostgreSQL grants execute privileges on public
  - Many roles could have access to public!
  - Solution: Revoke access from public and then grant access to specific roles
  - **REVOKE ALL ON FUNCTION <function> FROM public;**
  - **GRANT EXECUTE ON FUNCTION <function> TO <role>;**

# Roles and Functions II

- By default a function is executed with the privileges of the calling role
  - A function accessing a schema element which the calling role does not have privileges to, an InsufficientPrivileges error is raised
- Resolution: **SECURITY DEFINER**
  - Add this after **LANGUAGE plpgsql**
  - The function is run with the privileges of the user that created it
  - Be careful!

A C I D

Transaction Properties



Recovery

Constraints, Triggers

Concurrency Control

Recovery

# A TOMICITY

A Transaction is  
“one operation”

# C ONSISTENCY

Each Transaction  
moves the DB from one  
consistent state to  
another

# I SOLATION

Each Transaction  
is alone in the  
world

# D URABILITY

Persistence of  
successful  
transactions even  
through system failure

# Transaction Implementation Methods (Lecture 10)

- Consistency
  - PRIMARY and FOREIGN KEY = limited DBMS support
- Isolation
  - Historically locking = (strict/rigorous) two-phase locking
  - Recently multi-version concurrency control
- Atomicity / Durability
  - Logging all changes to disk
  - Write Ahead Logging protocol (WAL)

# Introduction to Database Systems

## IDBS - Fall 2024

### Lecture 5 - Designing Databases

ER Diagrams

Translation to SQL DDL

---

Readings: PDBM 3.0-3.3, 6.3-6.4

Omar Shahbaz Khan

# General Info

## **EXERCISE 5: ER Design and Implementation**

- Large and comprehensive
- More than 2hrs, but use it to practice

## **HOMEWORK 2 - OUT NOW!**

- DEADLINE: 14. October 2024 23:59

# -- TODO

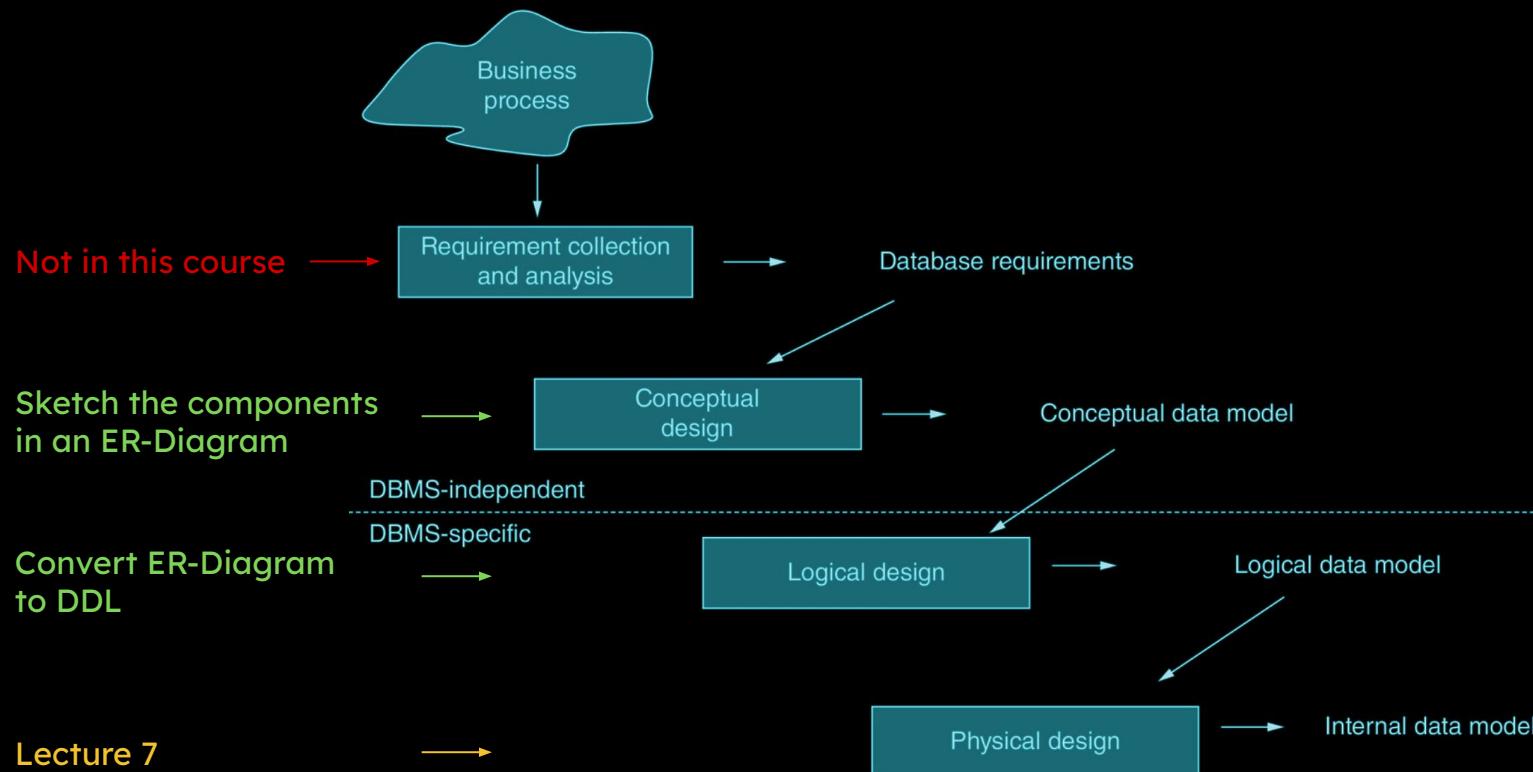
- Conceptual Data Modeling (ER Diagram)
  - Entities and Attributes
  - Relationships
    - Cardinalities
    - Partial Relationship Keys
  - Weak Entities
  - Aggregation
  - Generalization/Specialization
  - Categorization
- Translation to SQL DDL

# Conceptual Data Modeling

# Why do we need a Conceptual Model?

- The “higher-ups” know they want a database...  
... but not what should be in it!
- Need an effective method to develop the schema and document its structure

# Design Process



# ER: Entity-Relationships

- Conceptual Data Modeling technique was defined by Peter Chen (1976)
- ER = modeling concepts + visual representation
  - ER/EER notation is not standardized
  - Every textbook/company/tool has its own visual representation
  - Core concepts are universally accepted
  - EER vs ER rarely distinguished -> we just call it ER!
- UML can be used as design notation
  - Slight differences - NO UML IN THIS COURSE
- Focus on ER notation from the book
  - ... plus, some minor extensions - made clear in the lectures
  - In exercises, project, exam: Use the notation in book, lectures

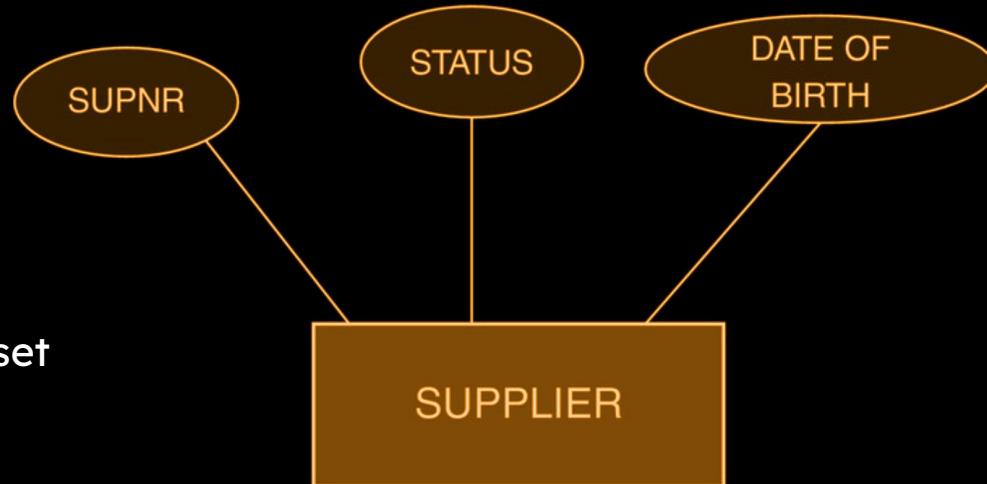
# Entity and Attribute Types

## Entity Type:

- Set of similar “things”
  - Ex: Movie, Supplier
- Entity = Instance
  - Ex: Interstellar, 2014

## Attribute Type:

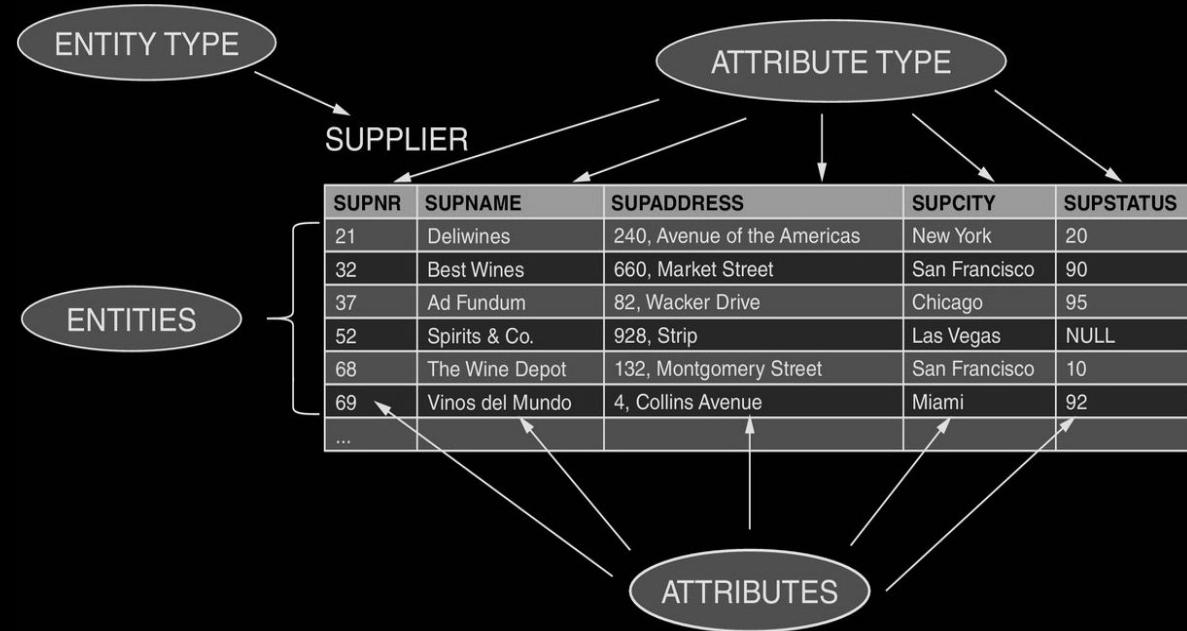
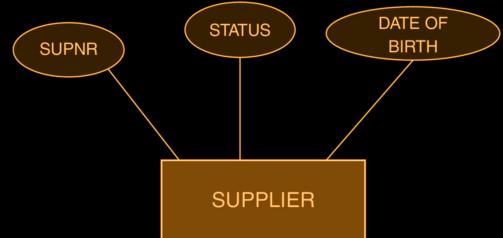
- Describes one aspect of an entity set
  - Ex: name, year, address



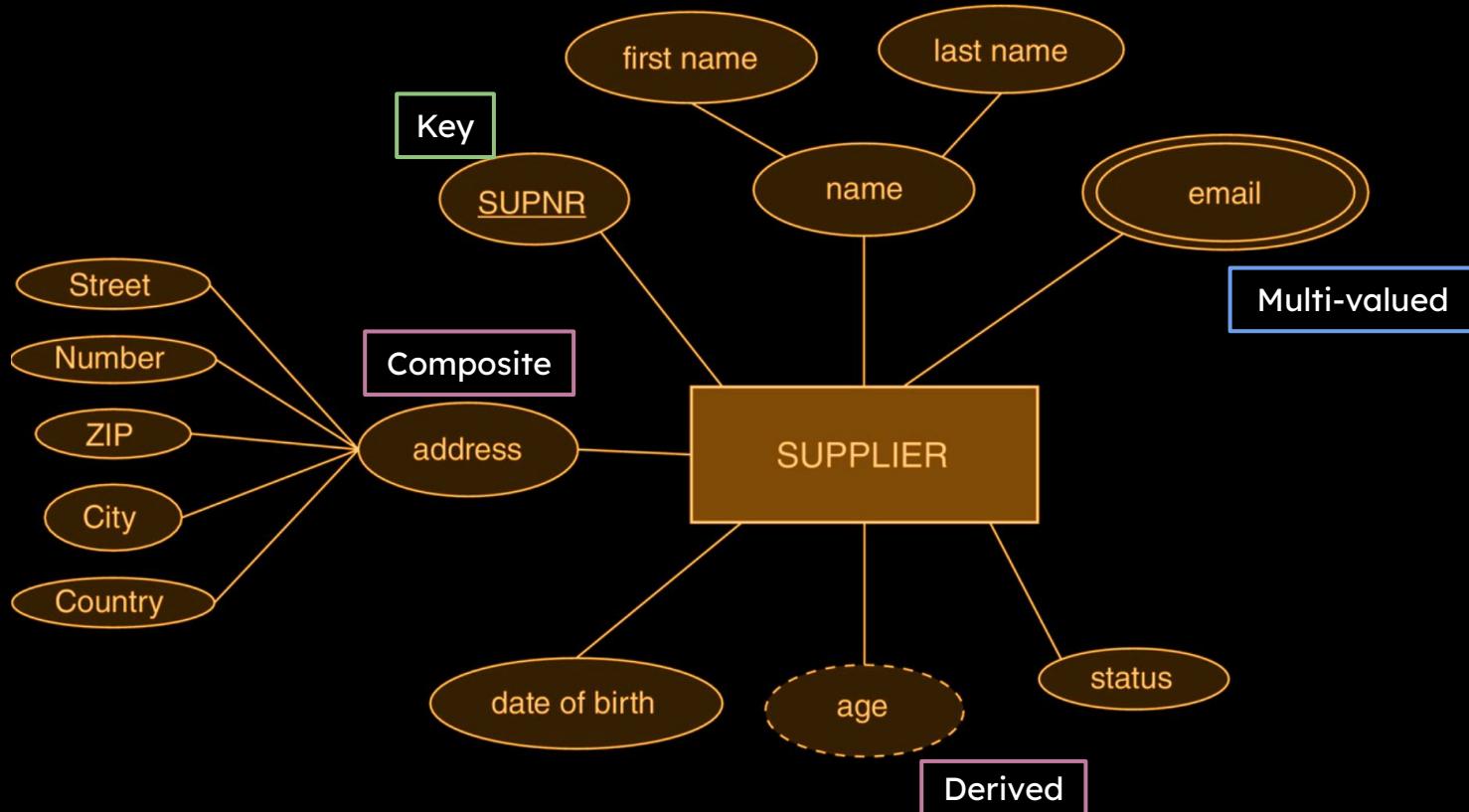
Is there something suspicious about this entity?

# Entity and Attribute Types

```
CREATE TABLE Supplier (
    SUPNR INT PRIMARY KEY,
    SUPNAME VARCHAR NOT NULL,
    ...
);
```



# More Attribute Types



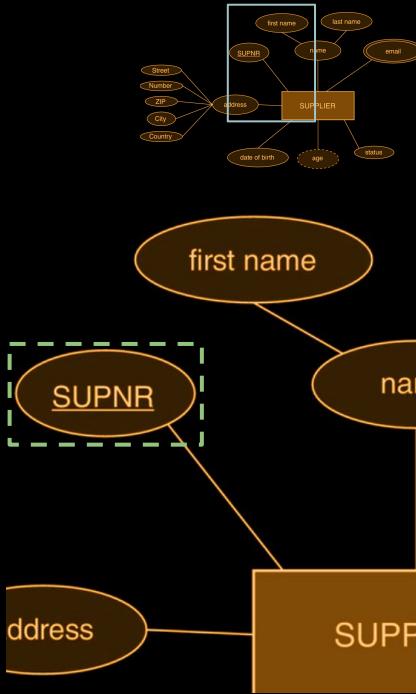
# Keys in ER-Diagrams and SQL DDL

- Underlined key = PRIMARY KEY
- ER diagrams cannot show secondary keys
  - They must be noted somewhere else!
  - They must still be UNIQUE in the SQL table!

```
CREATE TABLE Supplier (
    SUPNR INT PRIMARY KEY,
    ...
);
```

```
CREATE TABLE Supplier (
    SUPNR INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    ...
);
```

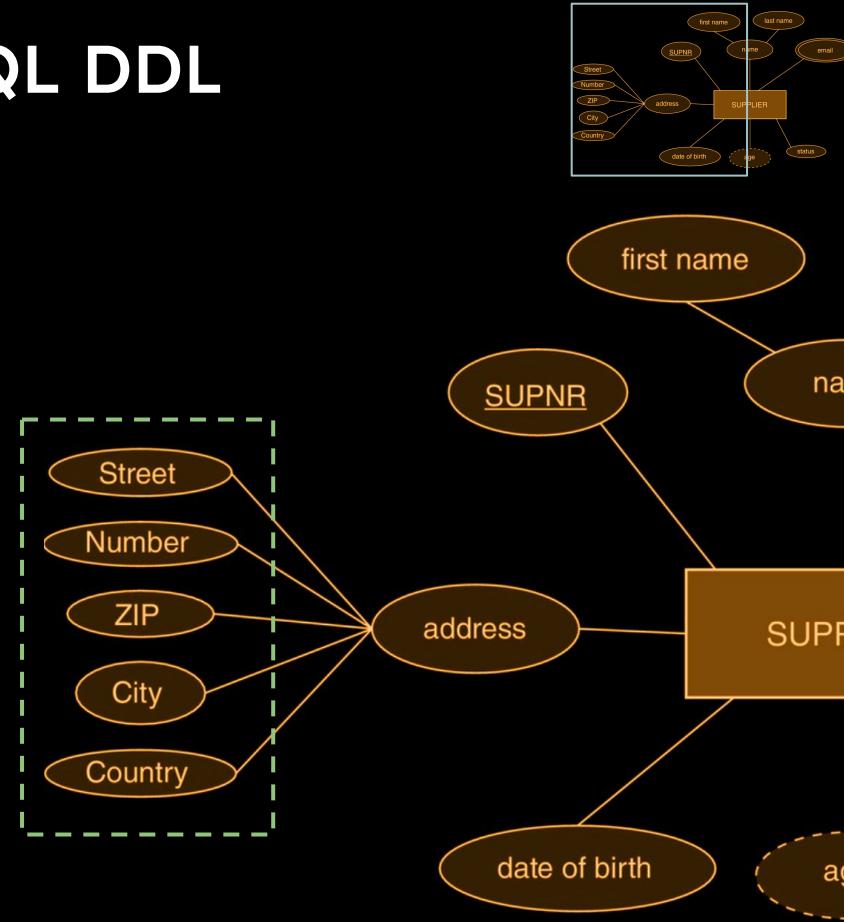
In case you need an  
incremental sequence



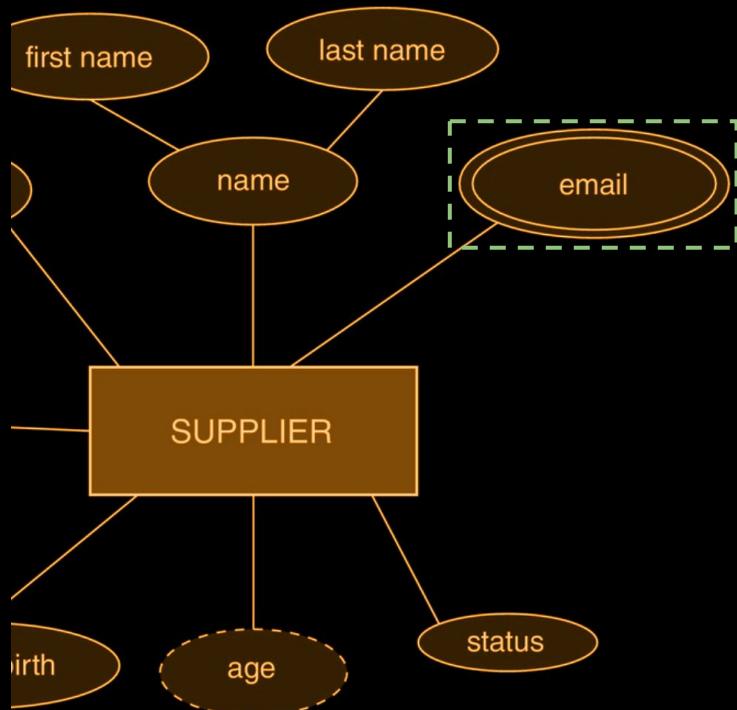
# Composite Attributes in SQL DDL

- Simply add the detailed attributes:

```
CREATE TABLE Supplier (
    SUPNR INT PRIMARY KEY,
    ...
    Street VARCHAR NOT NULL,
    Number INTEGER NOT NULL,
    ZIP INTEGER NOT NULL,
    City VARCHAR NOT NULL,
    Country VARCHAR NOT NULL,
    ...
);
```

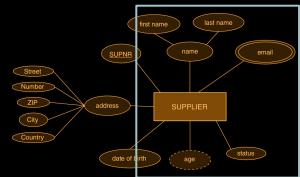


# Multi-Valued Attributes in SQL DDL



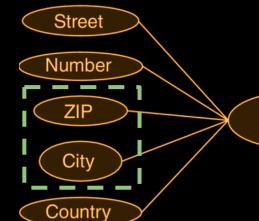
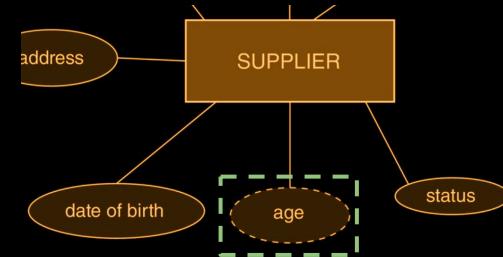
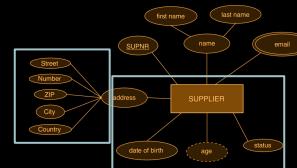
```
CREATE TABLE Emails (
    Email VARCHAR,
    SupNR INTEGER REFERENCES Supplier(SupNR),
    PRIMARY KEY (Email, SupNR)
);
```

- Could add **ON DELETE CASCADE** to **SupNR**
- Generally a good idea to add to foreign keys, if the related rows should not exist independent of that key



# Derived Attributes in SQL DDL

- Not discussed in the PDBM book!
- Option 1: Create an attribute and maintain it
  - E.g. with a trigger, or regular update processes
- Option 2: Create a view that computes it
- Neither is very good!
- Sometimes there is a second kind of inter-attribute relationship
  - Here: ZIP → City
  - Called functional dependencies (FDs)
  - ER diagrams may miss such relationships!
  - We fix this with normalization (Lecture 6)



# The CHECK Statement

- We can add constraints that check the value of a field directly in the DDL
- These can be on the column or on the table
  - Postgresql does not care but other database system may

```
CREATE TABLE Supplier (
    Id INTEGER PRIMARY KEY,
    ...
    status CHAR(10) NOT NULL
    CHECK (status IN ('Active', 'Inactive')),
    ...
);

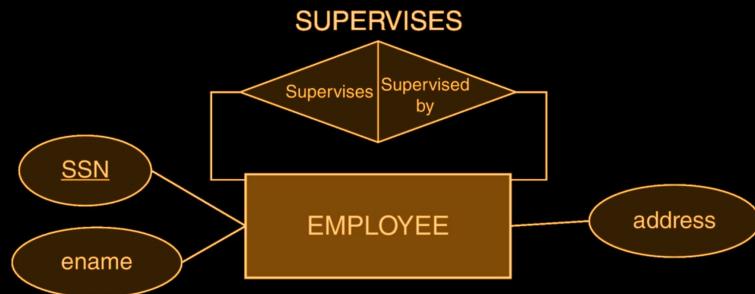
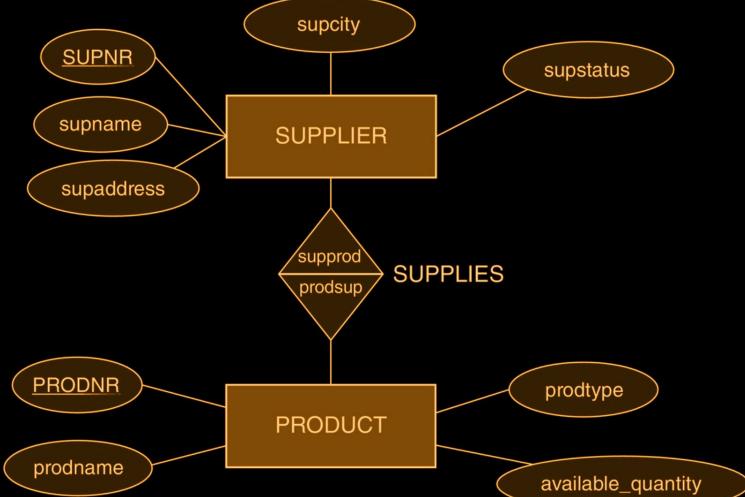
CREATE TABLE Product (
    Id INTEGER PRIMARY KEY,
    ...
    price FLOAT NOT NULL,
    ...
    CHECK (price > 0),
    ...
);
```

<https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-check-constraint/>

# Relationships

# Relationship Types

- Relate two or more entities (with roles)
- Ex: John majors in Computer Science
- Roles may be omitted when obvious



```
CREATE TABLE Supplies (
    SupNR INT REFERENCES Supplier,
    ProdNR INT REFERENCES Product,
    PRIMARY KEY (SupNR, ProdNR)
);
```

# ER-Diagram Exercise

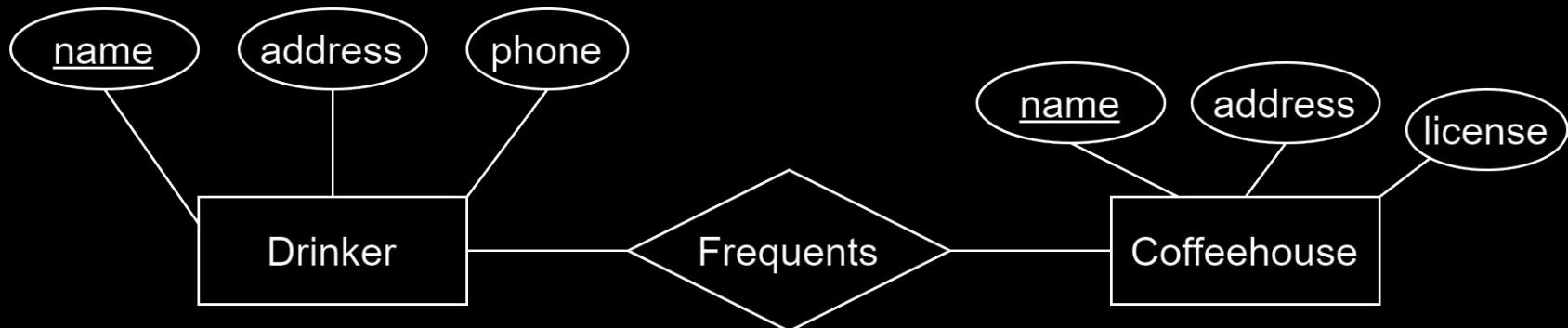
## DB Requirements:

- A drinker has a (unique) name, address, and phone.
- A coffeehouse has a (unique) name, address, and license.
- Store which drinkers frequent which coffeehouses.

# ER-Diagram Exercise

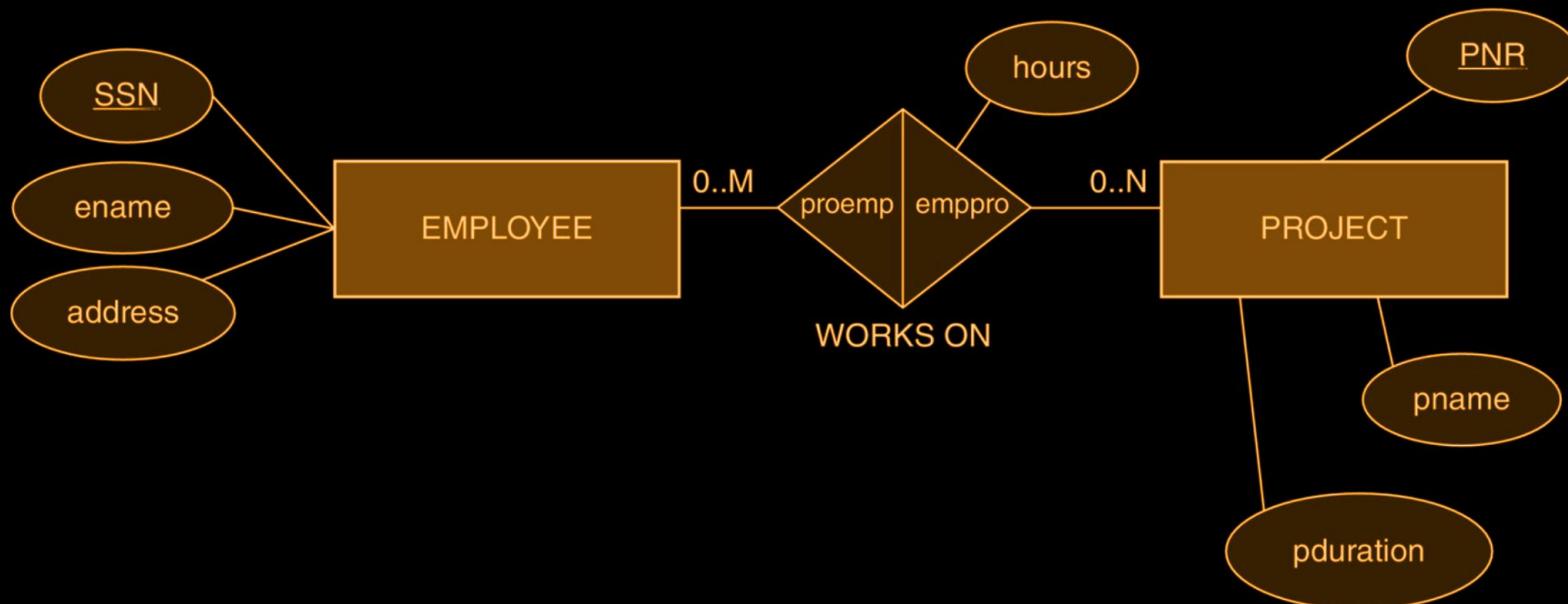
DB Requirements:

- A drinker has a (unique) name, address, and phone.
- A coffeehouse has a (unique) name, address, and license.
- Store which drinkers frequent which coffeehouses.



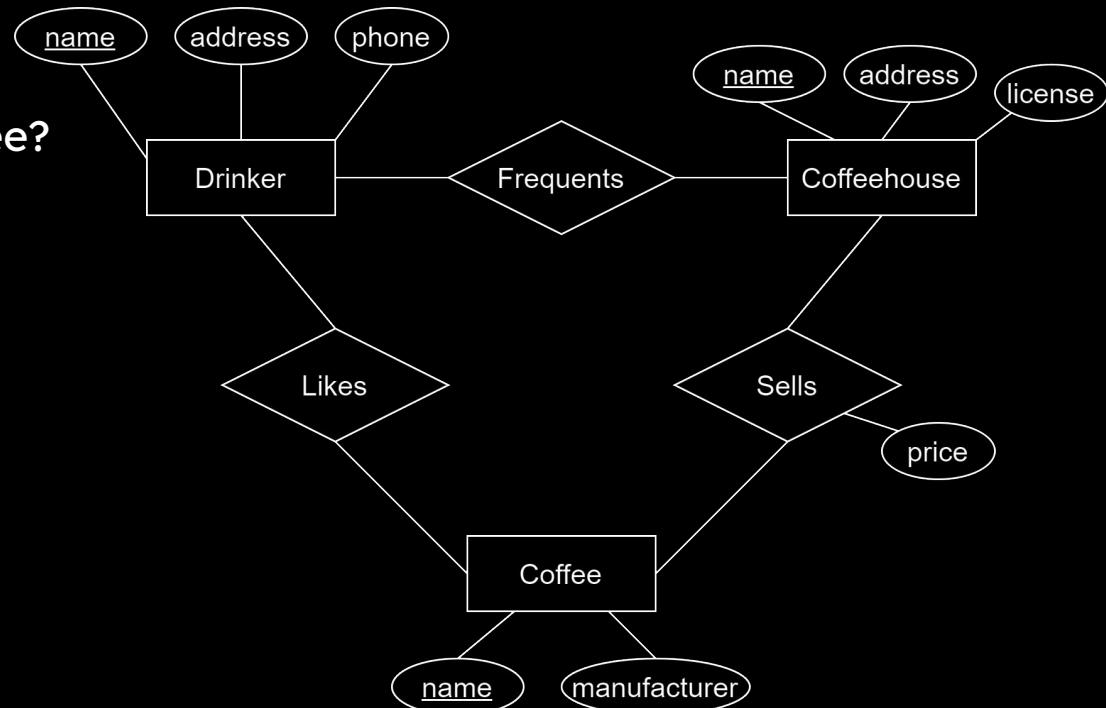
# Relationship Attribute Types

- Relationships can also have attribute types



# Example: Coffee DB

- What if...
  - price is an attribute of Coffee?
  - price is an attribute of Coffeehouse?



Coffees (name, manufacturer)

Coffeehouses (name, address, license)

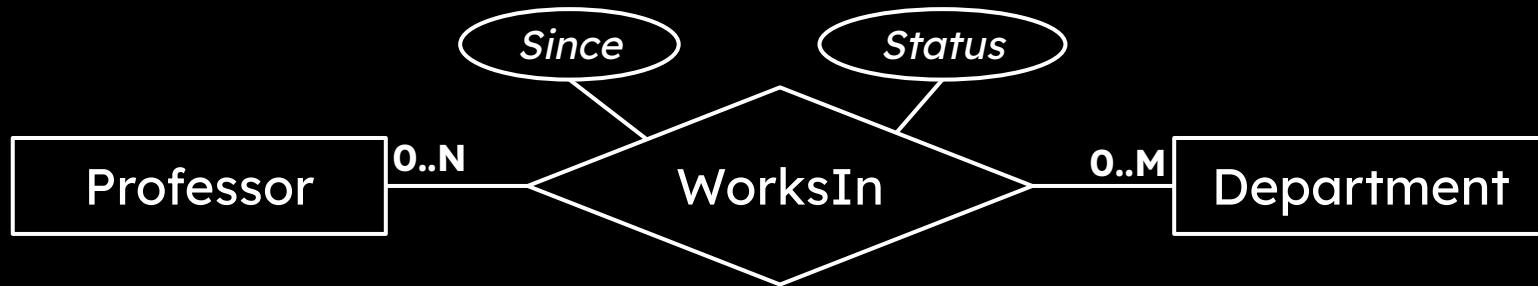
Drinkers (name, address, phone)

Likes (drinker, coffee)

Sells (coffeehouses, coffee, price)

Frequent (drinker, coffeehouse)

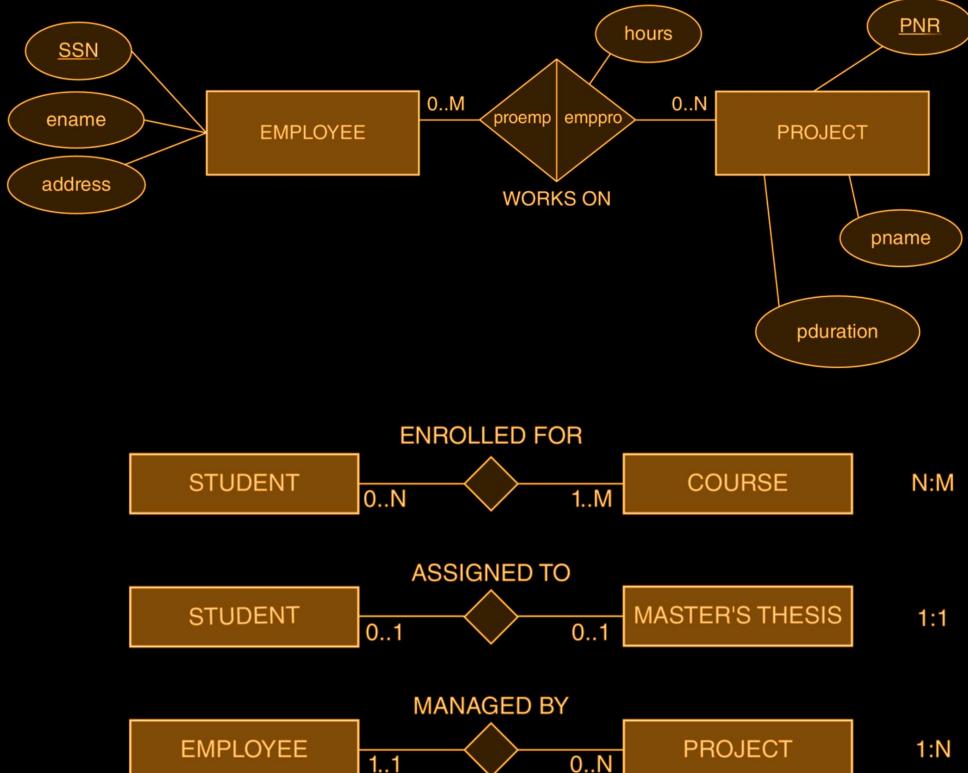
# Basic Relationship Table in SQL DDL



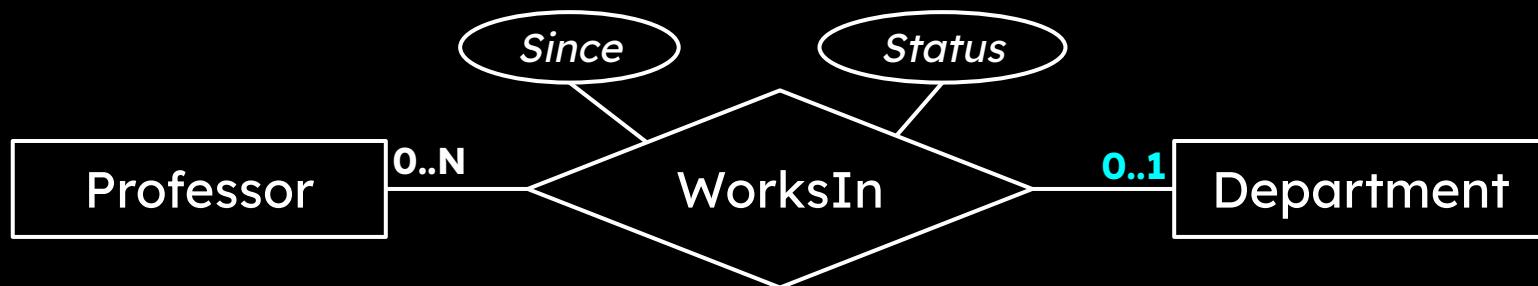
```
CREATE TABLE WorksIn (
    ProfID INT,                      -- role (key of Professor)
    DeptID CHAR(4),                  -- role (key of Department)
    Since DATE NOT NULL,            -- attribute
    Status CHAR(10) NOT NULL,        -- attribute
    PRIMARY KEY (ProfID, DeptID),
    FOREIGN KEY (ProfID) REFERENCES Professor (ID),
    FOREIGN KEY (DeptID) REFERENCES Department (ID)
)
```

# Cardinalities

- Relationships always have cardinalities
  - Minimum: 0 or 1
  - Maximum: 1 or N / M / P / \* / ...
- Read:
  - Entity (ignore) Relationship Cardinality Entity
  - Employee can work on zero to many projects
- Do cardinalities impact the resulting table structure?



# Maximum 1



```
CREATE TABLE WorksIn (
    ProfID INT,                      -- role (key of Professor)
    DeptID CHAR(4) NOT NULL,          -- role (key off Department)
    Since DATE NOT NULL,             -- attribute
    Status CHAR(10) NOT NULL,         -- attribute
    PRIMARY KEY (ProfID),            -- each professor only once
    FOREIGN KEY (ProfID) REFERENCES Professor (ID),
    FOREIGN KEY (DeptID) REFERENCES Department (ID)
)
```

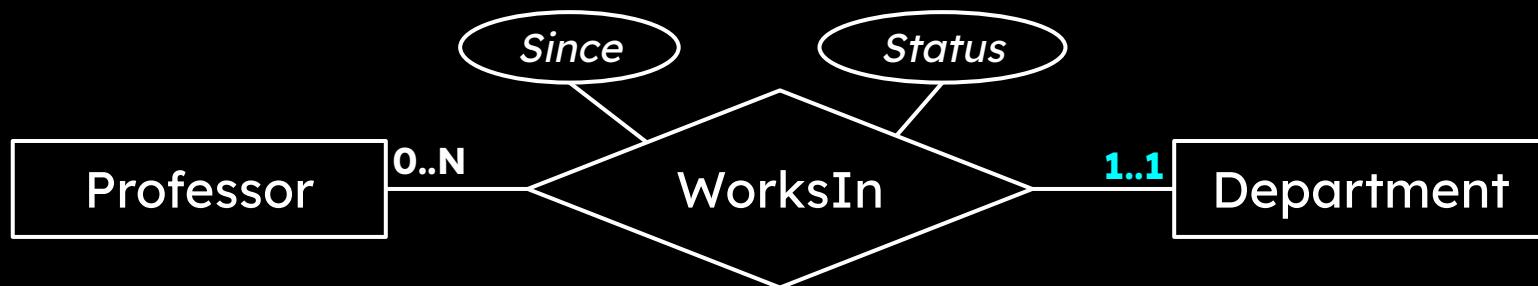
# Maximum 1 - No Attributes



~~CREATE TABLE WorksIn (....)~~

```
CREATE TABLE Professor (
    Id INT PRIMARY KEY,
    ...
    deptId INT, -- foreign key
    FOREIGN KEY (deptId) REFERENCES Department(Id)
)
```

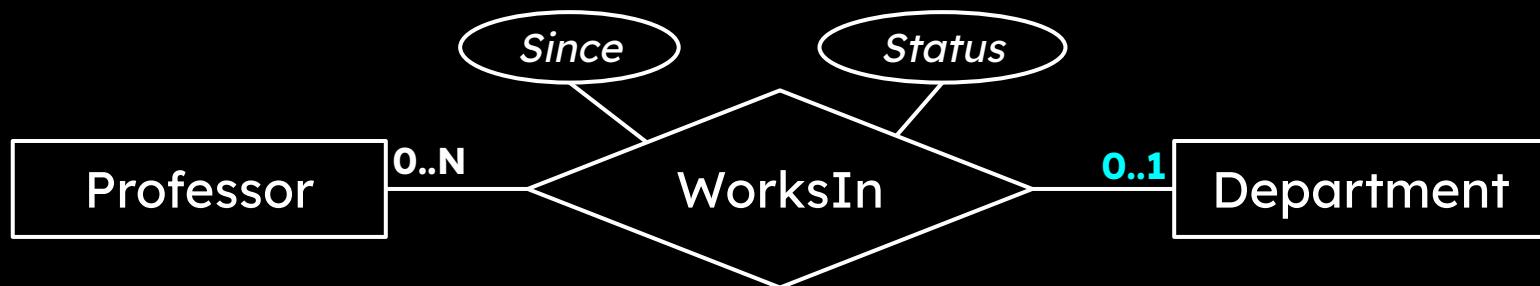
# Exactly 1



~~CREATE TABLE WorksIn (....)~~

```
CREATE TABLE Professor (
    Id INT PRIMARY KEY,
    ...
    deptId INT NOT NULL, -- foreign key
    Since DATE NOT NULL, -- attribute
    Status CHAR(10) NOT NULL, -- attribute
    FOREIGN KEY (deptId) REFERENCES Department(Id)
)
```

# Maximum 1 - Revisited

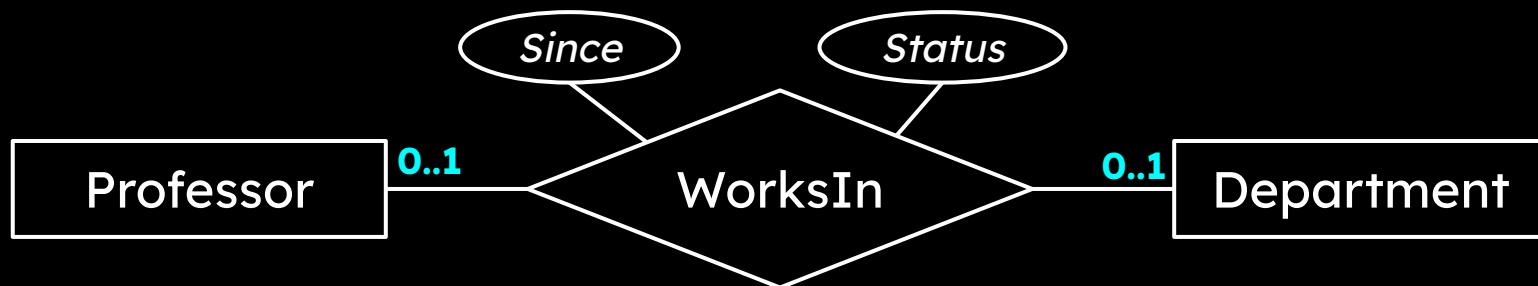


Could we do the same for 0..1?

- Yes, but the three attributes must be able to be NULL
- **We only do this if the relationship has no attributes**

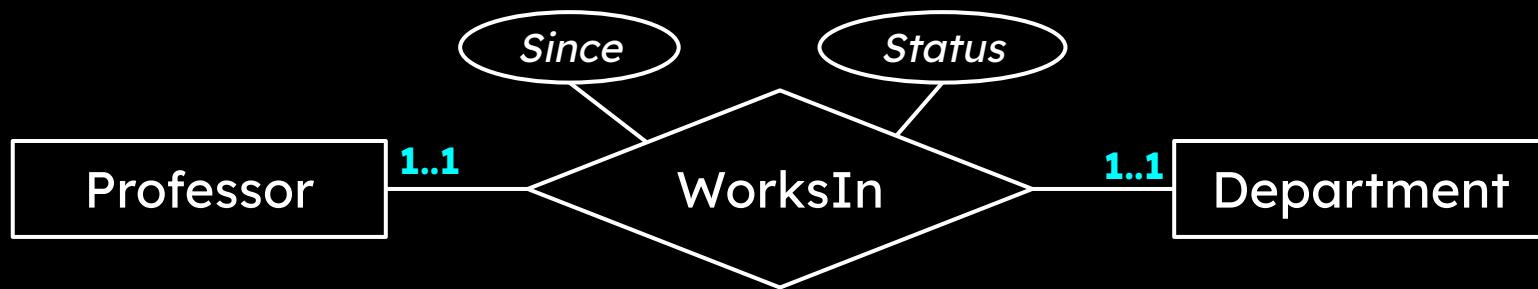
```
CREATE TABLE Professor (
    Id INT PRIMARY KEY,
    ...
    deptId INTEGER,      foreign key
    Since DATE,          attribute
    Status CHAR(10),     attribute
    FOREIGN KEY (deptId) REFERENCES Department(Id)
)
```

# Maximum 1 - Both directions



```
CREATE TABLE WorksIn (
    ProfID INT,                      -- role (key of Professor)
    DeptID CHAR(4) NOT NULL,          -- role (key of Department)
    Since DATE NOT NULL,             -- attribute
    Status CHAR(10) NOT NULL,         -- attribute
    PRIMARY KEY (ProfID),            -- each professor only once
    UNIQUE (DeptID),                -- each department only once
    FOREIGN KEY (ProfID) REFERENCES Professor(ID),
    FOREIGN KEY (DeptID) REFERENCES Department(ID)
)
```

# Exactly 1 - Both direction



```
CREATE TABLE Professor (
    Id INT PRIMARY KEY,
    ...
    DeptId INT NOT NULL,
    Since DATE NOT NULL,
    Status CHAR(10) NOT NULL,
    FOREIGN KEY (DeptId)
    REFERENCES Department(Id)
)
```

```
CREATE TABLE Department (
    Id INT PRIMARY KEY,
    ...
    ProfId INT NOT NULL,
    FOREIGN KEY (ProfId)
    REFERENCES Professor(Id)
)
```

# Exactly 1 - Both direction

- Can we use FK on both sides?
  - Yes... but it is neither easy nor portable
- Think about inserting the first prof and dept
  - Which comes first... Chicken or the egg?
- Alternative 1: Use Deferred FK or Trigger
  - Runs at the end of a transaction - many systems support neither!

```
CREATE TABLE Professor (
    Id INT PRIMARY KEY,
    ...
    DeptId INT NOT NULL,
    Since DATE NOT NULL,
    Status CHAR(10) NOT NULL,
    FOREIGN KEY (DeptId)
    REFERENCES Department(Id)
)
```

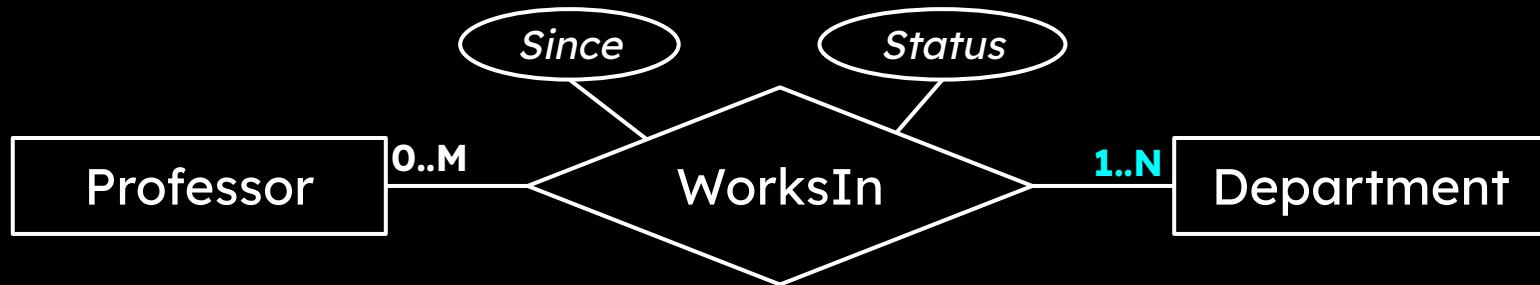
```
CREATE TABLE Department (
    Id INT PRIMARY KEY,
    ...
    ProfId INT NOT NULL,
    FOREIGN KEY (ProfId)
    REFERENCES Professor(Id)
)
```

# Exactly 1 - Both directions

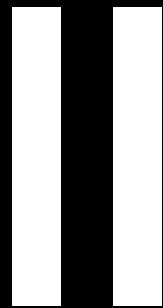
- Alternative 2: Merge Tables
  - May work well in some cases
  - Depends on entities
- Alternative 3: Pick one FK direction
  - Write down the other requirement
  - Do the best in software with the other direction

```
CREATE TABLE Professor (
    Id INT PRIMARY KEY,
    ...
    deptId INT NOT NULL,
    deptName VARCHAR NOT NULL,
    ...
    Since DATE NOT NULL,
    Status CHAR(10) NOT NULL
)
```

# Minimum 1 - Maximum N



- What about 1..N or 1..M cardinalities?
  - Or when requirements demand particular fixed numbers?
- No support in SQL DDL
  - Could use a trigger on Professor/WorksIn
- Normally:
  - Write down the requirement
  - Do the best in software



# Exercise

- Draw this schema as ER diagram...

BUT:

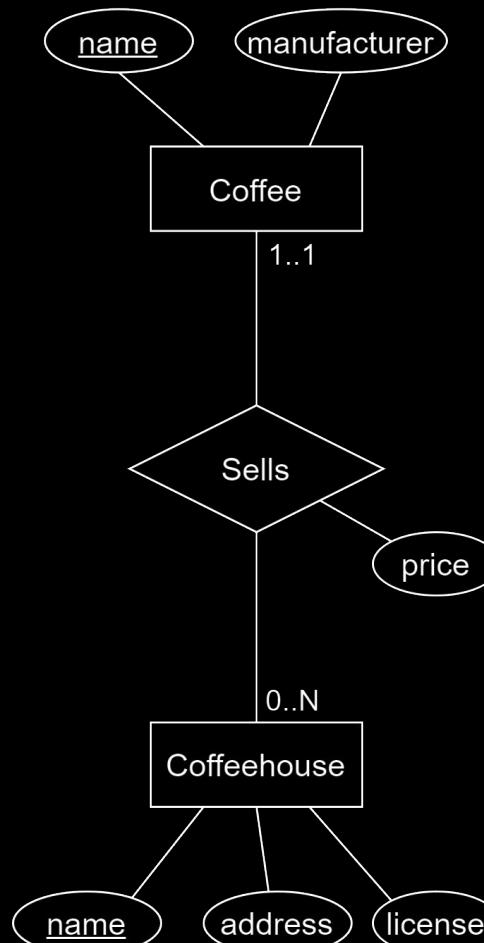
- Use IDs
- Assume each coffeehouse sells exactly one coffee

Coffees(name, manf)

Coffeehouses(name, addr, license)

Sells(coffeehouse, coffee, price)

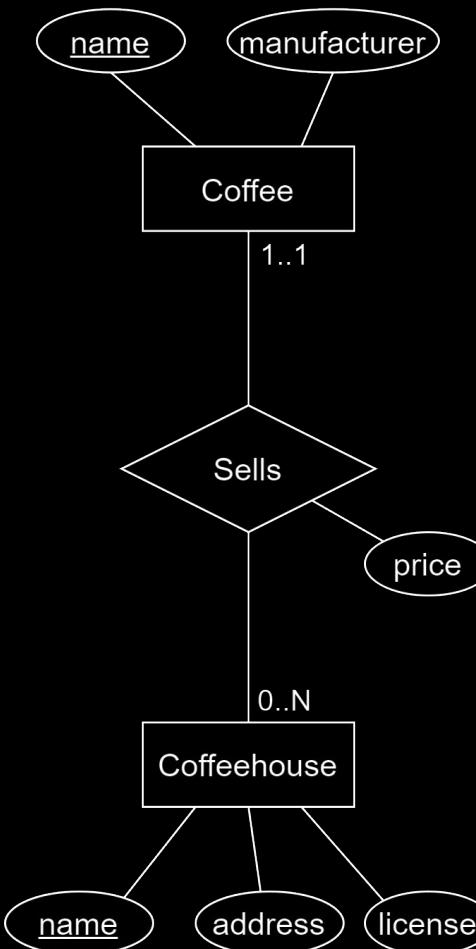
- Write SQL DDL to create the tables
  - How many tables?



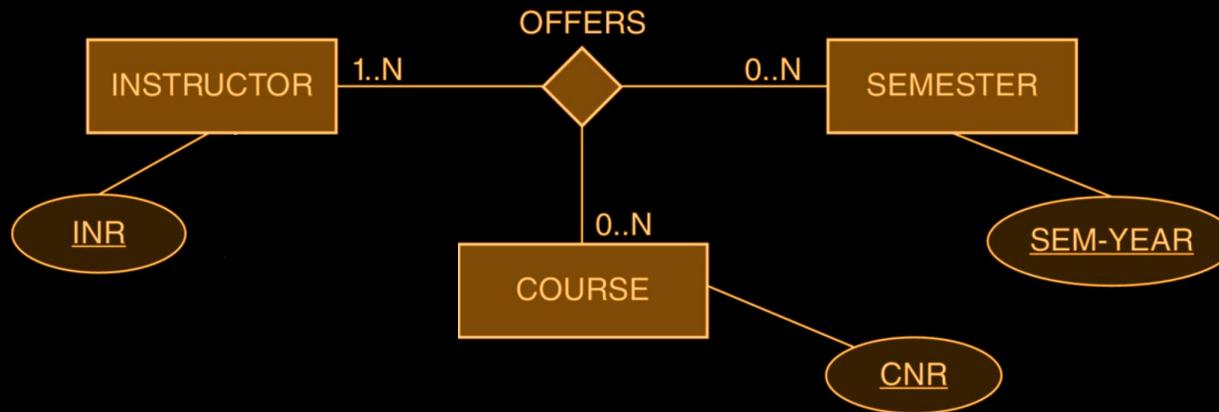
# Exercise Answer DDL

```
CREATE TABLE Coffee (
    ID INT PRIMARY KEY,
    name VARCHAR NOT NULL,
    manf VARCHAR NOT NULL
);

CREATE TABLE Coffeehouse (
    ID INT PRIMARY KEY,
    name VARCHAR NOT NULL,
    addr VARCHAR NOT NULL,
    lic VARCHAR NOT NULL,
    coffeeID INT NOT NULL REFERENCES Coffee,
    price INT NOT NULL
);
```



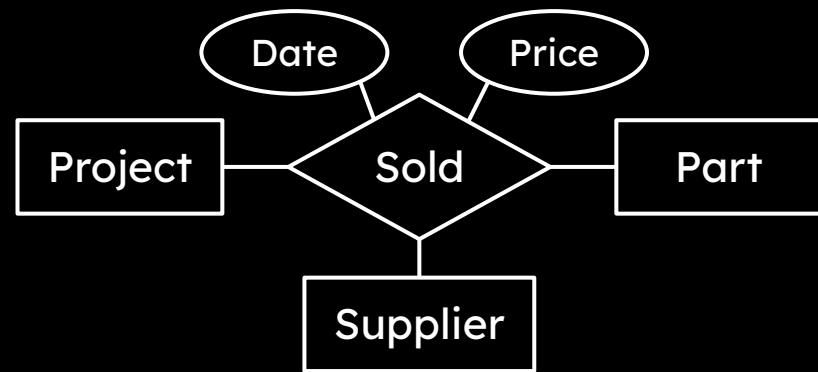
# Ternary Relationships (and Beyond)



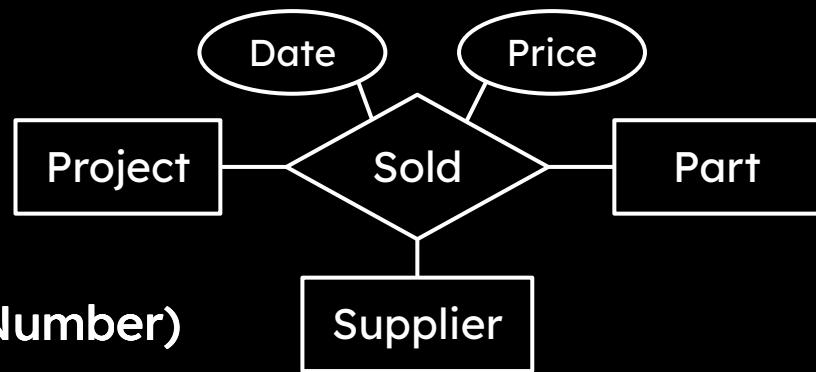
- An Instructor can offer many Courses during a Semester
- A Course offered in a given Semester must have at least one Instructor involved
- An Instructor can offer a given Course for many Semesters

# Example: DLL of Ternary Relationship

```
CREATE TABLE Sold (
    ProjID INT, -- role
    SupplierID INT, -- role
    PartNumber INT, -- role
    Date DATE NOT NULL, -- attribute
    Price FLOAT NOT NULL -- attribute
    | PRIMARY KEY (ProjID, SupplierID, PartNumber),
    FOREIGN KEY (ProjID) REFERENCES Project (ID),
    FOREIGN KEY (SupplierID) REFERENCES Supplier (ID),
    FOREIGN KEY (PartNumber) REFERENCES Part (Number)
```



# Extension: Partial Relationship Keys



**PRIMARY KEY (ProjID, SupplierID, PartNumber)**

- What does the key of the relationship table mean?
- What if the part should be sold many times?
- The book: No discussion! :(
- Our notation: *Partial relationship keys*
  - Attribute is underlined

# Extension: Partial Relationship Keys in SQL DDL

```
CREATE TABLE Sold (
```

```
    ProjID INT, -- role
```

```
    SupplierID INT, -- role
```

```
    PartNumber INT, -- role
```

```
    Date DATE NOT NULL, -- attribute
```

```
    Price FLOAT NOT NULL -- attribute
```

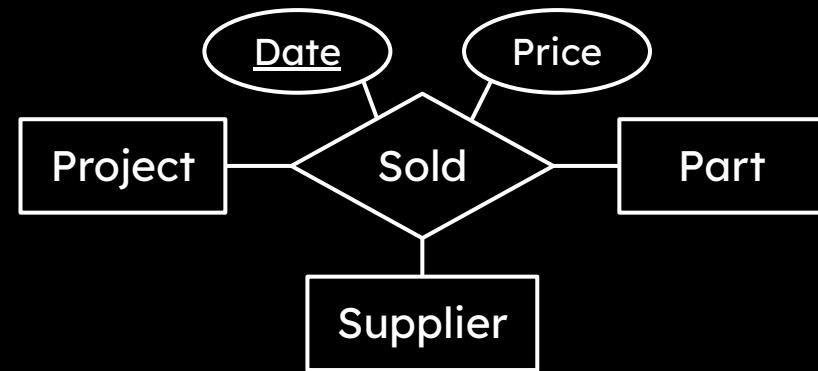
```
    | PRIMARY KEY (ProjID, SupplierID, PartNumber, Date),
```

```
    | FOREIGN KEY (ProjID) REFERENCES Project (ID),
```

```
    | FOREIGN KEY (SupplierID) REFERENCES Supplier (ID),
```

```
    | FOREIGN KEY (PartNumber) REFERENCES Part (Number)
```

```
)
```

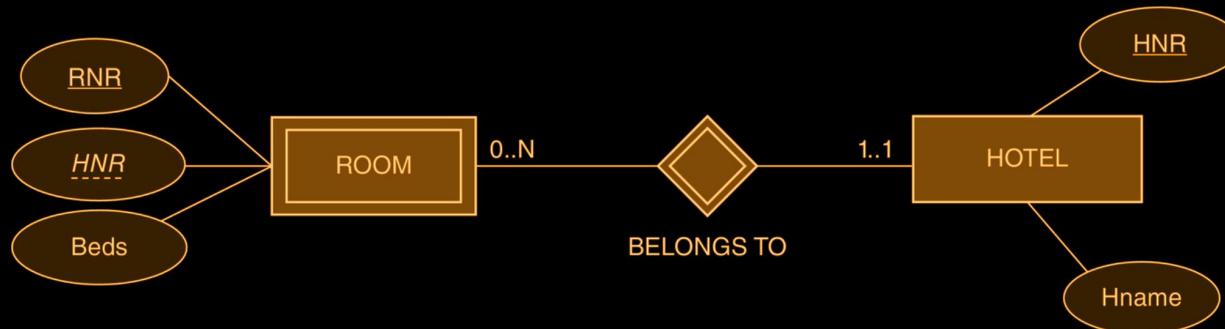


# -- TODO

- Conceptual Data Modeling (ER Diagram)
  - ✓ Entities and Attributes
  - ✓ Relationships
    - ✓ Cardinalities
    - ✓ Partial Relationship Keys
  - Weak Entities
  - Aggregation
  - Generalization/Specialization
  - Categorization
- Translation to SQL DDL

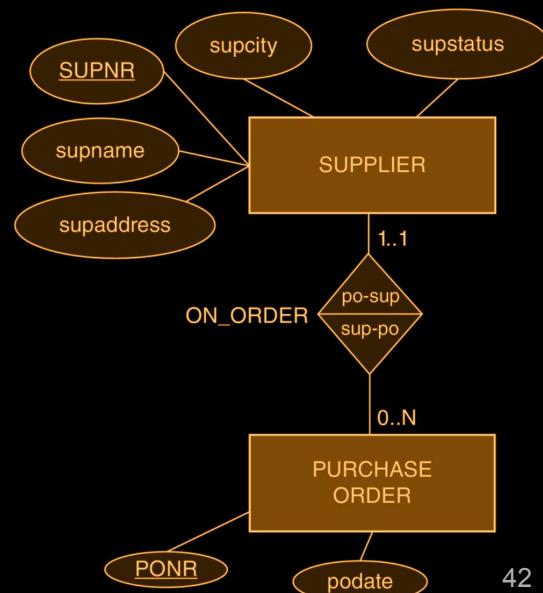
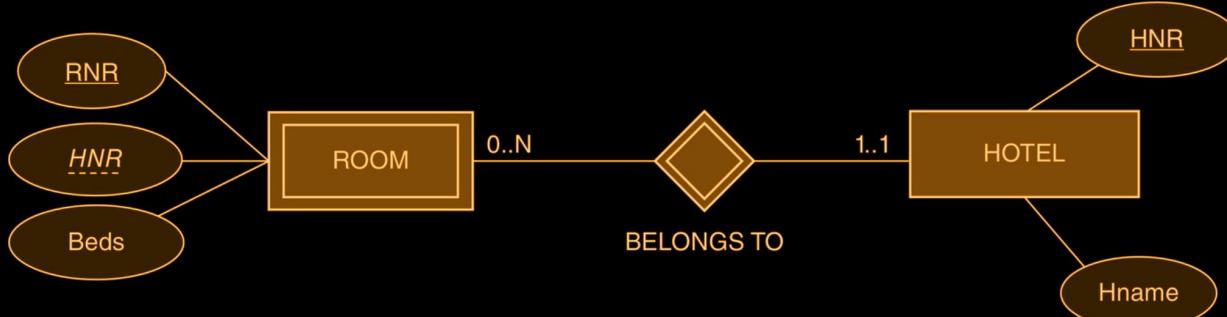
# Weak Entity Types

- Weak entities belong to another entity
  - They do not have a proper key, but include “parent” key
  - They have a 1 .. 1 participation in the relationship
  - If “parent” is deleted, so is the “child”
- Representation
  - Double outlines (entity, relationship)
  - Parent key is underlined with dashes



# Weak Entity Types vs 1..1 Relationship Types

- Weak entities have a 1..1 relationship type
  - Some 1..1 relationship types represent weak entities
  - Most 1..1 relationship types do not represent weak entities
- Main difference is presence of a natural key
  - Weak entity:  
Only unique within the parent entity!

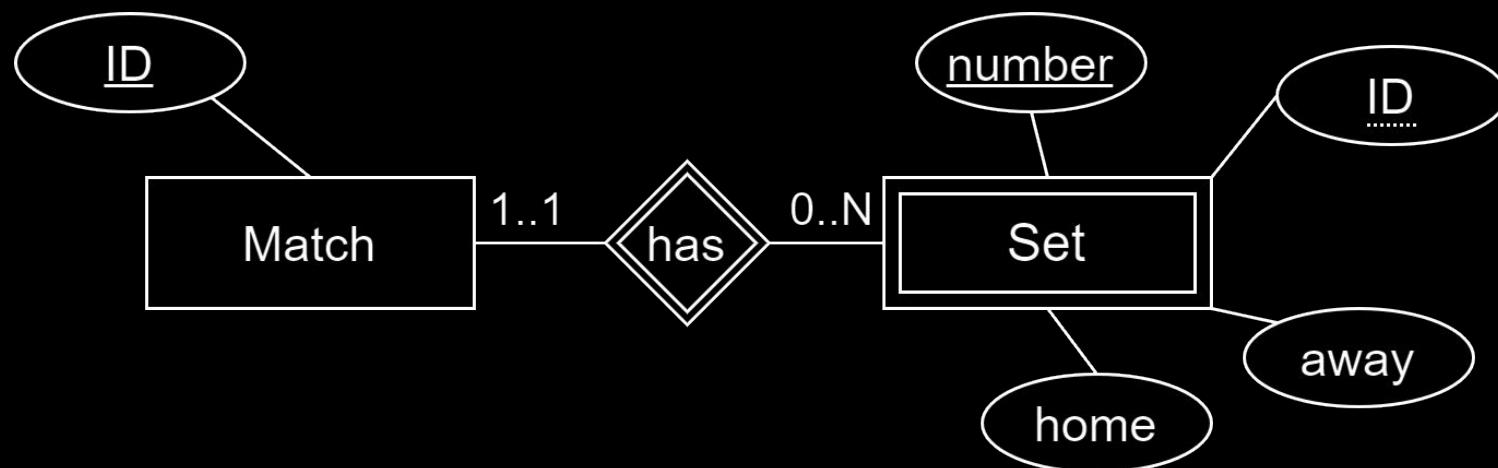


# Practice Weak Entities

- Each Volleyball Match consists of Sets, each set has a number, home score and away score

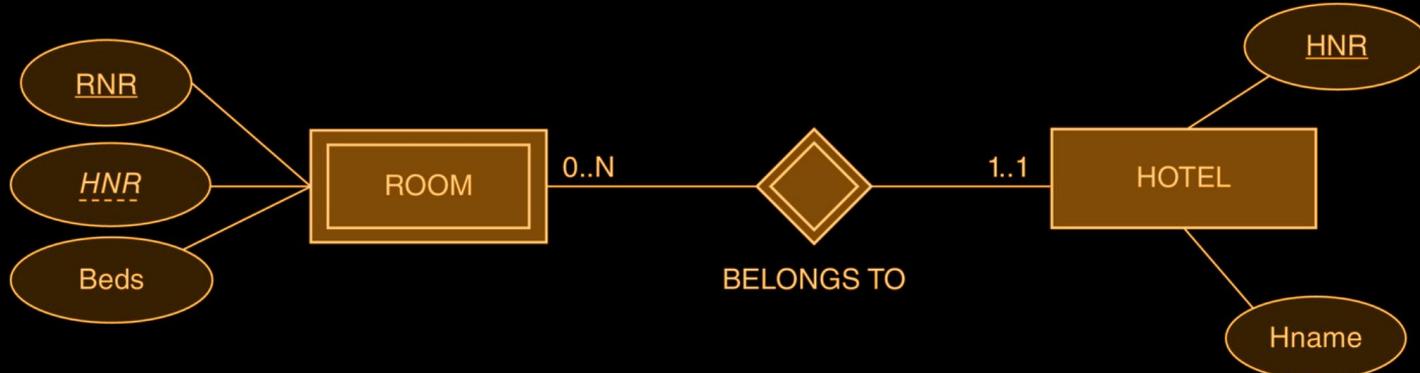
# Practice Weak Entities

- Each volleyball match consists of sets, each with set number, home score and away score



# Weak Entities in SQL DDL

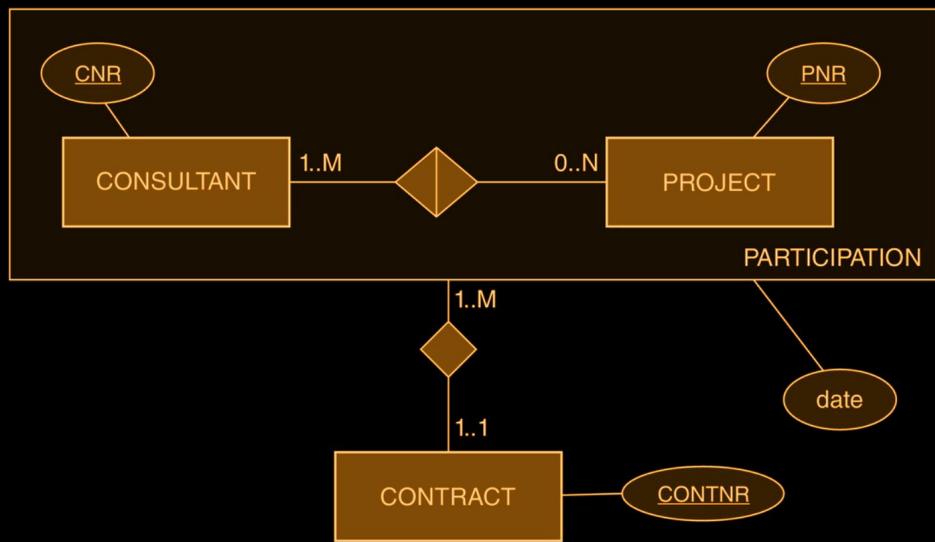
- Create a new table referencing the entity table
  - Primary key is parent key + partial key
  - Very similar to multi-valued attributes



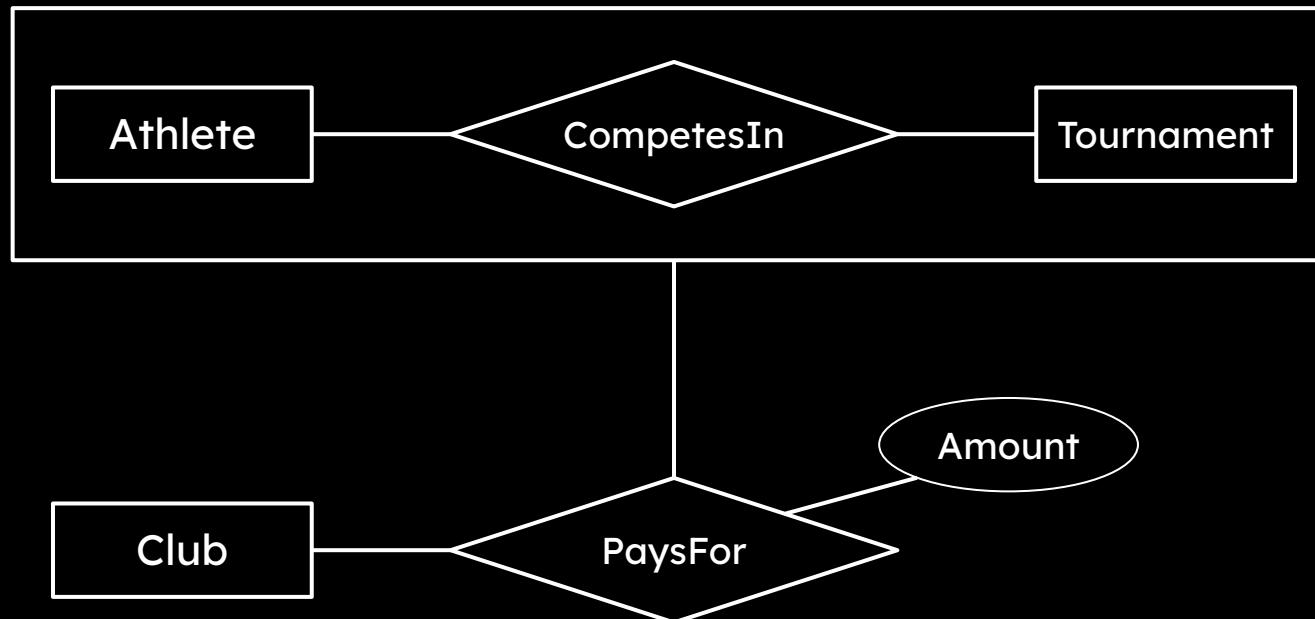
```
CREATE TABLE Room (
    RNR INT, -- Should not be a sequence!
    HNR INT, -- Must be a FOREIGN KEY!
    Beds INT NOT NULL,
    PRIMARY KEY (HNR, RNR),
    FOREIGN KEY REFERENCES Hotel (HNR),
);
```

# Aggregation

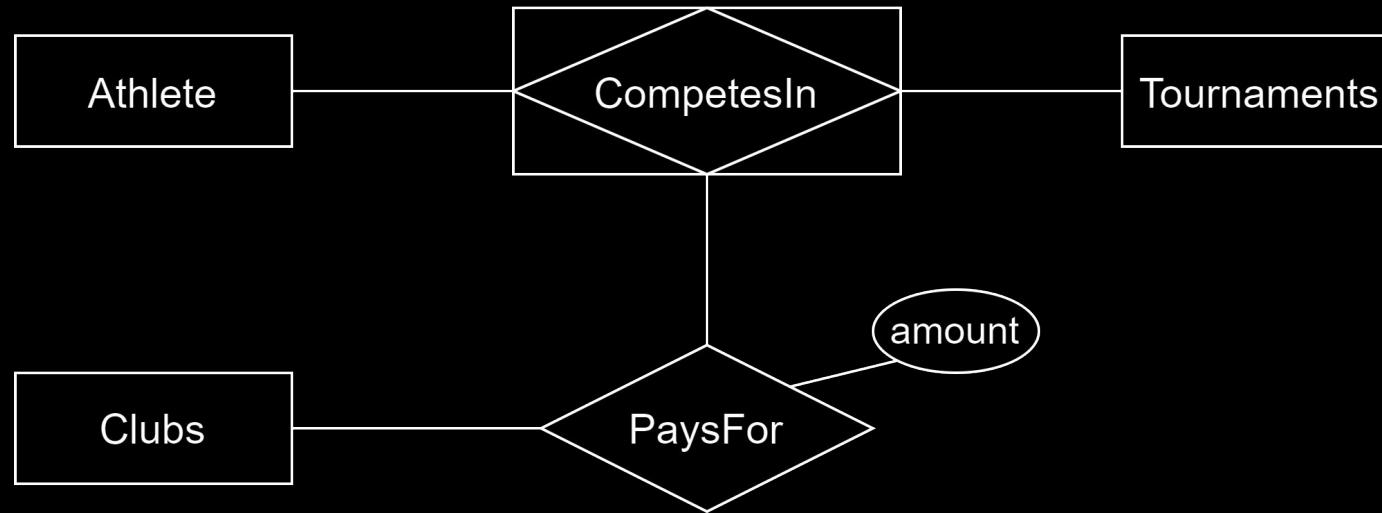
- Sometimes we need a relationship to a relationship
- Aggregation allows us to “convert” relationship types to entity types!
- Typical use: Monitoring, payments, contracts



# Relationship → Entity

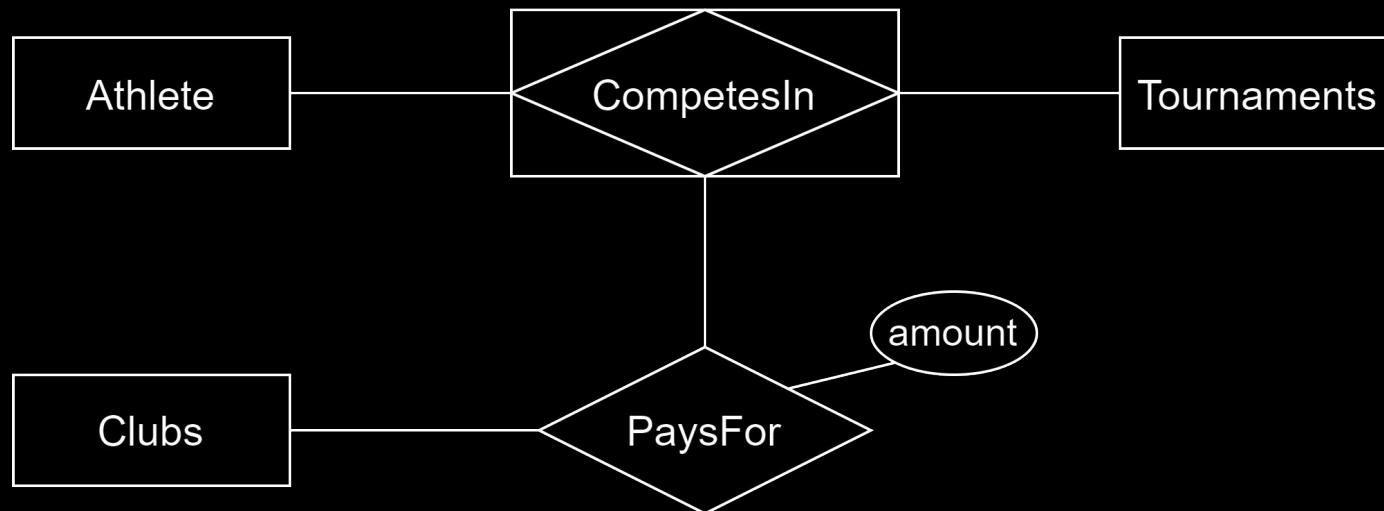


# Relationship → Entity



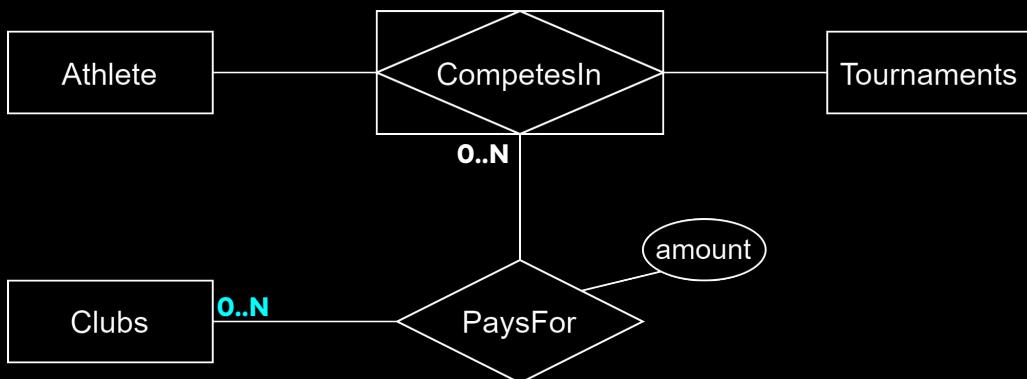
# Translation to SQL DDL

- Main observation: It is a relationship!
  - Use same method as translating relationship
  - Treat the aggregation table as the entity



# Translation to SQL DDL: 0..N

- Option 1: Use existing relationship key

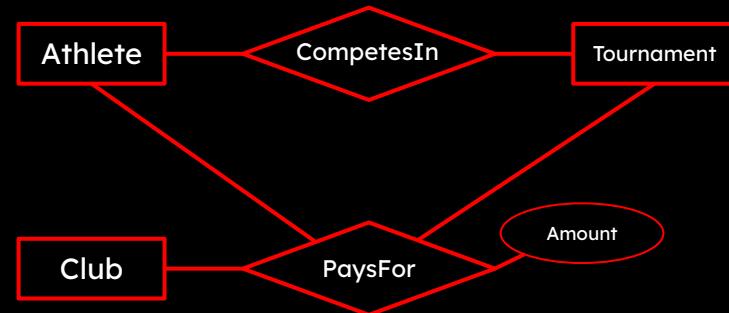
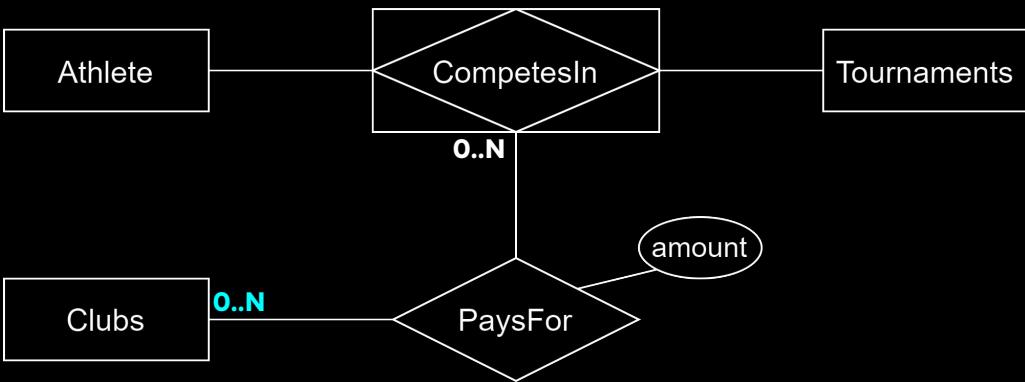


```
CREATE TABLE CompetesIn (
    AID INT REFERENCES Athlete,
    TID INT REFERENCES Tournament,
    PRIMARY KEY (AID, TID)
);

CREATE TABLE PaysFor (
    AID INT,
    TID INT,
    CID INT REFERENCES Clubs,
    amount INT NOT NULL,
    FOREIGN KEY (AID, TID)
        REFERENCES CompetesIn (AID, TID),
    PRIMARY KEY (AID, TID, CID)
);
```

# Entities and Aggregation: Impact on Tables

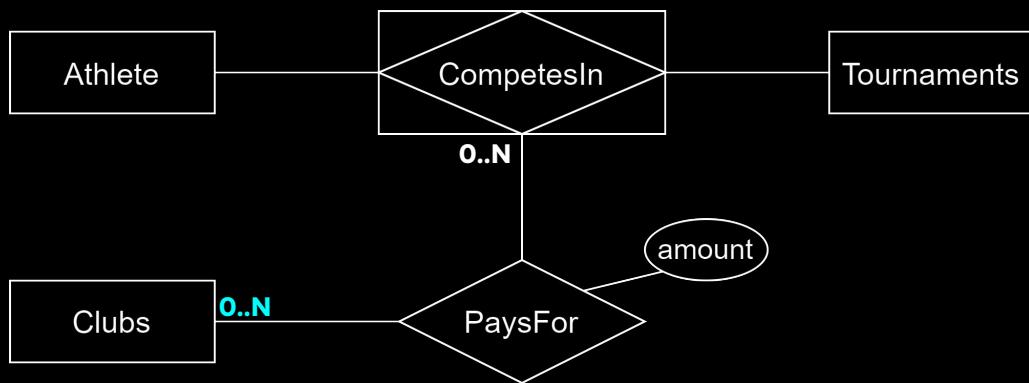
- A common error is to use FKs to entity tables



```
CREATE TABLE PaysFor (
    AID INT REFERENCES Athlete,
    TID INT REFERENCES Tournament,
    CID INT REFERENCES Clubs,
    amount INT NOT NULL,
    FOREIGN KEY (AID, TID)
    REFERENCES CompetesIn (AID, TID),
    PRIMARY KEY (AID, TID, CID),
);
```

# Translation to SQL DDL: 0..N

- Option 2: Create a new relationship key
  - Common error is to forget the existing key

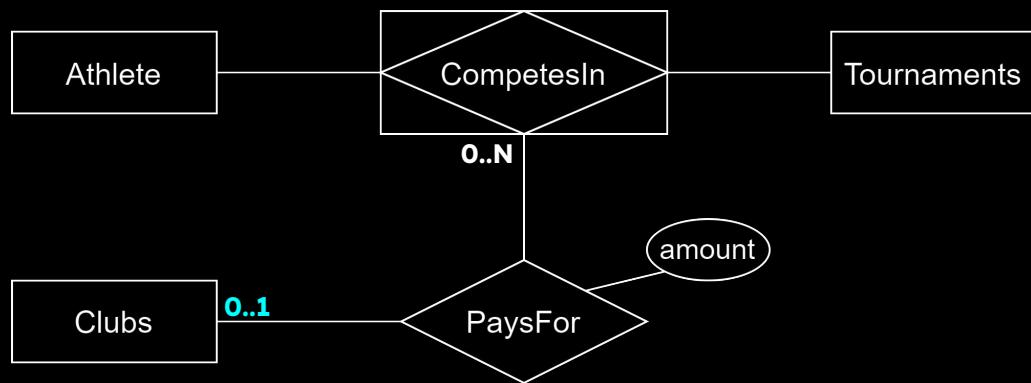


```
CREATE TABLE CompetesIn (
    CIID INT ALWAYS GENERATED AS IDENTITY
        PRIMARY KEY,
    AID INT REFERENCES NOT NULL Athletes,
    TID INT REFERENCES
        NOT NULL Tournament,
    UNIQUE (AID, TID)
);
```

```
CREATE TABLE PaysFor (
    CIID INT REFERENCES CompetesIn,
    CID INT REFERENCES Clubs,
    amount INTEGER NOT NULL,
    PRIMARY KEY (CIID, CID)
);
```

# Translation to SQL DDL: 0..1

- An Athlete competing in a Tournament can receive payment from maximum one Club
  - Change the Primary Key

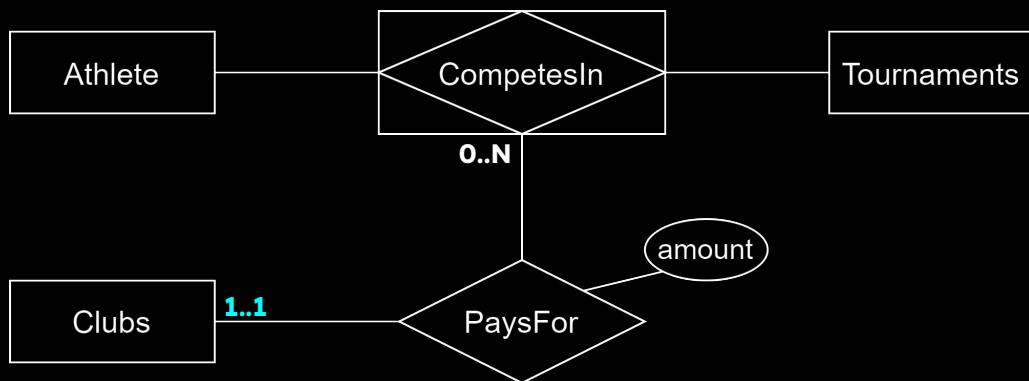


```
CREATE TABLE CompetesIn (
    AID INT REFERENCES Athletes,
    TID INT REFERENCES Tournament,
    PRIMARY KEY (AID, TID)
);

CREATE TABLE PaysFor (
    AID INT,
    TID INT,
    CID INT NOT NULL REFERENCES Clubs,
    amount INT NOT NULL,
    FOREIGN KEY (AID, TID)
    REFERENCES CompetesIn (AID, TID),
    PRIMARY KEY (AID, TID)
);
```

# Translation to SQL DDL: 1..1

- Change the relationship table
- This is the only case covered in the book!
- Here: No PaysFor table!



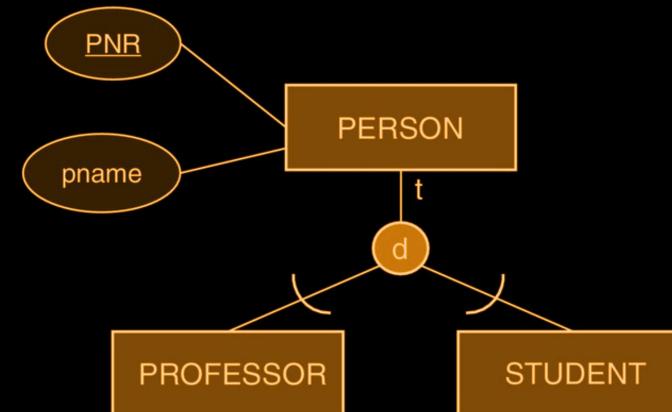
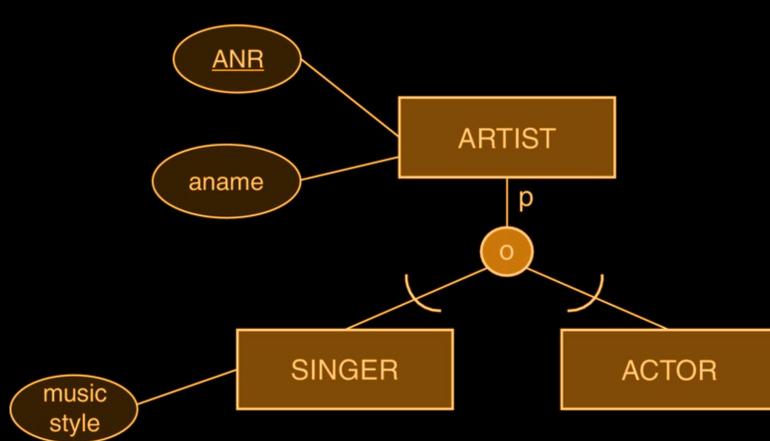
```
CREATE TABLE CompetesIn (
    AID INT REFERENCES Athletes,
    TID INT REFERENCES Tournament,
    CID INT NOT NULL REFERENCES Clubs,
    amount INT NOT NULL
    PRIMARY KEY (AID, TID)
);
```

# -- TODO -> DONE

- ✓ Conceptual Data Modeling (ER Diagram)
  - ✓ Entities and Attributes
  - ✓ Relationships
    - ✓ Cardinalities
    - ✓ Partial Relationship Keys
  - ✓ Weak Entities
  - ✓ Aggregation
    - Generalization/Specialization
    - Categorization
- Translation to SQL DDL

# Generalization/Specialization

- Like inheritance in Object-Oriented Programming (Java/C#/etc.)
- Partial vs Total = p/t on the line
- Overlapping vs Disjoint = o/d in the circle
- Please don't use colour in homeworks/exam
- Arcs matter → need to draw them (in the exam)!



# Specialization in SQL DDL

- One table for super-type, one per sub-type
  - The PK of supertype is also PK for all subtypes
  - Each subtype has a FK to the supertype
- Preferred option by far!
  - Redundancy is eliminated:
  - Name and DOB are stored only once
  - Adjusts well to hierarchies/lattices

Person			Employee			Student		
SSN	Name	DOB	SSN	Department	Salary	SSN	GPA	StartDate
1234	Mary	1950	1234	Accounting	35000	1234	3.5	1997

# Specialization in SQL DDL

```
CREATE TABLE Person (
    SSN INT PRIMARY KEY,
    Name VARCHAR NOT NULL,
    DOB DATE NOT NULL
);
```

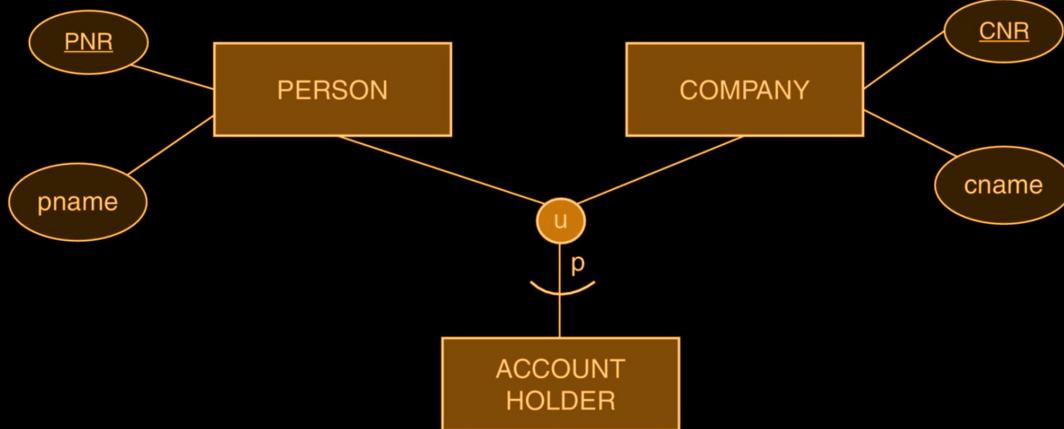
Person			Employee			Student		
SSN	Name	DOB	SSN	Department	Salary	SSN	GPA	StartDate
1234	Mary	1950	1234	Accounting	35000	1234	3.5	1997

```
CREATE TABLE Employee (
    SSN INT PRIMARY KEY REFERENCES Person,
    Department VARCHAR NOT NULL,
    Salary INTEGER NOT NULL
);
```

```
CREATE TABLE Student (
    SSN INT PRIMARY KEY REFERENCES Person,
    GPA REAL NOT NULL,
    StartDate DATE NOT NULL
);
```

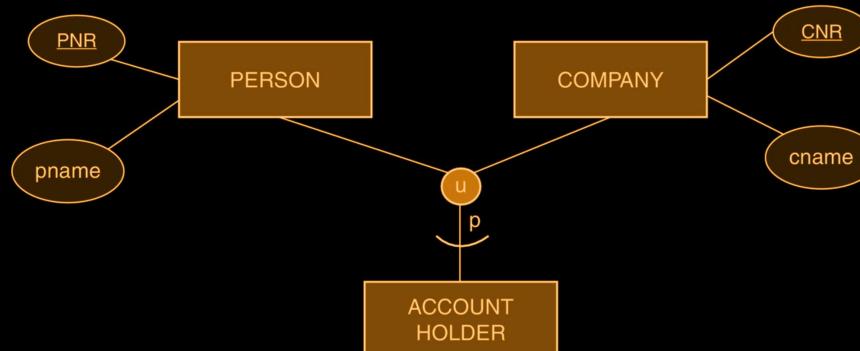
# Categorization

- **Grouping of otherwise unrelated entities**
- New entity is a union = u in the circle
  - Can be total or partial = p/t on the line
- Can be checked with triggers similarly to specialization
- Notice the direction of the arc to the union
  - Refers to flow of “inheritance” – or which comes first...
  - Arcs matter → need to draw them (in the exam)!



# Categorization in SQL DDL

- New table for the categorical entity
  - New abstract PK with INTEGER / IDENTITY
- Add the PK as attribute to the other entities
  - With FK to the categorical entity
- Why is this important?
  - Sometimes AccountHolder participates in relationships...



```
CREATE TABLE AccountHolder (  
    AcctID INT PRIMARY KEY  
)
```

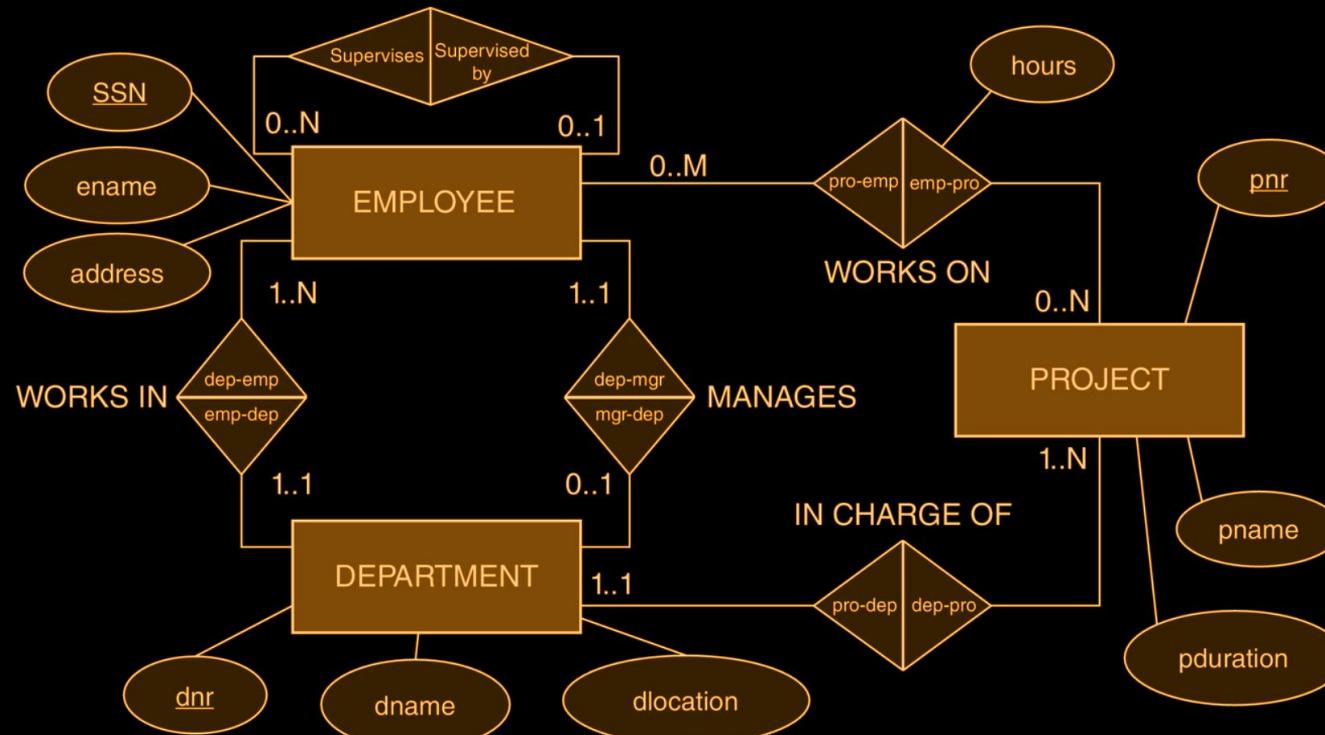
```
CREATE TABLE Person (  
    PNR INT PRIMARY KEY,  
    pname VARCHAR NOT NULL,  
    AcctID INT [NOT NULL]  
    REFERENCES AccountHolder (AcctID)  
)
```

NOT NULL = Total  
NULL = Partial

# -- TODO -> DONE

- ✓ Conceptual Data Modeling (ER Diagram)
  - ✓ Entities and Attributes
  - ✓ Relationships
    - ✓ Cardinalities
    - ✓ Partial Relationship Keys
  - ✓ Weak Entities
  - ✓ Aggregation
  - ✓ Generalization/Specialization
  - ✓ Categorization
- ✓ Translation to SQL DDL

# Exercise: How many tables? What are the keys?



# Requirements to ERD

- Nouns = entities
  - Descriptive elements = attributes
- Verbs = relationships
  - Descriptive elements = attributes
  - Look for words implying participation constraints
  - No words → 0..N
- Example:
  - Professors have an SSN, a name, an age, a rank, and a research specialty.
  - Projects have a project number, a sponsor name (e.g., NSF), a starting date, ...
  - Each project is managed by one professor (1..1 on profs, 0..N on projs)
  - Professors may work on many projects (0..\* on both sides)
  - Each project must be reviewed by some professors (1..N on profs, 0..N on projs)

# Dealing with very large ER Diagrams

- Method 1: Very large paper!
  - One diagram with all the details
- Method 2: Outline + Details
  - One diagram with main entities and their relationships
  - One diagram per entity with attributes and weak entities
- Method 3: Components
  - Break the model into components
  - Details inside components
  - Some edge entities are repeated (details in one place)

# Limitations of ER Design

- ER diagrams do not capture all design details
  - Example: Multiple candidate keys
  - Must note missing details somewhere!
- Some aspects do not map well to SQL DDL
  - Example: 1..M cardinalities
  - Triggers (Lecture 4) can be used to handle some problems
  - Normalization (Lecture 6) provides a mechanism for fixing some problems
  - Some must simply be noted and addressed in code or ignored!

# Summary of Notation Extensions

- Relationship roles are generally unnecessary
  - No need to label roles
  - Except for unary relationship types
  - May put relationship name inside rhombus
  - Except for unary relationship types
- We allow partial keys of relationships
  - Underlined relationship attribute
  - Part of the PK of the resulting relationship relation
- Aggregation entity may cover only the relationship
  - Much easier to read!
- Allow 0..\* in place of 0..N/M/L
  - ... or simply use 0..N everywhere!

## ER diagram captures entities and relationships

- Weak entities, specialization, categorization and aggregation allow capturing more detailed model characteristics
- This is hard – but also useful – so you must practice!

# Takeaways

one-to-zero  
one-to-one  
one-to-many  
many-to-many

## Conversion to SQL DDL

- Entities and relationships mapped to relations
- Essentially an algorithmic process (with some options)
- This is hard – but also useful – so you must practice!

## Notation: MANY VARIANTS EXIST!

- We use the one from the book (as extended in lecture)
- You must use this notation in the homework and exam!!!

**Next Time in IDBS...**

# Introduction to Database Systems

## IDBS - Fall 2024

### Lecture 6

#### Normalization

---

Readings: PDBM 6.2-6.4

Eleni Tzirita Zacharatou

# Introduction to Database Systems

## IDBS – Fall 2024

- Week 6:
- Normalization

Eleni Tzirita Zacharatou

Readings

PDBM 6.2

# Database Design

How do we design a “good” database schema?

We want to ensure the integrity of the data.

We also want to get good performance.

# Example Database

---

**student(sid,cid,room,grade,name,address)**

sid	cid	room	grade	name	address
123	15-445	GHC6115	A	Andy	Pittsburgh
456	15-721	GHC8102	B	Tupac	Los Angeles
789	15-445	GHC6115	A	Obama	Chicago
012	15-445	GHC6115	C	Waka Flocka	Atlanta
789	15-721	GHC8102	A	Obama	Chicago

# Example Database

---

**student(sid,cid,room,grade,name,address)**

sid	cid	room	grade	name	address
123	15-445	GHC6115	A	Andy	Pittsburgh
456	15-721	GHC8102	B	Tupac	Los Angeles
789	15-445	GHC6115	A	Obama	Chicago
012	15-445	GHC6115	C	Waka Flocka	Atlanta
789	15-721	GHC8102	A	Obama	Chicago

# Redundancy Problems

---

## Update Anomalies

- If the room number changes, we need to make sure that we change all student records.

## Insert Anomalies

- May not be possible to add a student unless they are enrolled in a course.

## Delete Anomalies

- If all the students enrolled in a course are deleted, then we lose the room number.

# Example Database

---

**student(sid,name,address)**

sid	name	address
123	Andy	Pittsburgh
456	Tupac	Los Angeles
789	Obama	Chicago
012	Waka Flocka	Atlanta

**courses(sid,id,grade)**

sid	cid	grade
123	15-415	A
456	15-721	B
789	15-415	A
012	15-415	C
789	15-721	A

**rooms(cid,room)**

cid	room
15-415	GHC6115
15-721	GHC8102

*Why this decomposition is better and how to find it.*

# Video on LearnIT

All about Functional Dependencies and how to derive them

# Functional Dependencies

---

A **functional dependency** (FD) is a form of a constraint.  
Part of a relation's schema to define a valid instance.

Definition:  $X \rightarrow Y$

→ The value of  $X$  functionally defines the value of  $Y$ .

# Functional Dependencies

---

Formal Definition:

$$\rightarrow X \rightarrow Y \Rightarrow (t_1[x] = t_2[x] \Rightarrow t_1[y] = t_2[y])$$

If two tuples  $(t_1, t_2)$  agree on the  $X$  attribute, then they must agree on the  $Y$  attribute too.

sid	name	address
123	Andy	Pittsburgh
456	Tupac	Los Angeles
789	Obama	Chicago
012	Waka Flocka	Atlanta

X      Y

✓  $\text{sid} \rightarrow \text{name}$

# Functional Dependencies

---

FD is a constraint that allows instances for which the FD holds.

You can check if an instance violates an FD, but you **cannot** prove that an FD is part of the schema using an instance.

R1(sid,name,address)

sid	name	address
123	Andy	Pittsburgh
456	Tupac	Los Angeles
789	Obama	Chicago
012	Waka Flocka	Atlanta
555	Andy	Providence

??? name → address

# Why Should you Care?

FDs seem important, but what can we do with them?

They allow us to decide whether a database design is correct.  
→ Note that this is different than whether it's a good idea for performance.

# Normal Forms

---

Identify “bad” functional dependencies

=>

Check for *normal forms*!

to ensure your relational schema design is free of certain issues  
functional dependencies are used to define normal forms

# Normal Forms

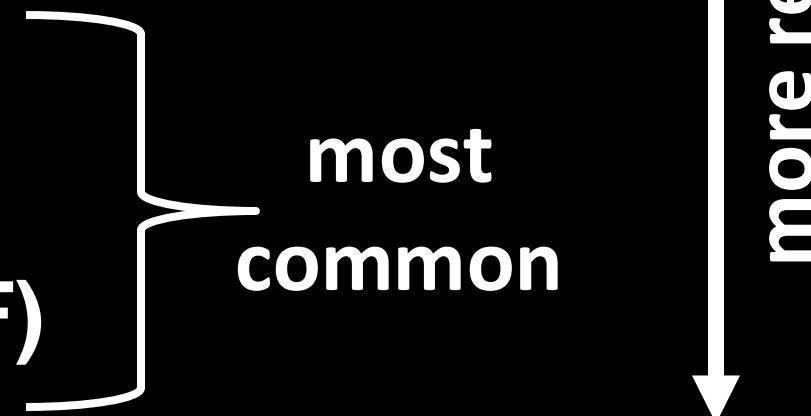
---

**first normal form (1NF)**

**second normal form (2NF)**

**third normal form (3NF)**

**Boyce-Codd normal form (BCNF)**

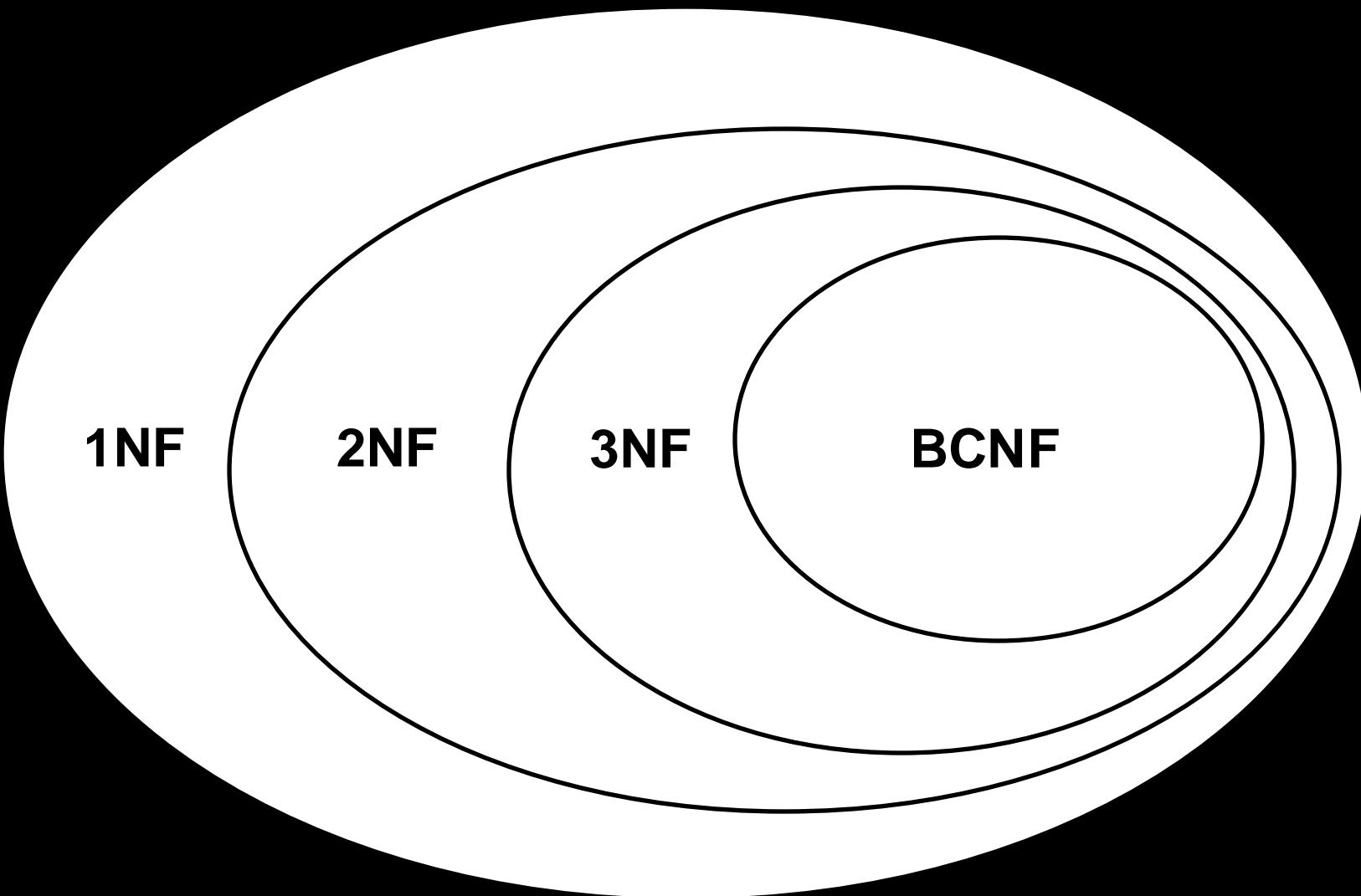


**there are more,**

**but you don't ever need to know them (i hope)**

# Normal Forms: Venn Diagram

---



all attributes are of atomic types  
no lists or sets as attributes

<b>id</b>	<b>name</b>	<b>email</b>	<b>office</b>	<b>age</b>	<b>position</b>
108	rory	{rory@itu.dk, lorelai@itu.dk}	414	35	assoc
118	marvin	marvin@itu.dk	556	35	assist
80	sabina	sabina@itu.dk	112	47	full
128	frodo	frodo@itu.dk	311	30	assoc

1NF &  
non-key attributes ***fully (not partially)*** depend on the  
whole candidate key(s)

if *manufacturer* → *country* & key: *manufacturer, model*



manufacturer	model	country	amount
a	v1	italy	500\$
b	v1	denmark	400\$
c	v1	germany	400\$
a	v2	italy	600\$

# 2NF

---

1NF &

non-key attributes *fully (not partially)* depend on the whole candidate key(s)

if *manufacturer* → *country*

manufacturer	country
a	italy
b	denmark
c	germany



key: *manufacturer*

manufacturer	model	amount
a	v1	500\$
b	v1	400\$
c	v1	400\$
a	v2	600\$

key: *manufacturer, model*

# 3NF

for all  $X \rightarrow A$  in  $F^+$  for a relation  $R$   
 $A \subseteq X$  (trivial functional dependency), or  
 $X$  contains key for  $R$ , or  
 $A$  is part of some key for  $R$ .

if  $position \rightarrow salary$  (neither attributes are keys)



	<b>id</b>	<b>name</b>	<b>office</b>	<b>age</b>	<b>position</b>	<b>salary</b>
108	rory	414	35	assoc	50000	
118	marvin	556	35	assist	40000	
80	sabina	112	47	full	70000	
128	frodo	311	30	assoc	50000	

# 3NF

---

for all  $X \rightarrow A$  in  $F^+$  for a relation  $R$   
 $A \subseteq X$  (trivial functional dependency), or  
 $X$  contains key for  $R$ , or  
 $A$  is part of some key for  $R$ .

*position → salary*

position	salary
assoc	50000
assist	40000
full	70000



id	name	office	age	position
108	rory	414	35	assoc
118	marvin	556	35	assist
80	sabina	112	47	full
128	frodo	311	30	assoc

# BCNF

---

for all  $X \rightarrow A$  in  $F^+$  for a relation  $R$   
 $A \subseteq X$  (trivial functional dependency), or  
 $X$  contains key for  $R$ .

$fID, progID \rightarrow start\_date$  (a faculty can start at a program at most once)

$start\_date \rightarrow progID$  (on a given day, at most one faculty can start at a program)

candidate keys:  $fID, progID$  or  $progID, start\_date$

**works\_in**

<b>fID</b>	<b>progID</b>	<b>start_date</b>
108	DS	01/01/2015
118	DS	02/02/2017
80	CS	03/03/2017
108	CS	02/03/2017

 **start\_date is not a key!**

**BCNF is ideal but  
can sometimes be  
too restrictive!**

# 3NF

---

for all  $X \rightarrow A$  in  $F^+$  for a relation R  
 $A \subseteq X$  (trivial functional dependency), or  
X contains key for R, or  
A is part of some key for R.

$fID, progID \rightarrow start\_date$  (a faculty can start at a program at most once)

$start\_date \rightarrow progID$  (on a given day, at most one faculty can start at a program)

candidate keys: fID,progID or progID,start\_date

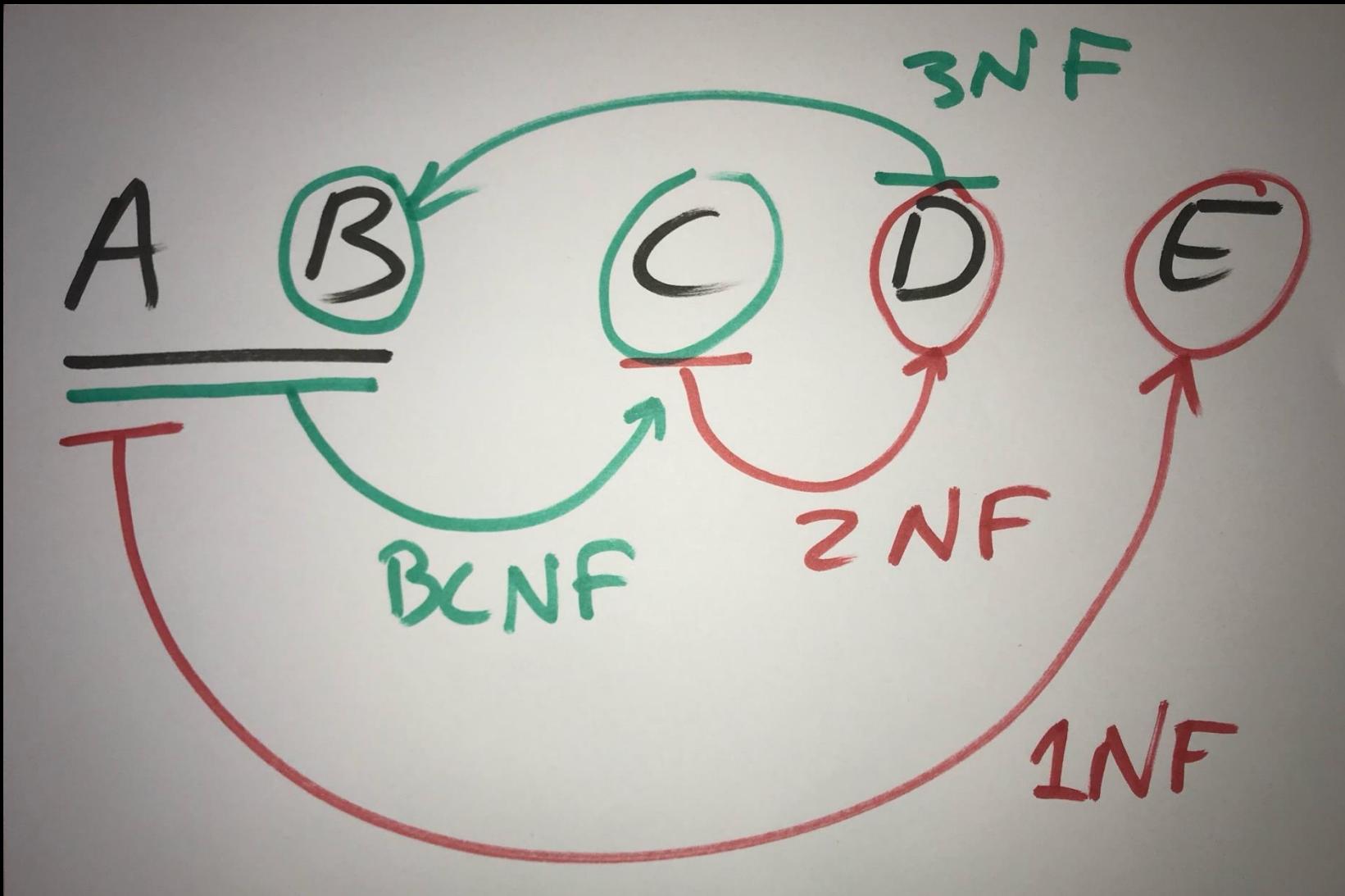
**works\_in**

fID	progID	start_date
108	DS	01/01/2015
118	DS	02/02/2017
80	CS	03/03/2017
108	CS	02/03/2017



**3NF allows this!**

# Visualizing FDs and Normal Forms



# Normalization

---

Look for ***functional dependencies!*** (video on LearnIT!)

Identify “bad” functional dependencies that break BCNF/3NF

If there are such, then decompose the tables into two tables

Check again and repeat for sub-tables

When done, the database schema is normalized

# Simple Decomposition Algorithm

---

1. Find all FDs
2. While FD < 3NF exists:  
    Decompose!

Will result in BCNF or 3NF  
No way from 3NF to BCNF ...  
... except by losing a dependency  
... so we don't decompose 3NF!!!

# Practical Decomposition

---

**Consider relation R and (important) functional dependency  
 $X \rightarrow Y$  that violates 3NF/BCNF**

Decompose R into R1 and R2 where

R1 = R – Y (everything but Y = the right side)

R2 = XY (the whole FD = both left and right side)

**This has the following nice properties**

R2 is (normally) in BCNF

Joining R1 and R2 (with = on all X attributes) yields R

**Example: Person(ID, Name, ZIP, City), ZIP  $\rightarrow$  City**

X = ZIP, Y = City

R-Y = Person(ID, Name, ZIP)

XY = ZIP(ZIP, City)

# Example Decomposition I

$R = \underline{ABCD}$

$AB \rightarrow CD$

$C \rightarrow D$

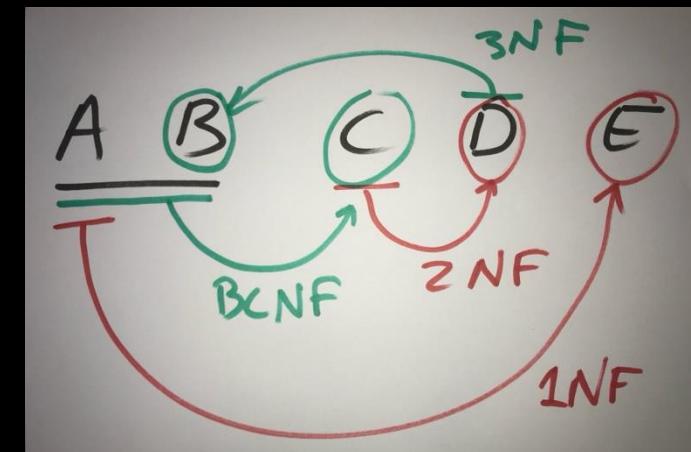
New Table: CD

Old Table: ABC

Both in BCNF

What happened to  $AB \rightarrow D$ ?

$X \rightarrow Y$   
 $R1 = R - Y$   
 $R2 = XY$



# Example Decomposition II

$R = \underline{AB}CD$  ( $AB \rightarrow CD$ )

$A \rightarrow D$

$A = \text{personID}$ ,  $B = \text{projectID}$ ,  
 $C = \text{start date}$ ,  $D = \text{name of person}$

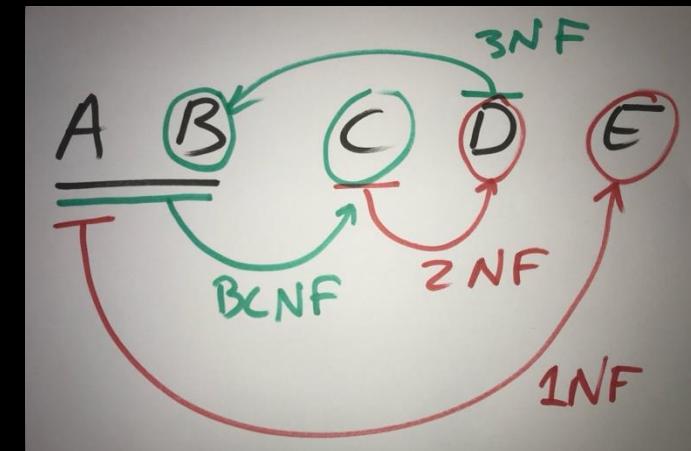
New Table: AD

Old Table: ABC

Both in BCNF

What happened to  $AB \rightarrow D$ ?

$X \rightarrow Y$   
 $R1 = R - Y$   
 $R2 = XY$



# Example Decomposition III

$R = \underline{ABCD}$

$A \rightarrow C$

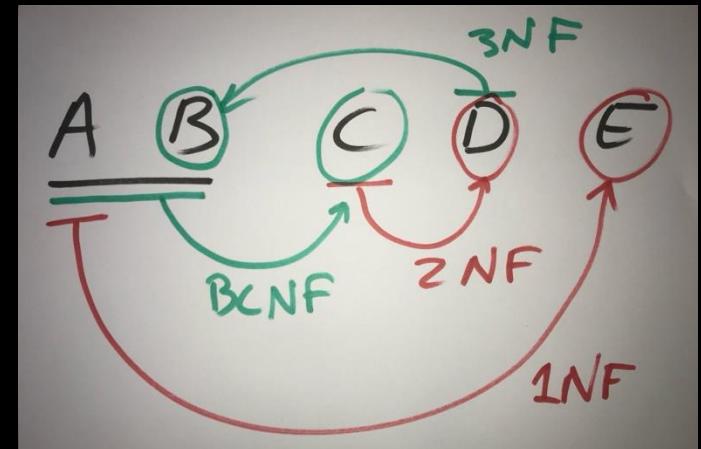
$A \rightarrow D$

New Table:  $\underline{ACD}$  // Merge tables with identical keys

Old Table:  $\underline{AB}$

Both in BCNF

$X \rightarrow Y$   
 $R1 = R - Y$   
 $R2 = XY$



# Example Decomposition IV

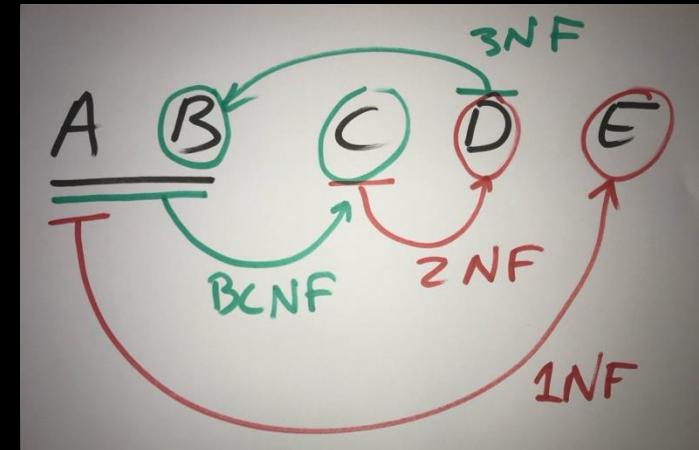
$R = \underline{ABCD}$

$D \rightarrow A$

$X \rightarrow Y$   
 $R1 = R - Y$   
 $R2 = XY$

$R$  is in 3NF

No decomposition into BCNF



# Example Decomposition V

$R = \underline{ABCD}$

$A \rightarrow C, C \rightarrow D$

Which one do we use first? ... and why?

New Table: CD

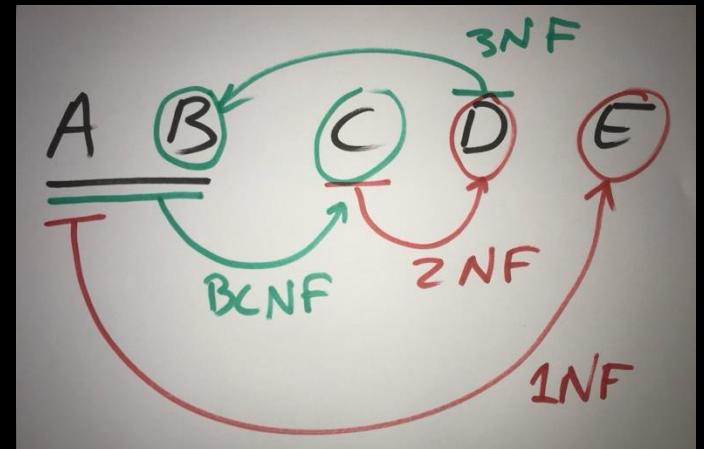
New Table: AC

Old Table: AB

All three in BCNF

What happened to  $AB \rightarrow D$

$X \rightarrow Y$   
 $R1 = R - Y$   
 $R2 = XY$



# Example Normalization

**Advises(Teacher, Student, Dept, Program, Course)**

Teacher → Dept

Course → Program

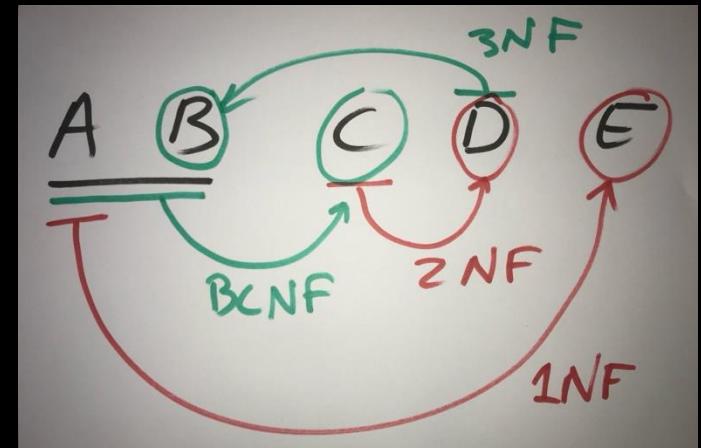
$X \rightarrow Y$   
 $R1 = R - Y$   
 $R2 = XY$

**Outcome:**

Faculty(Teacher, Dept)

Catalog(Course, Program)

Advises(Teacher, Student, Dept, Program, Course)



# The Inconvenient Truth

—  
or why many people don't like **normalization**

most NoSQL systems denormalize instead of decomposing a relation into two!

<b>id</b>	<b>name</b>	<b>office</b>	<b>age</b>	<b>position</b>	<b>salary</b>
108	rory	414	35	assoc	50000
118	marvin	556	35	assist	40000
80	sabina	112	47	full	70000
128	frodo	311	30	assoc	50000

**joining two tables (especially in a distributed environment) is expensive!!**

# Takeaways

- **Schema Refinement**
- **Functional Dependencies**
  - Unavoidable, trivial, and redundant FDs
- **Normal forms are important**
  - they form the foundation for schema refinement
  - always decompose 1NF and 2NF
  - but keep in mind you might have to take more things into account
- **Normalization Process**
  - decomposition of relations to BCNF (or 3NF)

# Introduction to Database Systems

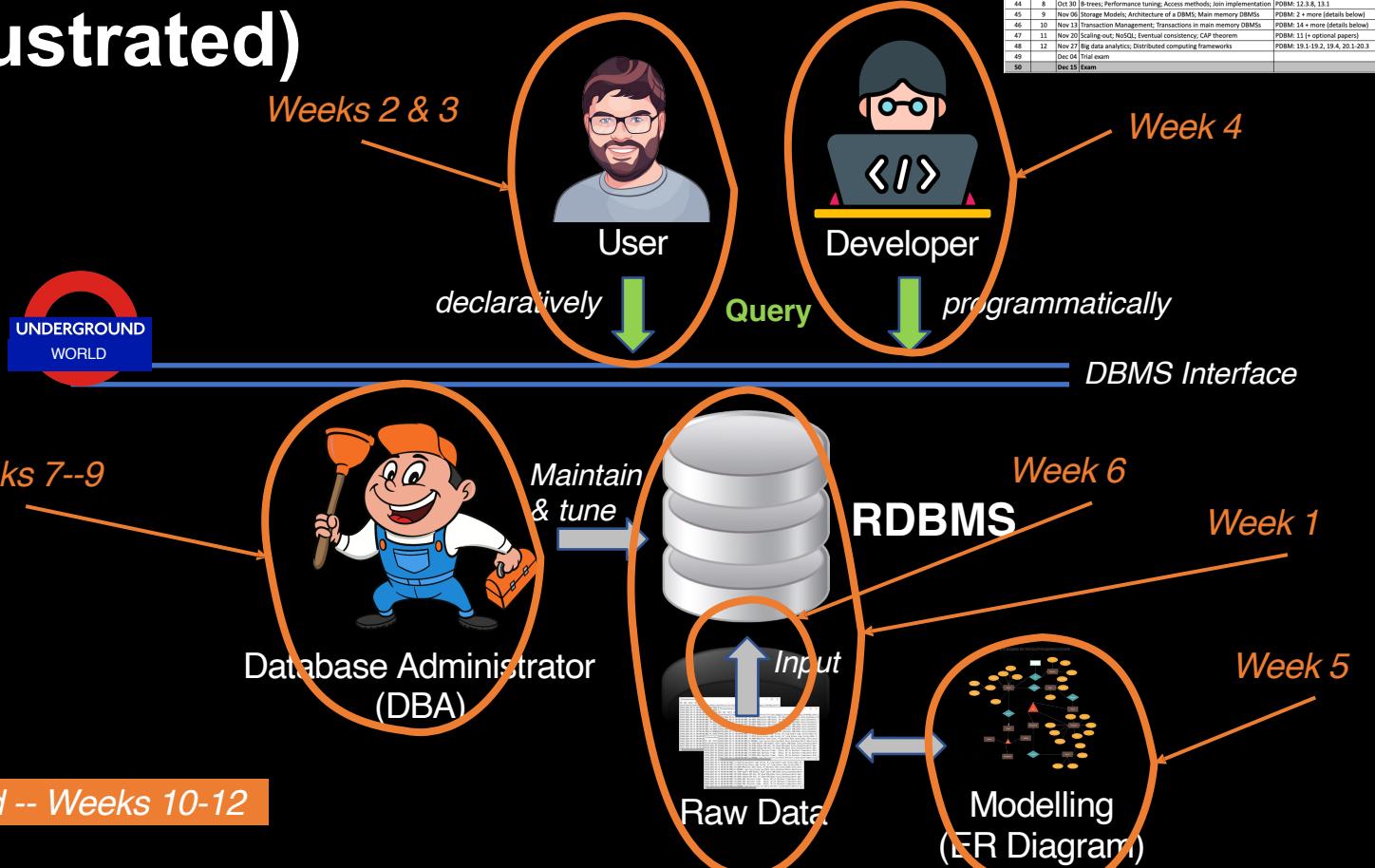
## IDBS – Spring 2024

# Functional Dependencies

**Readings**  
PDBM 6.2

**Eleni Tzirita Zacharatou**  
Based on slides by Andy Pavlo

# Course Schedule (Illustrated)



# **DATABASE DESIGN**

How do we design a “good” database schema?

We want to ensure the integrity of the data.

We also want to get good performance.

# EXAMPLE DATABASE

**student(sid,cid,room,grade,name,address)**

sid	cid	room	grade	name	address
123	15-445	GHC 6115	A	Andy	Pittsburgh
456	15-721	GHC 8102	B	Tupac	Los Angeles
789	15-445	GHC 6115	A	Obama	Chicago
012	15-445	GHC 6115	C	Waka Flocka	Atlanta
789	15-721	GHC 8102	A	Obama	Chicago

# EXAMPLE DATABASE

**student(sid,cid,room,grade,name,address)**

<b>sid</b>	<b>cid</b>	<b>room</b>	<b>grade</b>	<b>name</b>	<b>address</b>
123	15-445	GHC 6115	A	Andy	Pittsburgh
456	15-721	GHC 8102	B	Tupac	Los Angeles
789	15-445	GHC 6115	A	Obama	Chicago
012	15-445	GHC 6115	C	Waka Flocka	Atlanta
789	15-721	GHC 8102	A	Obama	Chicago

# **REDUNDANCY PROBLEMS**

## **Update Anomalies**

- If the room number changes, we need to make sure that we change all students records.

## **Insert Anomalies**

- May not be possible to add a student unless they're enrolled in a course.

## **Delete Anomalies**

- If all the students enrolled in a course are deleted, then we lose the room number.

# EXAMPLE DATABASE

**student(sid,name,address)**

sid	name	address
123	Andy	Pittsburgh
456	Tupac	Los Angeles
789	Obama	Chicago
012	Waka Flocka	Atlanta

**rooms(cid,room)**

cid	room
15-415	GHC 6115
15-721	GHC 8102

**courses(sid,cid,grade)**

sid	cid	grade
123	15-415	A
456	15-721	B
789	15-415	A
012	15-415	C
789	15-721	A

*Why this decomposition is better and how to find it.*

# THIS VIDEO

Functional Dependencies: Constraints between attributes

# FUNCTIONAL DEPENDENCIES

A functional dependency (FD) is a form of a constraint.

Part of a relation's schema to define a valid instance.

Definition:  $X \rightarrow Y$

→ The value of  $X$  functionally defines the value of  $Y$ .

# FUNCTIONAL DEPENDENCIES

Formal Definition:

$$\rightarrow X \rightarrow Y \Rightarrow (t_1[x] = t_2[x] \Rightarrow t_1[y] = t_2[y])$$

If two tuples ( $t_1, t_2$ ) agree on the  $X$  attribute, then they must agree on the  $Y$  attribute too.

R1(sid, name, address)

sid	name	address
123	Andy	Pittsburgh
456	Tupac	Los Angeles
789	Obama	Chicago
012	Waka Flocka	Atlanta

# FUNCTIONAL DEPENDENCIES

Formal Definition:

$$\rightarrow X \rightarrow Y \Rightarrow (t_1[x] = t_2[x] \Rightarrow t_1[y] = t_2[y])$$

If two tuples ( $t_1, t_2$ ) agree on the  $X$  attribute, then they must agree on the  $Y$  attribute too.

R1(sid, name, address)

sid	name	address
123	Andy	Pittsburgh
456	Tupac	Los Angeles
789	Obama	Chicago
012	Waka Flocka	Atlanta

X      Y

✓ sid  $\rightarrow$  name

# FUNCTIONAL DEPENDENCIES

FD is a constraint that allows instances for which the FD holds.

You can check if an FD is violated by an instance, but you cannot prove that an FD is part of the schema using an instance.

R1(sid, name, address)

sid	name	address
123	Andy	Pittsburgh
456	Tupac	Los Angeles
789	Obama	Chicago
012	Waka Flocka	Atlanta
555	Andy	Providence

???  $\text{name} \rightarrow \text{address}$

# FUNCTIONAL DEPENDENCIES

Two FDs  $X \rightarrow Y$  and  $X \rightarrow Z$  can be written in shorthand as  $X \rightarrowYZ$ .

But  $XY \rightarrow Z$  is not the same as the two FDs  $X \rightarrow Z$  and  $Y \rightarrow Z$ .

# WHY SHOULD I CARE?

FDs seem important, but what can we actually do with them?

They allow us to decide whether a database design is correct.

→ Note that this different then the question of whether it's a good idea for performance...

# IMPLIED DEPENDENCIES

**student(sid,cid,room,grade,name,address)**

sid	cid	room	grade	name	address
123	15-445	GHC 6115	A	Andy	Pittsburgh
456	15-721	GHC 8102	B	Tupac	Los Angeles
789	15-445	GHC 6115	A	Obama	Chicago
012	15-445	GHC 6115	A	Waka Flocka	Atlanta

Provided FDs

$\text{sid} \rightarrow \text{name, address}$   
 $\text{sid, cid} \rightarrow \text{grade}$

Implied FDs

$\text{sid, cid} \rightarrow \text{grade}$   
 $\text{sid, cid} \rightarrow \text{sid}$   
 $\text{sid, cid} \rightarrow \text{cid}$

# IMPLIED DEPENDENCIES

Given a set of FDs  $\{f_1, \dots, f_n\}$ , how  
do we decide whether FD  $g$  holds?

Compute the closure using  
Armstrong's Axioms  
→ This is the set of all implied FDs.

# ARMSTRONG'S AXIOMS

**Reflexivity:**

$$\rightarrow X \supseteq Y \Rightarrow X \rightarrow Y$$

- This is a **trivial functional dependency**

**Augmentation:**

$$\rightarrow X \rightarrow Y \Rightarrow XZ \rightarrow YZ$$

**Transitivity:**

$$\rightarrow (X \rightarrow Y) \wedge (Y \rightarrow Z) \Rightarrow X \rightarrow Z$$

**Union:**

$$\rightarrow (X \rightarrow Y) \wedge (X \rightarrow Z) \Rightarrow X \rightarrow YZ$$

**Decomposition:**

$$\rightarrow X \rightarrow YZ \Rightarrow (X \rightarrow Y) \wedge (X \rightarrow Z)$$

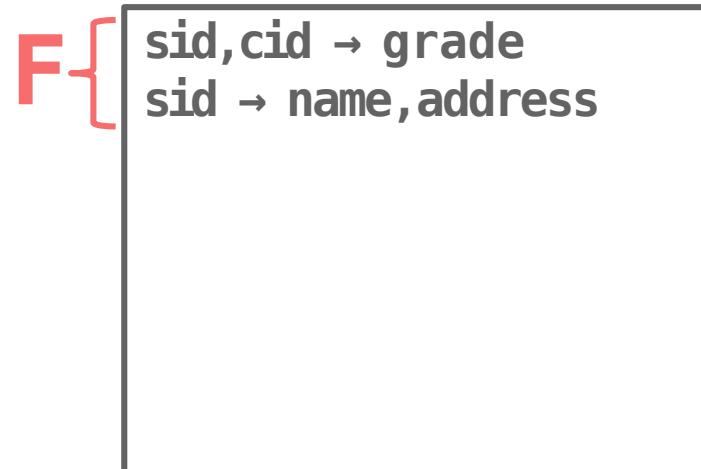
**Pseudo-transitivity:**

$$\rightarrow (X \rightarrow Y) \wedge (Y \leftarrow Z) \Rightarrow X \leftarrow Z$$

# CLOSURES

Given a set  $F$  of FDs  $\{f_1, \dots, f_n\}$ , we define the closure  $F^+$  is the set of all implied FDs.

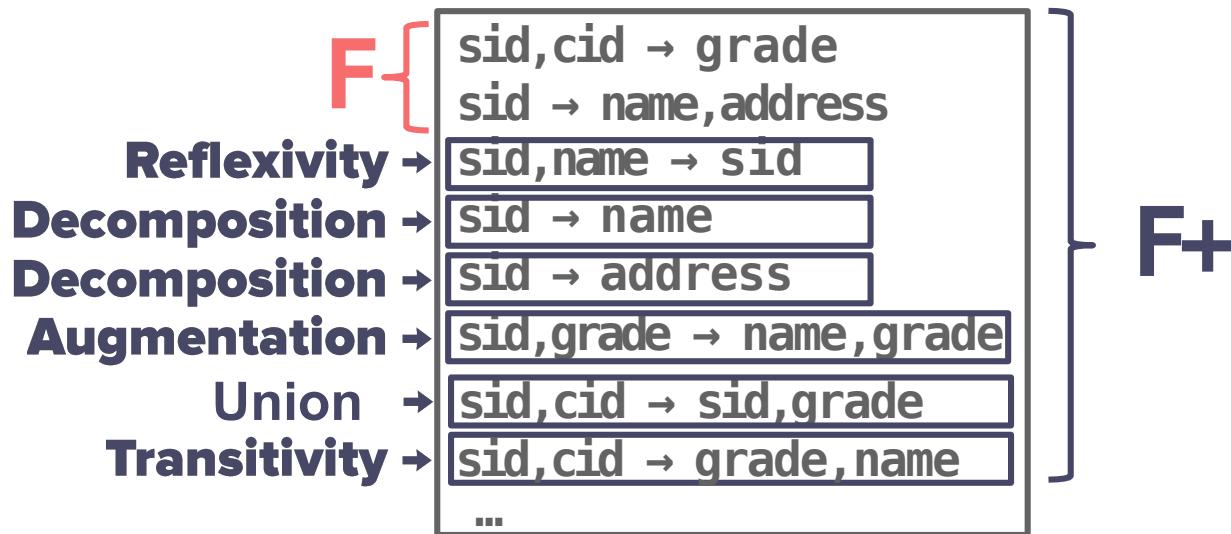
`student(sid, cid, room, grade, name, address)`



# CLOSURES

Given a set  $F$  of FDs  $\{f_1, \dots, f_n\}$ , we define the closure  $F^+$  is the set of all implied FDs.

`student(sid, cid, room, grade, name, address)`



# UNAVOIDABLE AND REDUNDANT FDs

- FDs with a superkey on the left-hand side are **unavoidable**
  - If A is a candidate key for a relation, then clearly  $A \rightarrow B$  for any attribute B
  - Similarly, if  $\{A_1, A_2\}$  forms a superkey, we have  $A_1 A_2 \rightarrow B$  for any B
  - Etc
- FDs that can be computed from others are **redundant**
  - They do not require decomposition!
- Only decompose when not trivial, unavoidable, or redundant

# **CONCLUSION**

Functional dependencies are simple to understand.

They will allow us to reason about schema decompositions.

➤ **This week's lecture**

# Introduction to Database Systems

## IDBS – Fall 2024

- Week 7:
- Storage Hierarchy
- Physical Database Design
- Indexing

---

Eleni Tzirita Zacharatou

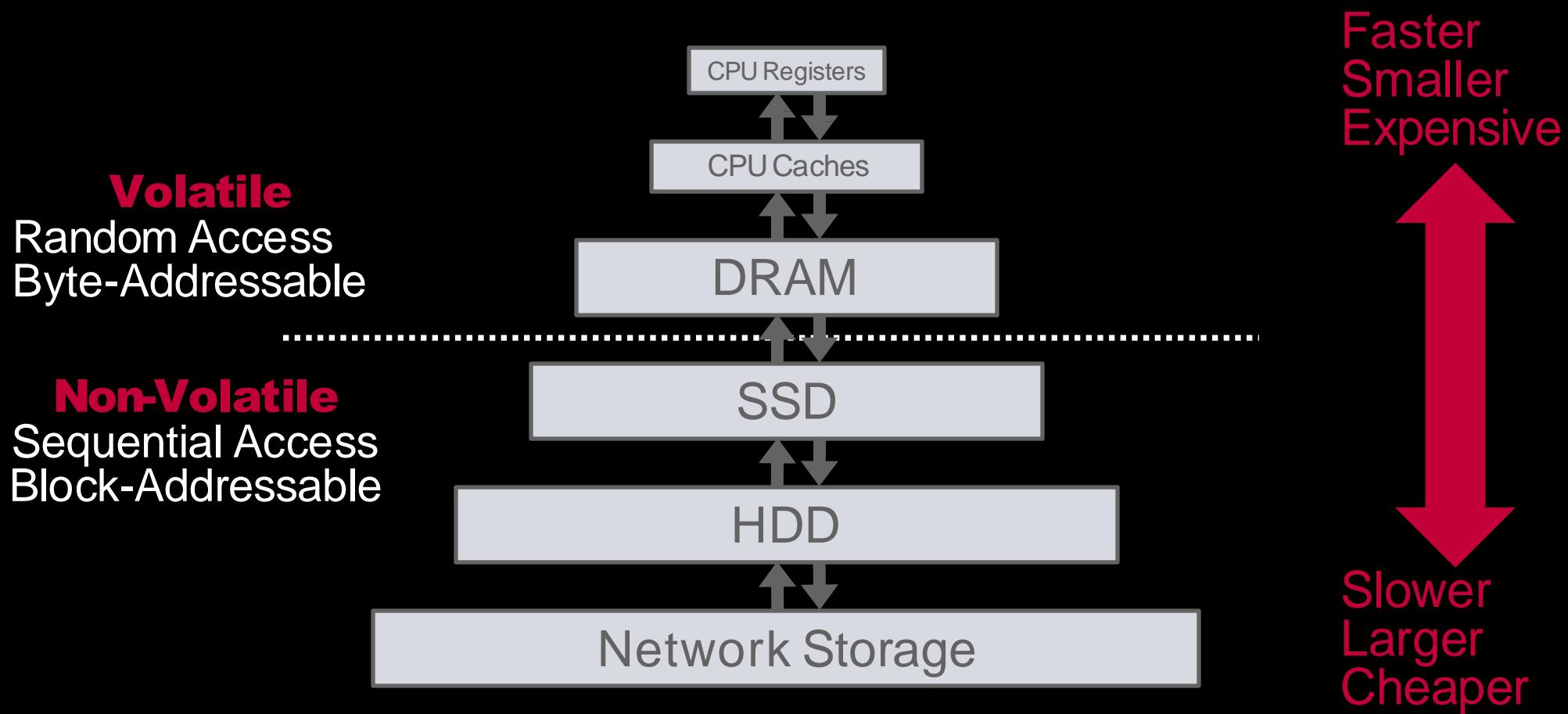
**Readings:**

PDBM 12.1, 12.2, 12.3.1 - 12.3.7

## Storage Hierarchy

Physical Database  
Design

# Storage Hierarchy



# Access Times

---

<b>1ns</b>	L1 Cache Ref	←	<b>1 sec</b>
<b>4 ns</b>	L2 Cache Ref	←	<b>4 sec</b>
<b>100 ns</b>	DRAM	←	<b>100 sec</b>
<b>16,000 ns</b>	SSD	←	<b>4.4 hours</b>
<b>2,000,000 ns</b>	HDD	←	<b>3.3 weeks</b>
<b>~50,000,000 ns</b>	Network Storage	←	<b>1.5 years</b>
<b>1,000,000,000 ns</b>	Tape Archives	←	<b>31.7 years</b>

Source: [Colin Scott](#)

# Disk-Based Architecture

---

- The DBMS assumes that the primary storage location of the database is on non-volatile disk.
- The DBMS's components manage the movement of data between non-volatile and volatile storage.

# Sequential vs. Random Access

- Random access on non-volatile storage is almost always much slower than sequential access.
- DBMS will want to maximize sequential access.
  - Algorithms try to reduce number of writes to random pages so that data is stored in contiguous blocks.

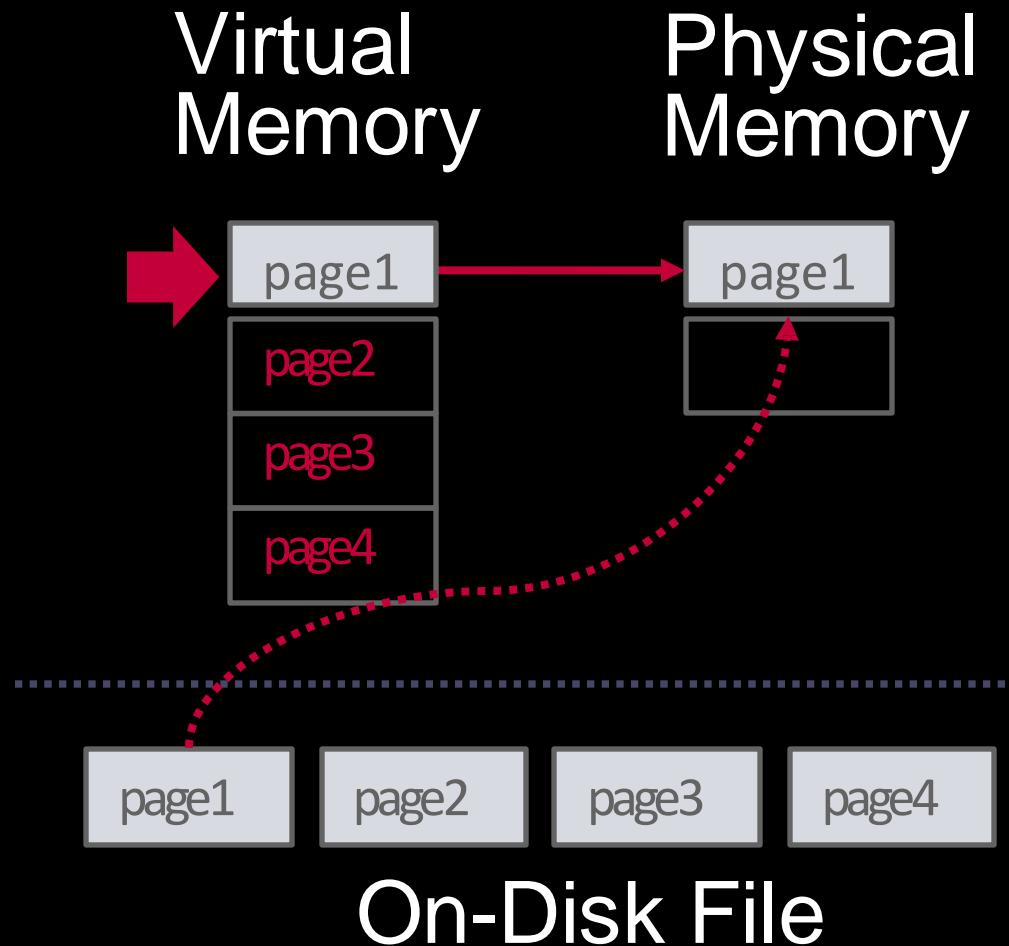
# System Design Goals

---

- Allow the DBMS to manage databases that exceed the amount of memory available.
- Reading/writing to disk is expensive, so it must be managed carefully to avoid large stalls and performance degradation.
- Random access on disk is usually much slower than sequential access, so the DBMS will want to maximize sequential access.

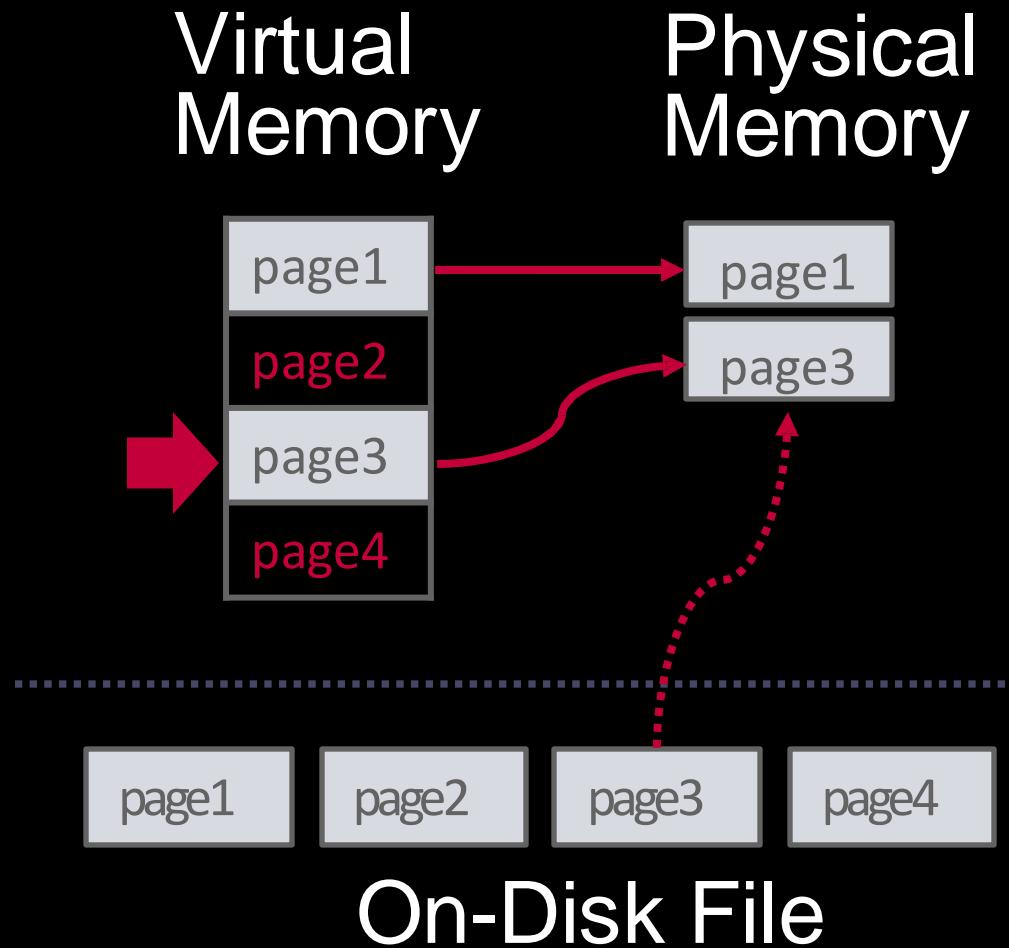
# Why Not Use the OS?

- The DBMS can use memory mapping (*mmap*) to store the contents of a file into the address space of a program.
- OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it.



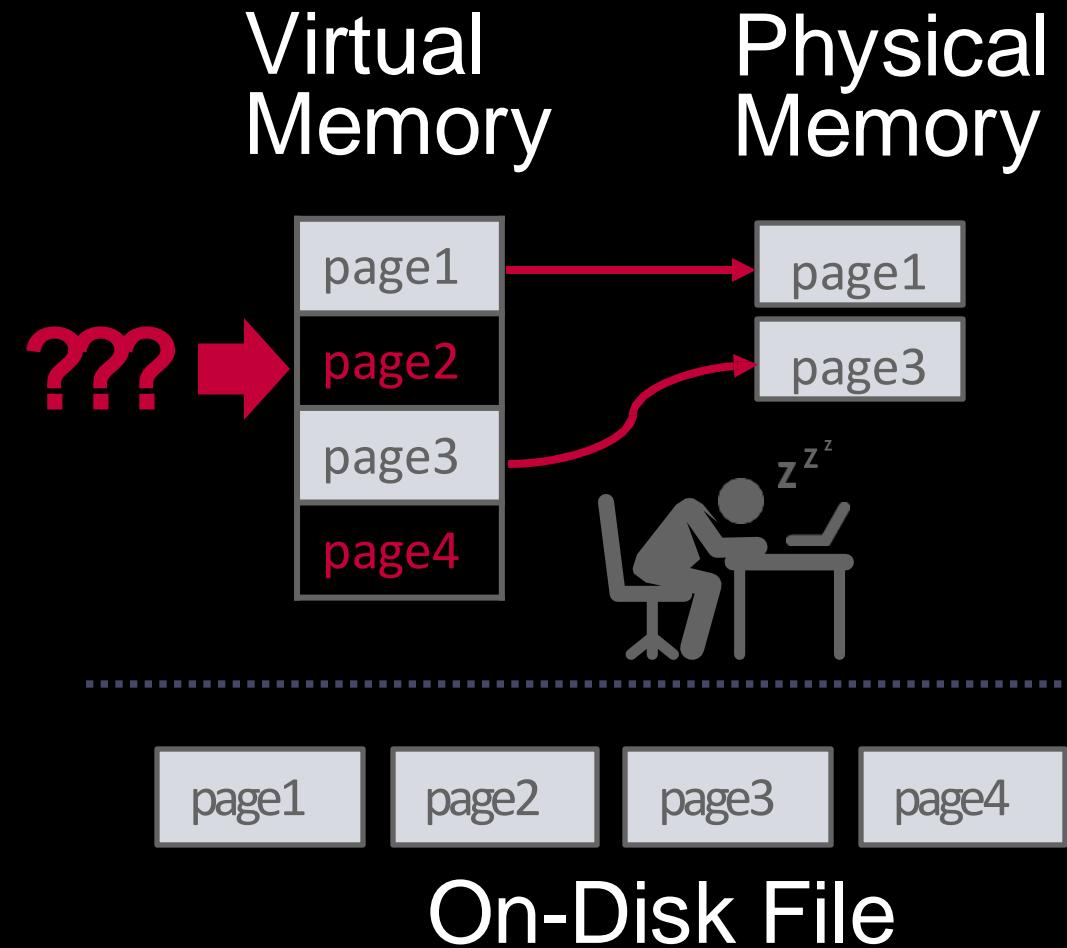
# Why Not Use the OS?

- The DBMS can use memory mapping (*mmap*) to store the contents of a file into the address space of a program.
- OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it.



# Why Not Use the OS?

- The DBMS can use memory mapping (*mmap*) to store the contents of a file into the address space of a program.
- OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it.



# Memory Mapped I/O Problems

## **Problem #1: Transaction Safety**

→ OS can flush dirty pages at any time.

## **Problem #2: I/O Stalls**

→ DBMS doesn't know which pages are in memory. The OS will stall a thread on page fault.

## **Problem #3: Error Handling**

→ Difficult to validate pages. Any access can cause a **SIGBUS** that the DBMS must handle.

## **Problem #4: Performance Issues**

→ OS data structure contention. TLB shootdowns.

# Why Not Use the OS?

- DBMS (almost) always wants to control things itself and can do a better job than the OS.
  - Flushing dirty pages to disk in the correct order.
  - Specialized prefetching.
  - Buffer replacement policy.
  - Thread/process scheduling.
- The OS is *not* your friend.

# Storage Manager

- The DBMS stores a database as one or more files on disk typically in a proprietary format.
- The storage manager is responsible for maintaining the files of a database.
- It organizes the files as a collection of *pages*.
  - Tracks data read/written to pages.
  - Tracks the available space.

# Database Pages

---

- A page is a fixed-size block of data.
  - It can contain tuples, meta-data, indexes, log records...
  - Most systems do not mix page types.
  - Some systems require a page to be self-contained.
- Each page is given a unique identifier.
  - The DBMS uses an indirection layer to map page IDs to physical locations.

# Indexing

---

# Full Table Scans

---

When a DBMS sees a query of the form

```
SELECT *
FROM R
WHERE <condition>
```

one obviously possible thing to do is read through **all** the tuples of R and report those tuples that satisfy the condition.

This is called a **full table scan**.

# Selective Queries

---

Consider the query from before:

- If we have to report 80% of the tuples in R, it makes sense to do a full table scan.
- On the other hand, if the query is very **selective**, and returns just a small percentage of the tuples, we might hope to do better.

```
SELECT *
FROM  R
WHERE <condition>
```

# Point Queries

---

- Consider a selection query with a single equality condition:

```
SELECT *
  FROM person
 WHERE birthdate='1975-02-06' ;
```

- This is a *point query*: We look for a single value of birthdate.
  - We may still return > 1 record!
- Point queries are easy if data is sorted by the attribute used in the condition.
  - How? What algorithms would work?

# Range Queries

- Consider a selection query of the form:

```
SELECT *
FROM person
WHERE birthdate BETWEEN
'1975-02-01' and '1975-02-28';
```

- This is a *range query*: We look for a range of values of birthdate.
- Range queries are also easy if data is sorted by the right attribute.
  - But often not as **selective** as point queries.

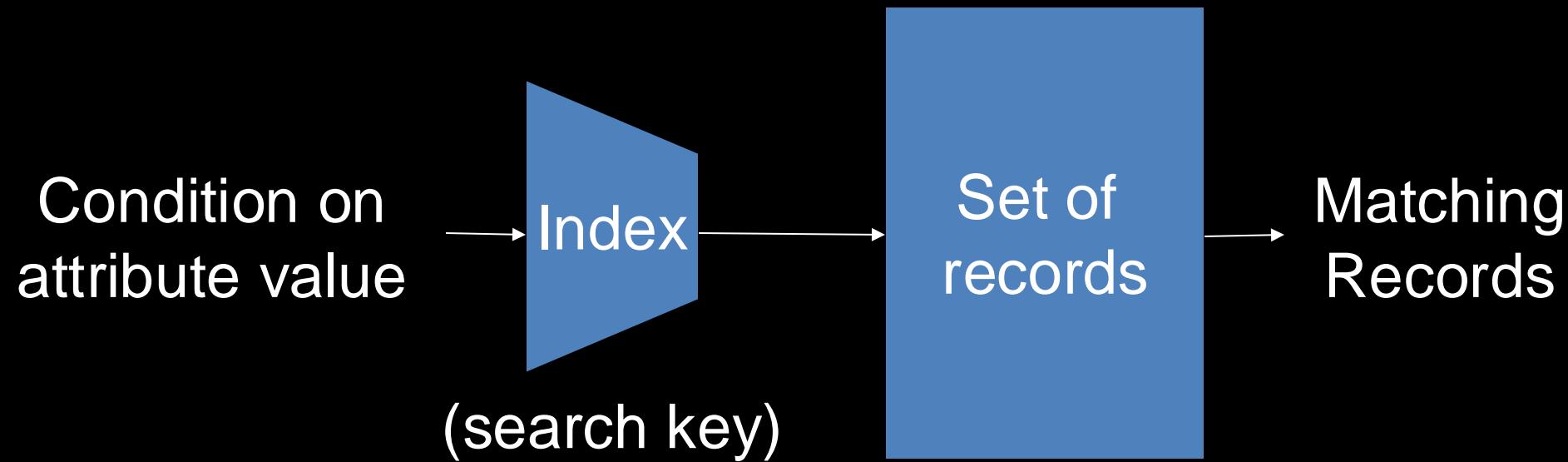
# Indexes

---

- To speed up queries the DBMS may build an index on the birthdate **attribute**.
- A database index is similar to an index in the back of a book:
  - For every piece of data you might be interested in (e.g., the attribute value 1975-02-06), the index says where to find the row with the actual data!
  - The index itself is organized such that one can quickly do the lookup.
- Looking for information in a relation with the help of an index is called an **index scan** (range) or **index lookup** (point).

# Indexing

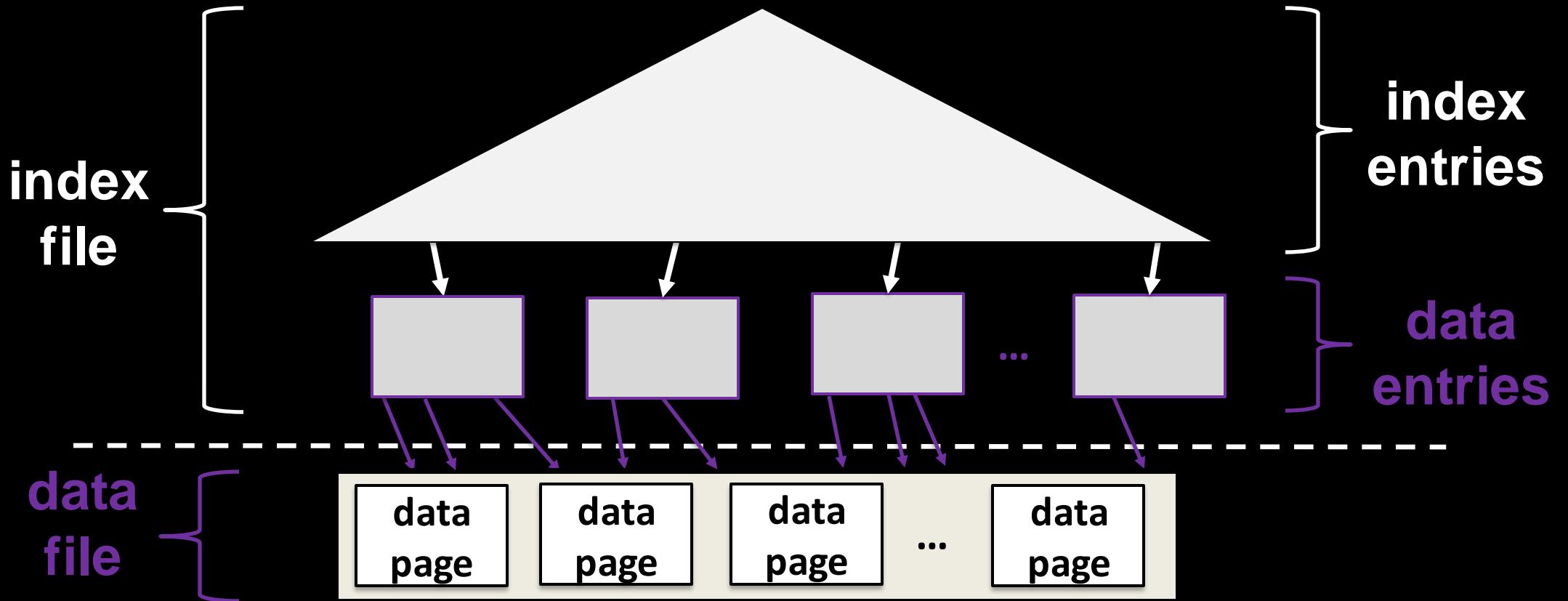
- An index is a data structure that supports efficient access to data
  - In databases, indexes are also stored on disk



# Two Techniques

- **Hashing:** Use a fixed transformation algorithm to convert the attribute value into a database address.
- **Tree search:** A dynamic search structure is built that guides the search for a given attribute value to the proper database location.
- Hashing supports equality queries only
  - Typically used dynamically for large-scale joins
  - Rarely available to developers
- Tree search is more versatile and accessible

# Clustered Index

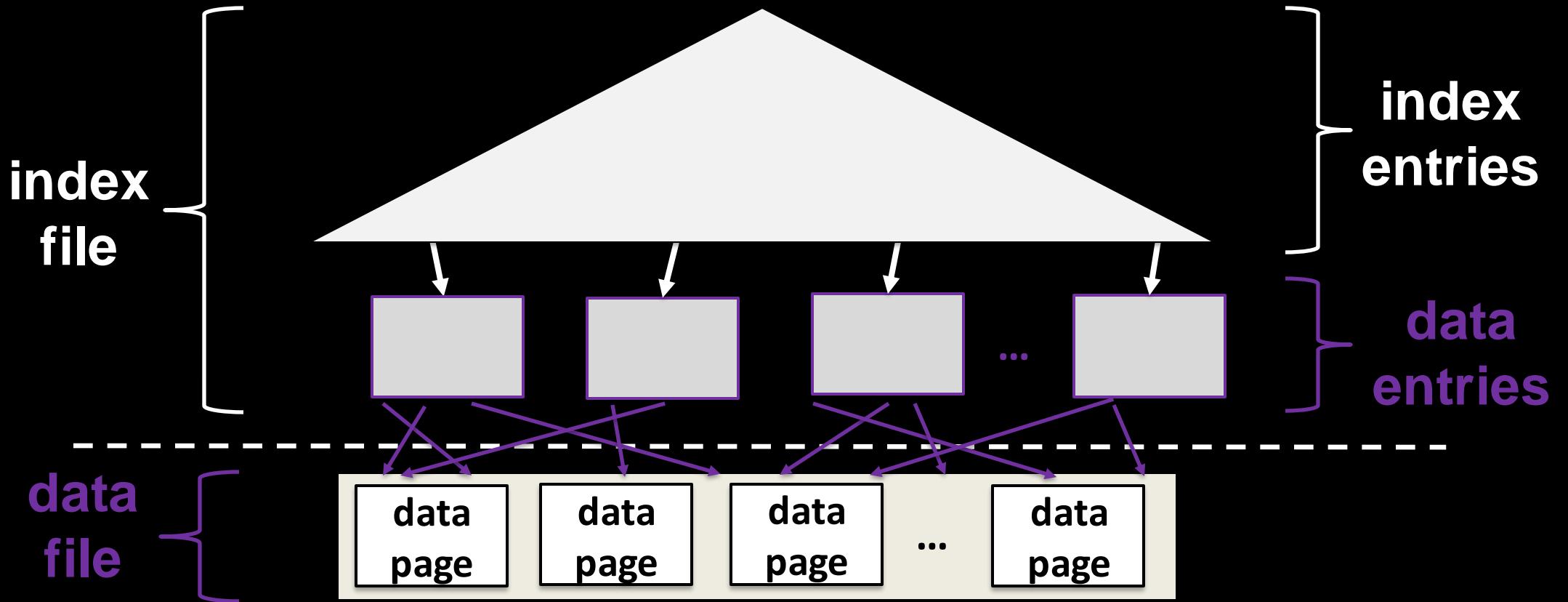


order of data entries matches order of data records

# Clustered Index

- If the tuples of a relation are stored sorted according to some attribute, an index on this attribute is called clustered.
  - Clustered indexes make point and range queries on the key *very efficient*
  - Why? Sequential reads + As few reads as possible!
- Many DBMSs automatically build a clustered index on the primary key of each relation.
- A clustered index is sometimes referred to as a primary or sparse index.
  - There can only be one clustered index! Why?

# Unclustered Index



order of data entries != order of data records

# Unclustered Index

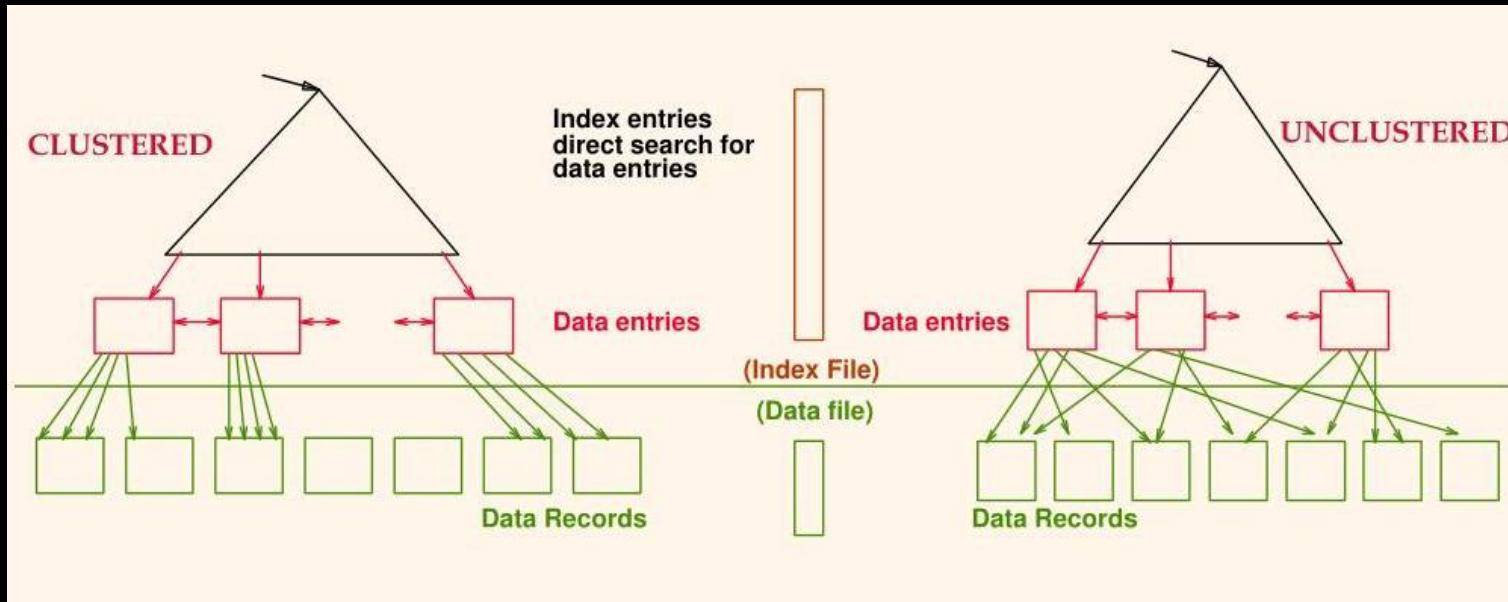
---

- It is possible to create further indexes on a relation. Typical syntax:

```
CREATE INDEX myIndex ON involved(actorId);
```

- The unclustered indexes are sometimes called non-clustered, secondary or dense indexes.
- Unclustered indexes:
  - Make most point queries more efficient.
  - Make some (narrow) range queries more efficient.

# Clustered vs. Unclustered Index

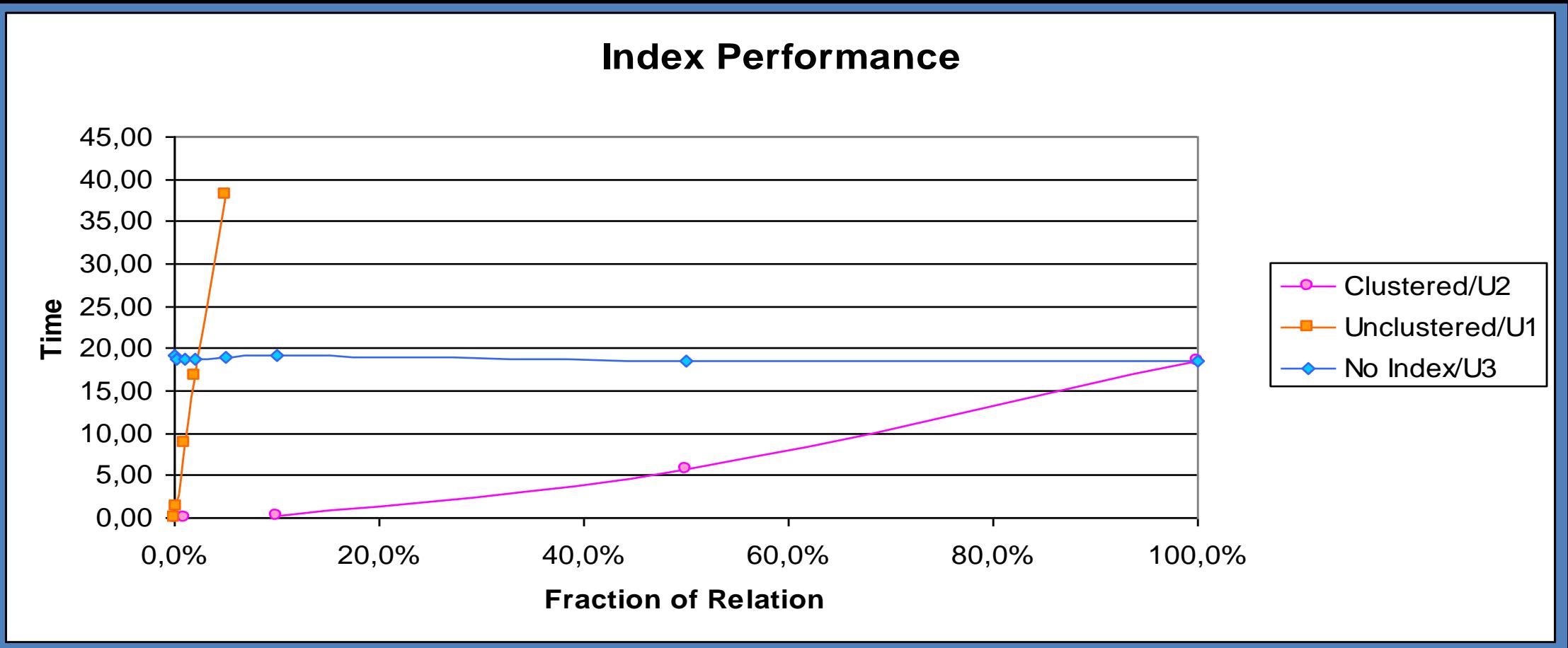


- To retrieve  $M$  records, where  $M$  is small:
  - Clustered: Probably one disk read
  - Unclustered: Probably  $M$  random disk reads
- To retrieve  $M$  records, where  $M$  is large:
  - Clustered: Probably  $M/\text{records\_per\_page}$  sequential disk reads
  - Unclustered: Up to  $M$  random disk reads
- We still need to read the index itself – same for both!

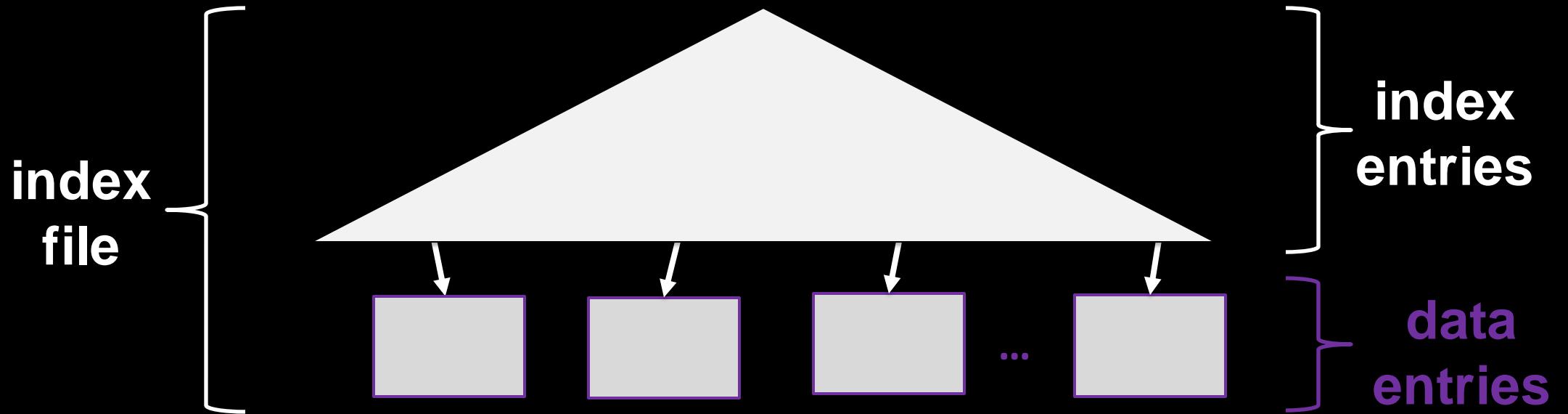
# Index Scan vs. Full Table Scan

- Point and range queries on the attribute(s) of the clustered index are almost always best performed using an index scan.
- Non-clustered indexes should only be used with high selectivity queries.
- Rule of thumb?

# Impact of Clustering on Performance



# Covering Index



**data entries contain ALL queried attributes**

# Covering Index

---

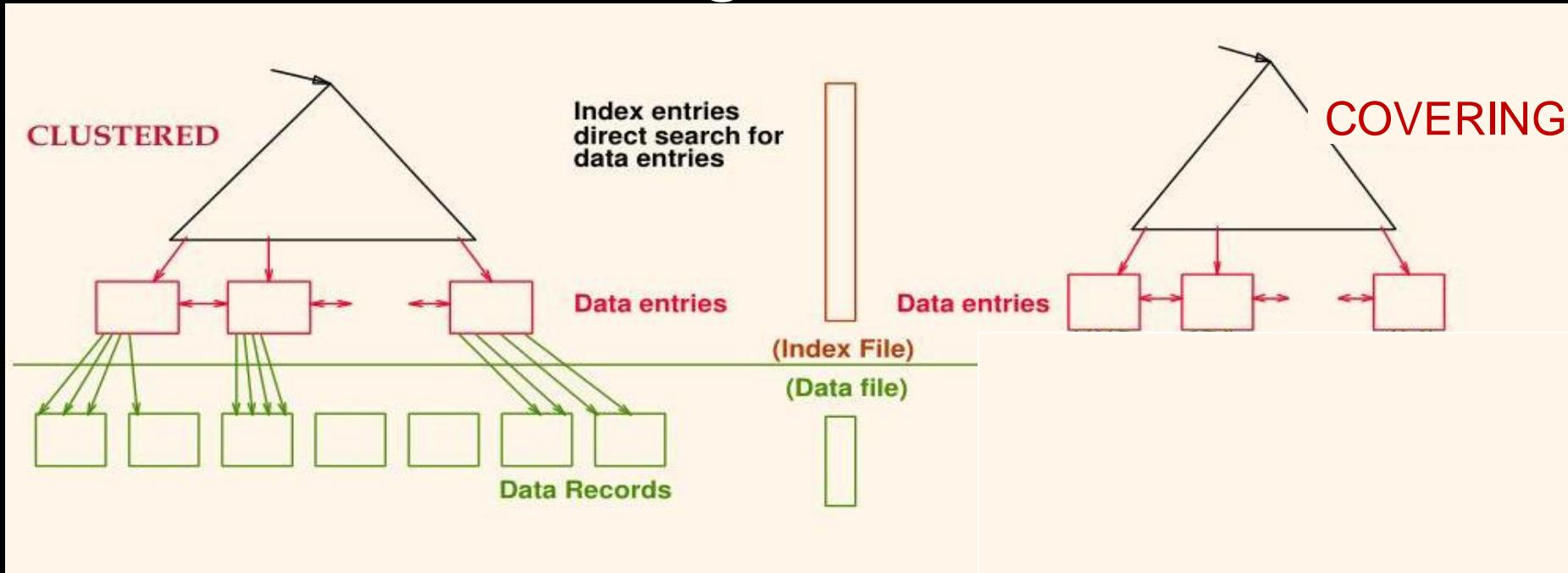
- An index that contains ALL attributes used in a query is called covering
  - The resulting query plans are index-only

```
SELECT COUNT(*) FROM movie WHERE year=1948;  
CREATE INDEX movieyear ON movie(year);
```

```
SELECT name FROM person WHERE height=170;  
CREATE INDEX phn ON person(height, name);
```

- The data from the relation is not needed = no disk reads required to retrieve tuples
  - Should a covering index be clustered or non-clustered?

# Clustered vs. Covering Index



- To retrieve  $M$  records, where  $M$  is small:
  - Clustered: Probably one disk read
  - Covering: Definitely need 0 disk reads
- To retrieve  $M$  records, where  $M$  is large:
  - Clustered: Probably  $M/\text{records\_per\_page}$  sequential disk reads
  - Covering: Definitely need 0 disk reads
- We still need to read the index itself – same for both!

# Problem Session

---

- For each of the following queries:
  - Which index would give the best plan?
  - Would the index be covering?
  - Would you prefer clustered or unclustered index?
  - Based on these queries, if you could choose, which index should be clustered?

1. SELECT \* ... WHERE birthdate = '20-02-2002'

2. SELECT \* ... WHERE height < 170

CREATE INDEX bIndex

3. SELECT \* ... WHERE ID = 4564

ON person(birthdate);

4. SELECT AVG(birthdate) ...

CREATE INDEX hIndex

ON person(height);

CREATE INDEX iIndex

ON person(ID);

# Multi-Attribute Indexes

---

- Defining an index on several attributes:

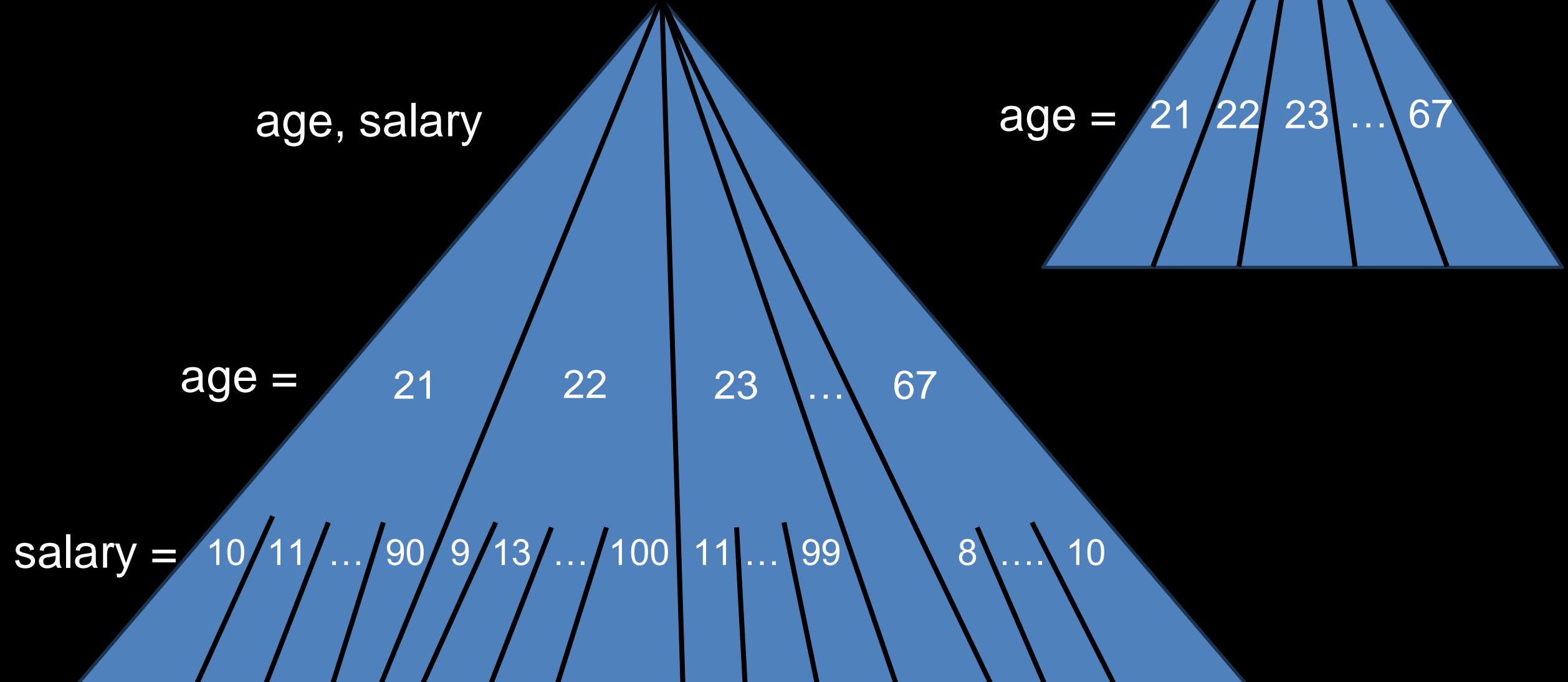
```
CREATE INDEX myIndex  
ON person (height,birthdate);
```

- Speeds up point/range queries such as:

```
SELECT *  
FROM person  
WHERE height=170 and birthdate<'1945-08-08'
```

- An index on several attributes usually gives index for any **prefix** of these attributes, due to lexicographic sorting.

# Lexicographic Sorting



# Problem Session

---

- Which point and range queries are "easy" (equality on a prefix, range on one) when the relation is indexed with this two-attribute index?

```
CREATE INDEX myIndex  
ON person (height, birthdate);
```

1. A range query on height?
2. A range query on birthdate?
3. A point query on birthdate?
4. A point query on birthdate combined with a range query on height?
5. A point query on height combined with a range query on birthdate?

# Choosing to Use an Index

---

- The choice of whether to use an index is made by the DBMS *for every instance of a query*
  - May depend on query parameters
  - You do not need to take indexes into account when writing queries
- Estimating selectivity is done using statistics
  - In PostgreSQL, statistics are collected by executing statements such as `ANALYZE table_name;`

# Choosing Columns

---

- Candidates for index search keys
  - Columns in WHERE clauses
  - Columns in GROUP BY clauses
  - Columns in ORDER BY clauses
- Columns that are rarely candidates
  - Large columns (too much space)
  - Frequently updated columns (too much maintenance)
  - Columns in SELECT clauses (not used to find tuples)
    - ... but see **covering indices!**

# What Speaks Against Indexing

- Space usage:
  - Similar to size of indexed columns (plus pointer)
  - Most space for leaves, less for tree nodes
  - Not really important!
- Time usage for keeping indexes updated under data insertion/change:
  - Depends on the index architecture
  - This IS important!

- Storage hierarchy:
  - Different layers have different characteristics.
  - Sequential access is faster than random access for all layers, but especially for hard disks.
  - DBMSs have various optimizations to minimize data access latency.
- Databases are organized in pages.
- Large databases need to be equipped with suitable indexes
  - Need understanding of what indexes might help a given set of queries.
  - *Key distinction:* Clustered vs. Unclustered.
  - A detailed understanding of various index types is beyond the scope of this course.

# Introduction to Database Systems

## IDBS - Autumn 2024

### Lecture 8 - Indexing & Query Processing

B+ Trees

Performance Tuning

Query Processing

---

Readings: PDBM 12.3.8, 13.1

Omar Shahbaz Khan

# Last Time in IDBS...

# Indexing

- Indexes are data structures that facilitate access to data from disk
  - ... if conditions are a prefix of indexed attributes
  - Clustered indexes store tuples that match a range condition together
  - Some queries can be answered looking only at the index (a covering index for query)
  - Indexes slow down updates and insertions
- The choice of whether to use an index is made by the DBMS for every instance of a query
  - May depend on query parameters
  - Don't have to name indexes when writing queries

# Wake Up Task!

- Normalize the following relation  $R(A,B,C,D,E,F)$  into 3NF/BCNF
  - Functional Dependencies:
    - $AB \rightarrow EF$
    - $A \rightarrow C$
    - $E \rightarrow D$
  - Candidate key(s):
  - Prime Attributes:
  - Non-Prime Attributes:
  - Current Normal Form:
  - Decomposition:

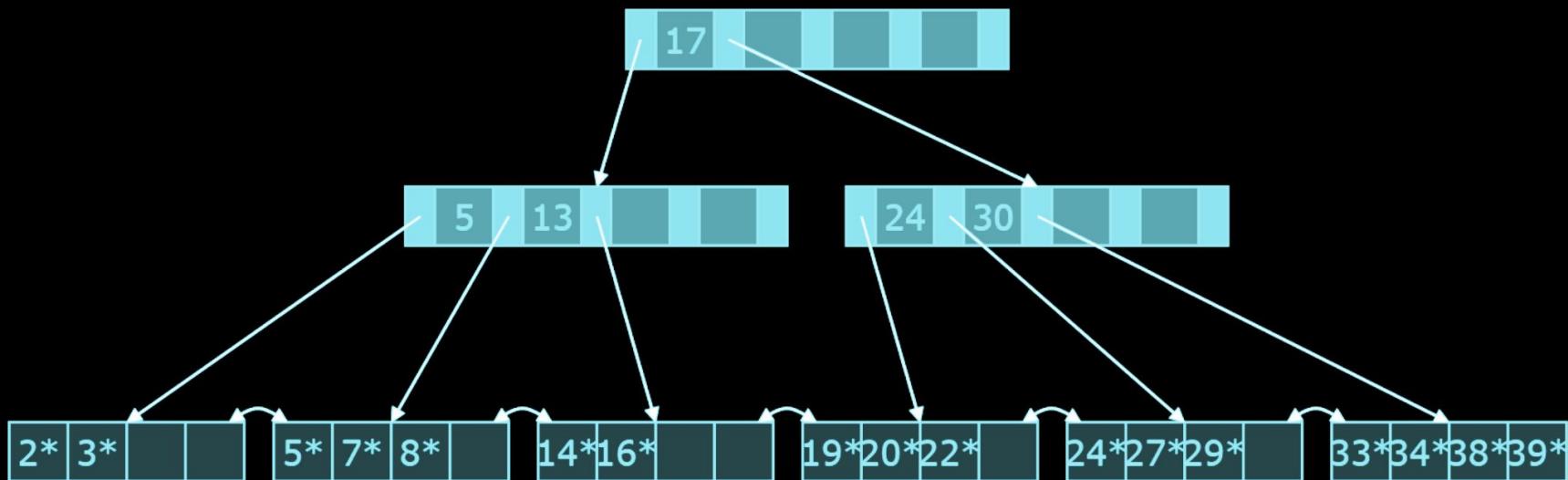
# Wake Up Task!

- Normalize the following relation  $R(A,B,C,D,E,F)$  into 3NF/BCNF
  - Functional Dependencies:
    - $AB \rightarrow EF$
    - $A \rightarrow C$
    - $E \rightarrow D$
  - Candidate key(s): AB
  - Prime Attributes: A, B
  - Non-Prime Attributes: C, D, E, F
  - Current Normal Form: 1NF
  - Decomposition:  $R1(A,C)$     $R2(D,E)$     $R3(A,B,E,F)$

# This Time...

# -- TODO

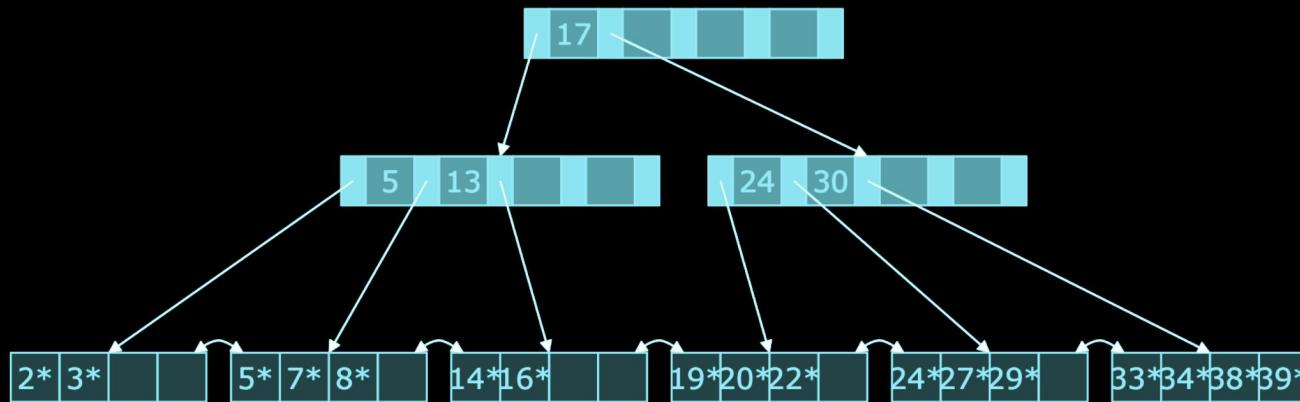
- B+ Tree
- Processing Selections
- Query Evaluation
  - Intro to Join Evaluation
  - Intro to Grouping/Aggregation (brief)



# B+ Trees

- The most common index type
  - ... in relational systems
- Supports equality and range queries
- Dynamic structure
  - Adapts to insertions and deletions
  - Maintains a balanced tree

# A Sample B+ Tree



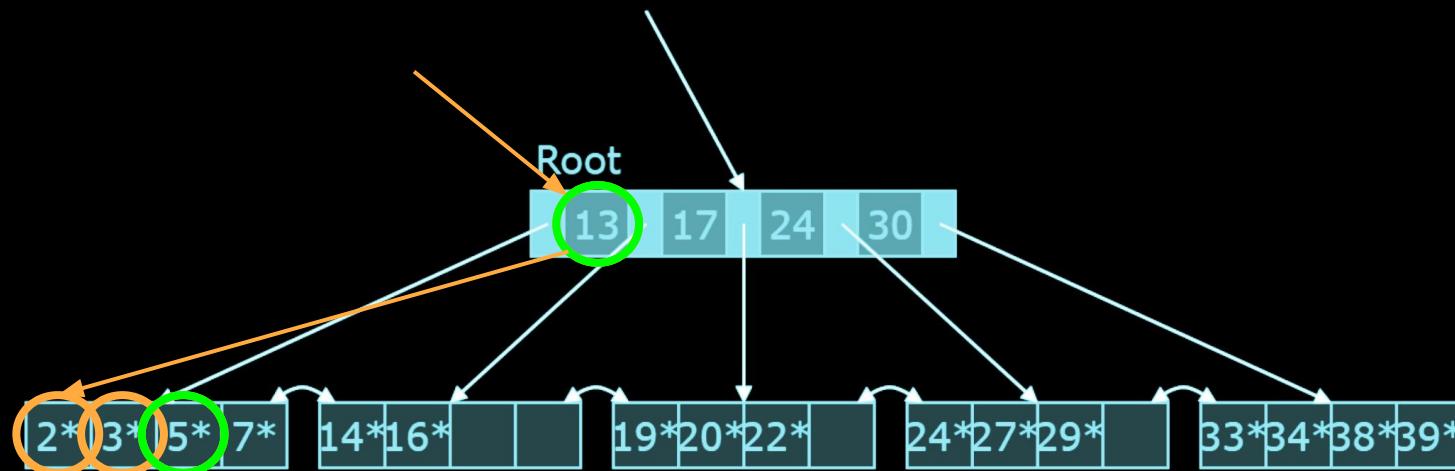
- $X^*$  represents (search key, pointer list) pairs  
( $X$ , [address of tuple  $X_1$ , ...])
  - Unique index: Only one entry in list

- Key values are sorted:  $K_1 < \dots < K_d$   
( $d$  is maximum capacity or order of node)
  - For any two adjacent key values  $(K_i, K_{i+1})$ ,  
the pointer  $P_i$  points to a node covering all values in the interval  $[K_i, K_{i+1}]$

Each node / leaf is one disk page!  
Nodes have a minimum and maximum capacity

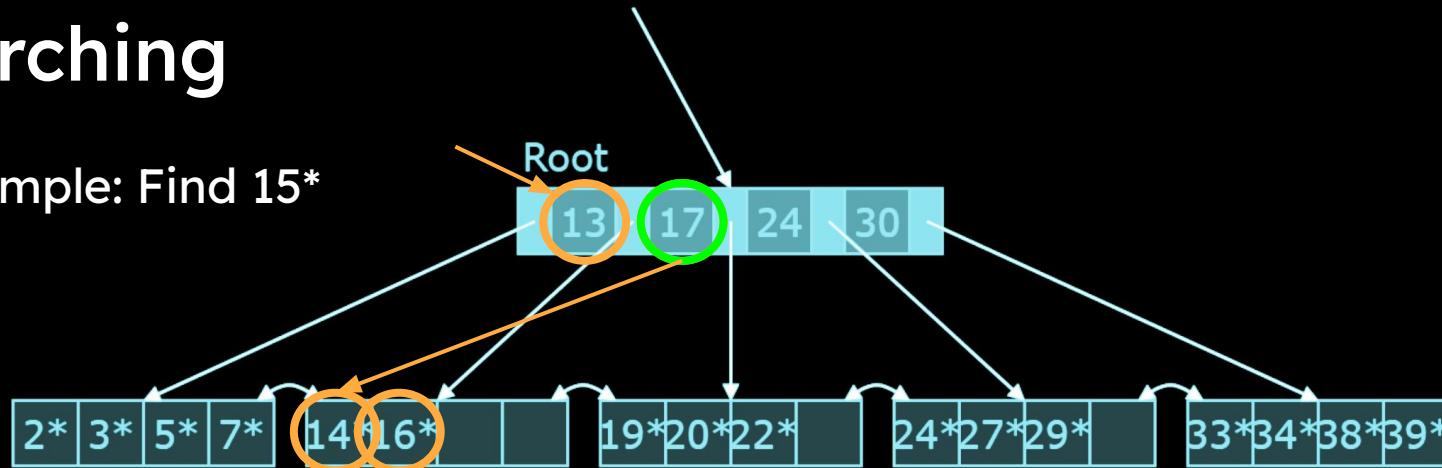
# Searching

- Begin at the root
- Comparisons guide the search to the appropriate leaf
- Example: Find 5\*

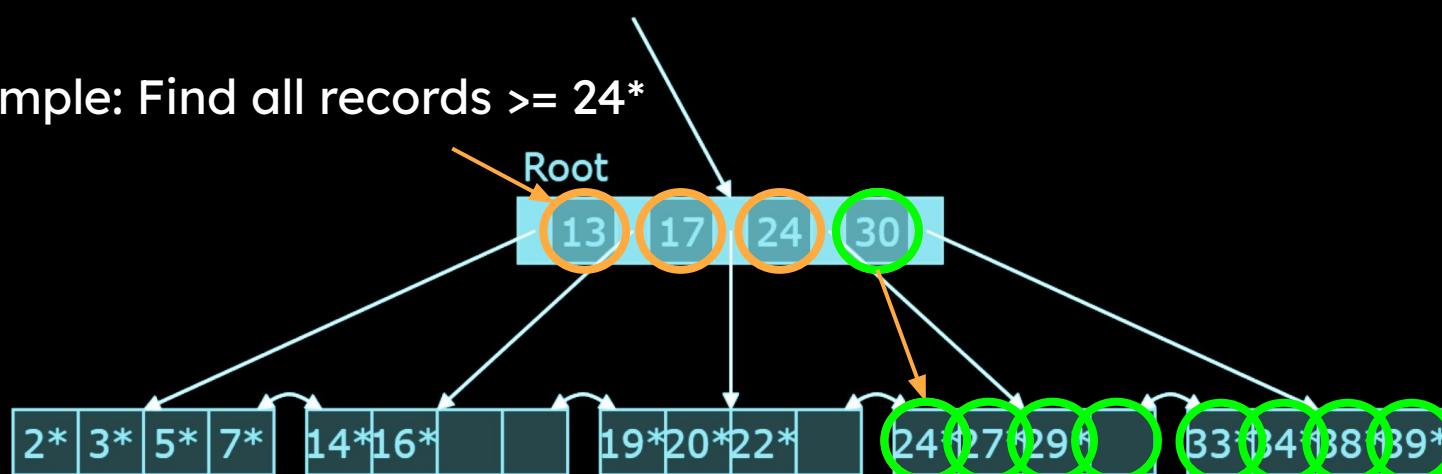


# Searching

- Example: Find  $15^*$



- Example: Find all records  $\geq 24^*$

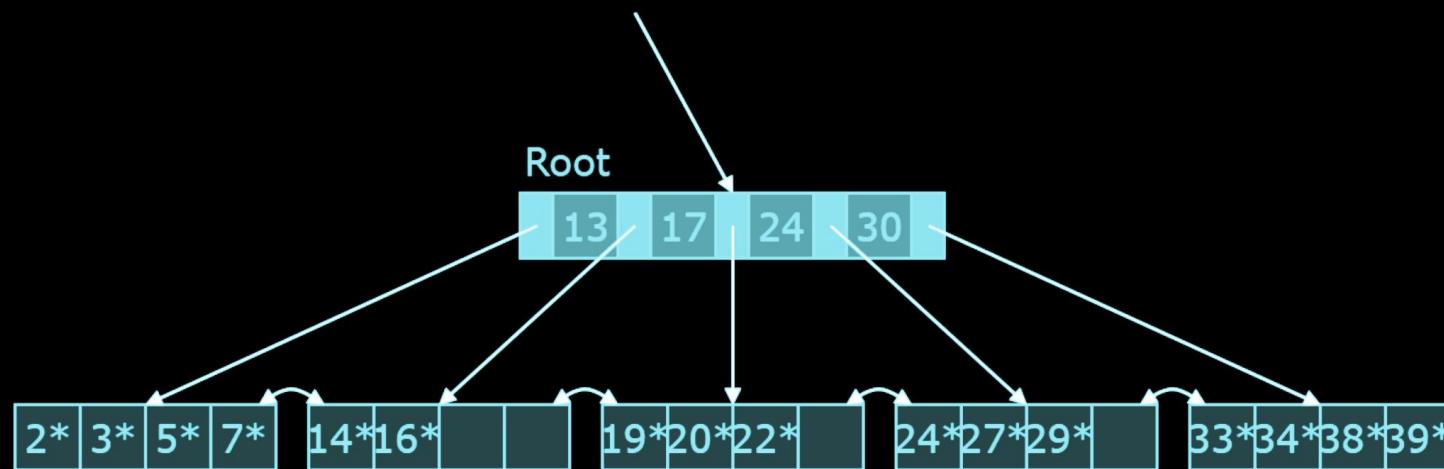


# Intra-Node Searching

- I have used scans
- B+ tree nodes have hundreds of key values
- Use binary search!

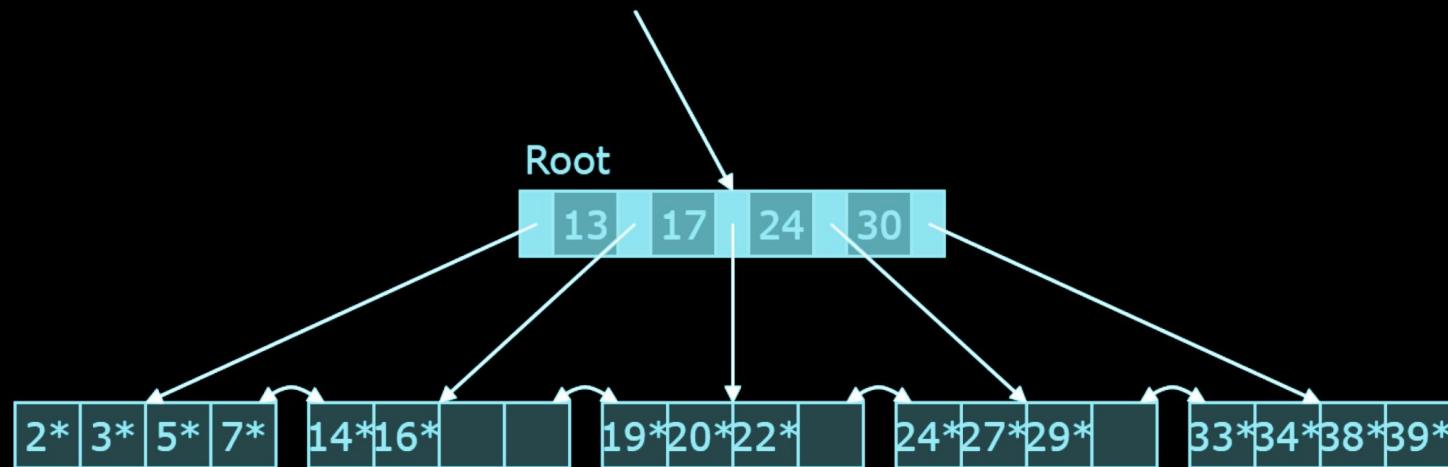
# Inserting 8\*

- Leaf is full, must therefore split

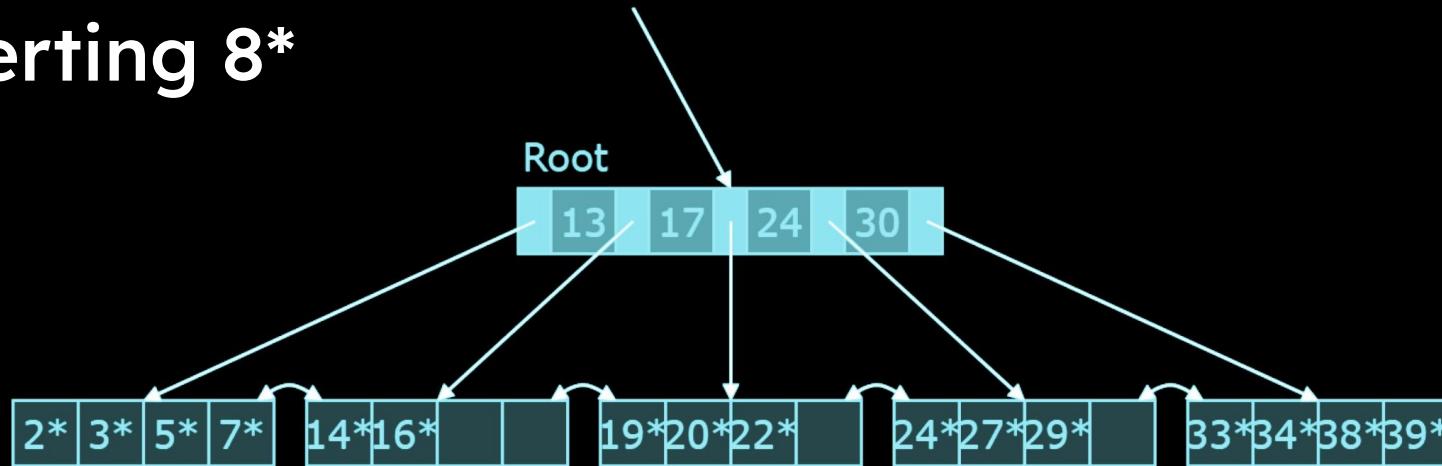


# Inserting 8\*

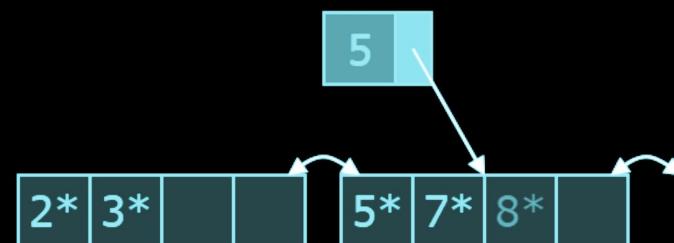
- Leaf is full, must therefore split
- Root is full, must therefore split



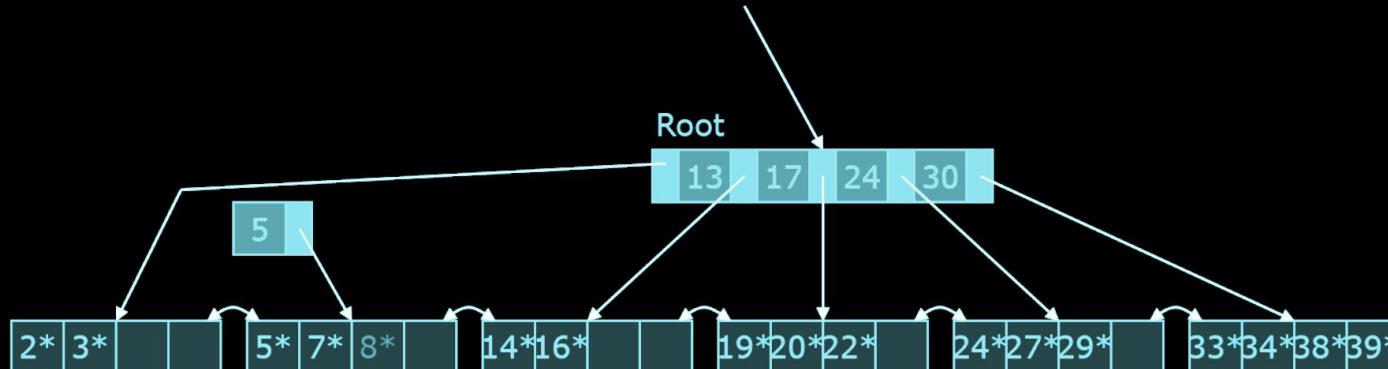
# Inserting 8\*



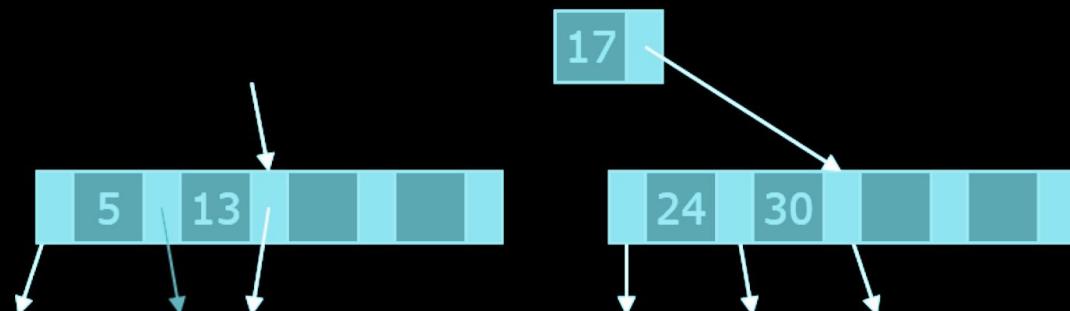
- First split the leaf
- Copy middle search key to parent



# Inserting 8\*

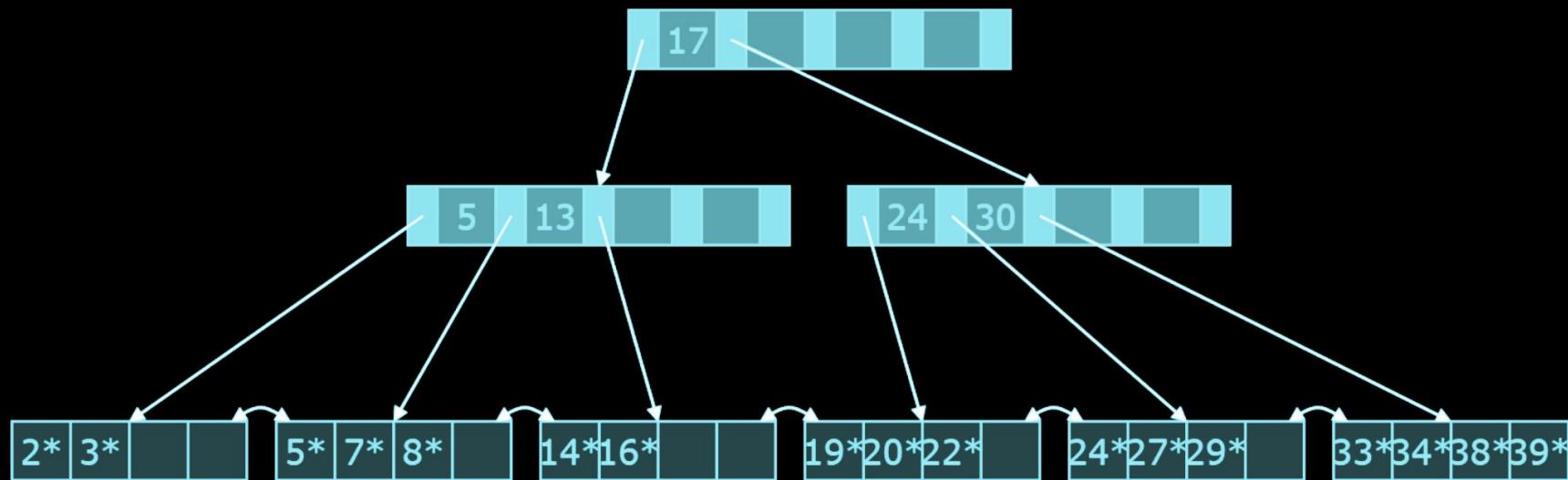


- Then split the root
- Move the middle search key  
Into the new root



# After Inserting 8\*

- Trees grow wider, then higher



# Storage Capacity

- A typical tree:
  - Order: 1000 ( $\sim=16$  KB per page / 16 bytes per entry)
  - Utilization: 67% (usual numbers in real life)
  - Fanout: 670
- Capacity
  - Root: 670 records
  - Two levels:  $670^2 = 448,900$  records
  - Three levels:  $670^3 = 300,763,000$  records
  - Four levels:  $670^4 = 201,511,210,000$  records
- Top levels may fit in memory
  - Level 1 = 1 page = 16 KB
  - Level 2 = 670 pages = 11 MB
  - Level 3 = 448,900 pages = 7 GB

# Clustered Indexes in Major DBMS

- PostgreSQL
  - Cannot specify a clustered index!
  - Manual `CLUSTER` command!
- SQL Server
  - Table stored in clustered index
  - Primary keys can be unclustered
  - Indexes maintained dynamically
- DB2
  - Table stored in clustered index
  - Explicit command for index reorganization
- Oracle
  - No clustered index until 10g
  - Index organized table (unique/clustered)
  - Indexes maintained dynamically
- MySQL
  - Primary key is clustered
  - Table stored in clustered index
  - Indexes maintained dynamically

# -- TODO

✓ B+ Tree

- Processing Selections
- Query Evaluation
  - Intro to Join Evaluation
  - Intro to Grouping/Aggregation (brief)

# Selections

# Processing Simple Selections

- Point and range queries on the attribute(s) of a clustered index are almost always best performed using an index seek/scan
- Unclustered indexes should only be used with high selectivity queries
- Exception: Covering index is good for any selectivity
- If no index exists, a full table scan is required!
- If no “good” index exists, a full table scan is preferred!

# Processing Complex Selections

- We consider the conjunction ("and") of equality and range conditions
- No relevant index: Full table scan
- One index relevant:
  - Highly selective: Use that index
  - If not: Full table scan
- Multiple relevant indexes:
  - One is highly selective: Use that index
  - No single condition matching an index that is highly selective: Can “intersect” the returned sets

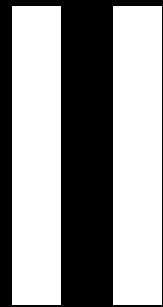
```
SELECT *
  FROM A
 WHERE year = 2024
   AND lecture = 8
```

# Using a Highly Selective Index

- Basic idea:
  - Retrieve all matching tuples (few)
  - Filter according to remaining conditions
- If index is clustered or *covering*:  
Retrieving tuples is particularly efficient, and the index does not need to be highly selective.

# Using several Less Selective Indexes

- For several conditions  $C_1, C_2, \dots$  matched by indexes:
  - Retrieve the addresses  $R_i$  of tuples matching  $C_i$ 
    - The addresses are in the index leaves!
  - Compute the intersection  $R = R_1 \cap R_2 \cap \dots$
  - Retrieve the tuples in  $R$  from disk (in sorted order)
- Remaining problem:
  - How can we estimate the selectivity of a condition or of a combination of conditions?
  - Use some stats and probabilistic assumptions...



# Example

Query:

```
SELECT title  
      FROM Movie  
     WHERE year = 1990  
       AND studioName = 'Disney';
```

**Examples of strategies:**

1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine two indexes to identify rows fulfilling both conditions.
5. Use one composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 and their titles in a composite covering index.

# Example - Variant 1

Query:

```
SELECT title  
      FROM Movie  
     WHERE year = 1990  
           AND studioName = 'Disney';
```

## Available Indexes

```
CREATE INDEX yearIdx ON Movie(year);  
CREATE INDEX studIdx ON Movie(studioName);
```

Which strategies are possible and which indexes would be used?

## Examples of strategies:

1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine two indexes to identify rows fulfilling both conditions.
5. Use one composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 and their titles in a composite covering index.

# Example - Variant 2

Query:

```
SELECT title  
      FROM Movie  
     WHERE year = 1990  
           AND studioName = 'Disney';
```

## Available Indexes

```
CREATE INDEX yearIdx ON Movie(year);  
CREATE INDEX yearStudIdx ON Movie(year,  
studioName);  
CREATE INDEX coveringIdx ON Movie(year,  
studioName, title); --INCLUDE (title);
```

Which strategies are possible and what indexes would be used?

## Examples of strategies:

1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine two indexes to identify rows fulfilling both conditions.
5. Use one composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 and their titles in a composite covering index.

# Example - Variant 3

Query:

```
SELECT title  
      FROM Movie  
     WHERE year = 1990  
       AND studioName = 'Disney';
```

## Available Indexes

```
CREATE INDEX idIdx ON Movie(id);  
CREATE INDEX titleIdx ON Movie(title);
```

Which strategies are possible and what indexes would be used?

## Examples of strategies:

1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine two indexes to identify rows fulfilling both conditions.
5. Use one composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 and their titles in a composite covering index.

# Processing Complex Selections Revisited

- We have considered the conjunction ("and") of a number of equality and range conditions.
- What about disjunctive ("or") selections?
  - One full table scan
  - OR
  - Multiple "and" queries

# -- TODO

- ✓ B+ Tree
- ✓ Processing Selections
- Query Evaluation
  - Intro to Join Evaluation
  - Intro to Grouping/Aggregation (brief)



imgflip.com

# Query Evaluation in a Nutshell

- SQL rewritten to (extended) relational algebra
- The building blocks in DBMS query evaluation are algorithms that implement relational algebra operations.
  - **Join** is the most important one!
- May be based on:
  - Reading everything / Sorting / Hashing
  - Using indexes can sometimes help!
- The DBMS optimizer knows the characteristics of each approach, and attempts to use the best one in a given setting

# Join Evaluation in a Nutshell

- Join is the most important operation!
- May be based on:
  - Reading everything / Sorting / Hashing
  - Using indexes can sometimes help!
- We consider a simple join:

R JOIN S ON S.ID = R.ID

- Extends to more complex joins in a straightforward way

# Nested Loop Joins

- The following basic algorithm can be used for any join

```
for each tuple in R
    for each tuple in S
        if r.ID = s.ID
            then output (r, s)
```

- If the join condition is complex/broad, sometimes this is the only/best choice

R JOIN S ON S.ID <>> R.ID

- See animation example on LearnIT

# Role of Index in Nested Loop Joins

- If there is an index that matches the join condition, the following algorithm can be considered:
  - For each tuple in R use the index to locate matching tuples in S
- See animation example on LearnIT
- Good if  $|R|$  is small compared to  $|S|$
- If many tuples match each tuple, a clustered or covering index is preferable

# Example

## Query:

```
SELECT *
  FROM Movie M,
       JOIN Producer P ON M.producer = P.id
 WHERE M.year = 2015
   AND P.birthdate < '1940-01-01';
```

## Some Possible strategies:

1. Use index to find 2015 tuples, use index to find matching tuples in Producer.
2. Use index to find producers born before 1940, use index to find matching movies.
3. NL join Movie and Producer, then filter.

# Problem Session

What would be good indexes for this query?

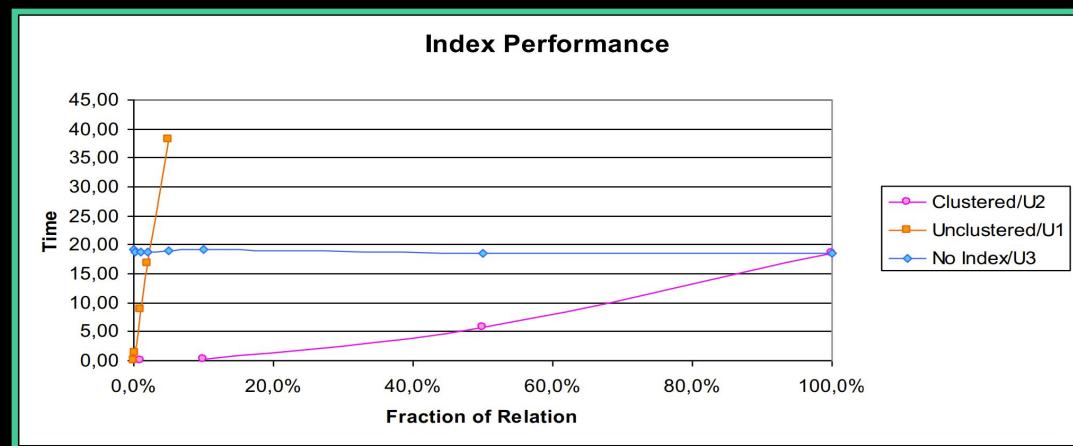
```
SELECT A.street, A.streetno  
  FROM person P  
 JOIN address A ON A.person_id = P.id  
 WHERE P.lastname = 'Bohr'  
   AND P.firstname LIKE 'Niels%';
```

# Merge Join

- $R \text{ JOIN } S \text{ ON } R.ID = S.ID$
- Merge Join:
  - Step 0: Sort R and S on ID
  - Step 1: Merge the sorted R and S
- See animation example on LearnIT
- Cost:
  - If already sorted:  $O(|R| + |S|)$
  - If not sorted:  $O(|R|\log|R| + |S|\log|S| + |R| + |S|)$
  - Can we do better?

# Role of Index in Merge Joins

- Indexes can be used to read data in sorted order
- When is this a win?
  - Index is clustered
  - Index is covering
- When is this a loss?
  - Index is unclustered

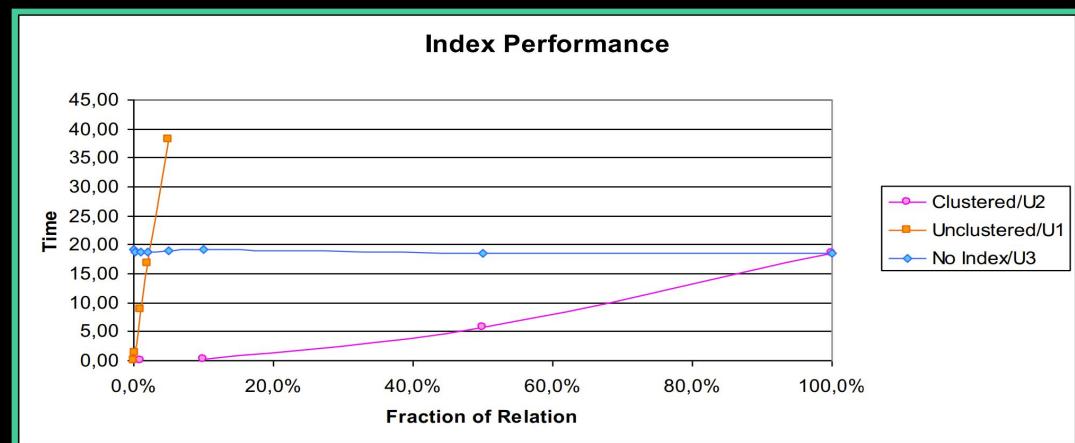


# Hash Join

- $R \text{ JOIN } S \text{ ON } R.ID = S.ID$ 
  - Best if  $S$  fits in RAM
- Hash Join:
  - Step 0: Create a good hash function for ID
  - Step 1: Create a hash table for  $S$  in memory
  - Step 2: Scan  $R$  and look for matching tuples in the hash table
- See animation example on LearnIT
- Cost:  $O(|R| + |S|)$ 
  - What if  $S$  does not fit in RAM?

# Role of Index in Hash Joins

- Hash joins read all the relations by default
- How can indexes be useful?
  - Apply to non-join conditions - before join
  - Index is covering



# Comparison of Join Algorithms

- Nested loops join
  - Very costly  $O(|R| \cdot |S|)$
  - Works for any condition → sometimes only option
- Merge join
  - Works well if data is clustered
  - Works well if relations are large and similar in size
  - Can deal with range conditions iff the tables are sorted on the join key
- Hash join
  - Works well if one relation is small
  - Is that often the case?
  - Can only deal with equi-joins

# Grouping Operations

- Many operations are based on grouping records (of one or more relations) according to the values of some attribute(s):
  - Join (group by join attributes)
  - Group by and aggregation (obvious)
  - Set operations (group by all attributes)
  - Duplicate elimination (group by all attributes)
- Most database systems implement such grouping efficiently using sorting or hashing

# -- TODO

- ✓ B+ Tree
- ✓ Processing Selections
- ✓ Query Evaluation
  - ✓ Intro to Join Evaluation
  - ✓ Intro to Grouping/Aggregation (brief)

# Takeaways

My query is so slow

Have you tried  
creating an index?

**The performance difference between well-tuned  
and poorly-tuned applications can be massive!**

**The DBMS does its best to optimize queries, but  
sometimes it needs help!**

- Query tuning - rewrite as joins or non-correlated subqueries
- Indexes - solve 90+% of all other performance problems
- **EXPLAIN <QUERY>; EXPLAIN ANALYZE <QUERY>;**

**If that is not sufficient...**

- Materialized views / Partitioning / Denormalization
- [PostgreSQL Query Planner](#)
- Beyond the scope of this course!

# Next Time in IDBS...

# Introduction to Database Systems

## IDBS - Autumn 2024

### Lecture 9 - Relational Databases

Storage models

Architecture of a DBMS

Main memory DBMSs

---

Readings: PDBM 2

Martin Hentschel

# Partitioning of Tables

- A table might be a performance bottleneck
  - If it is heavily used, causing locking contention (more on this later in course)
  - If its index is deep (table has many rows or search key is wide), increasing I/O
  - If rows are wide, increasing I/O
- Table partitioning might be a solution to this problem.

# Horizontal Partitioning

- If accesses are confined to disjoint subsets of rows, partition table into smaller tables containing the subsets
  - Geographically, organizationally, active/inactive
- Advantages:
  - Spreads users out and reduces contention
  - Rows in a typical result set are concentrated in fewer pages
- Disadvantages:
  - Added complexity
  - Difficult to handle queries over all tables

# Vertical Partitioning

- Split columns into two subsets, replicate key
- Useful when table has many columns and
  - it is possible to distinguish between frequently and infrequently accessed columns
  - different queries use different subsets of columns
- Example: Employee table
  - Columns related to compensation (tax, benefits, salary) split from columns related to job (department, projects, skills)
- DBMS trend (for analytics):
  - Column stores, with full vertical partitioning.
  - More on this next week.

# Introduction to Database Systems

## IDBS – Fall 2024

- Week 9:
- Storage Models
- Architecture of a DBMS
- Main-Memory DBMSs

---

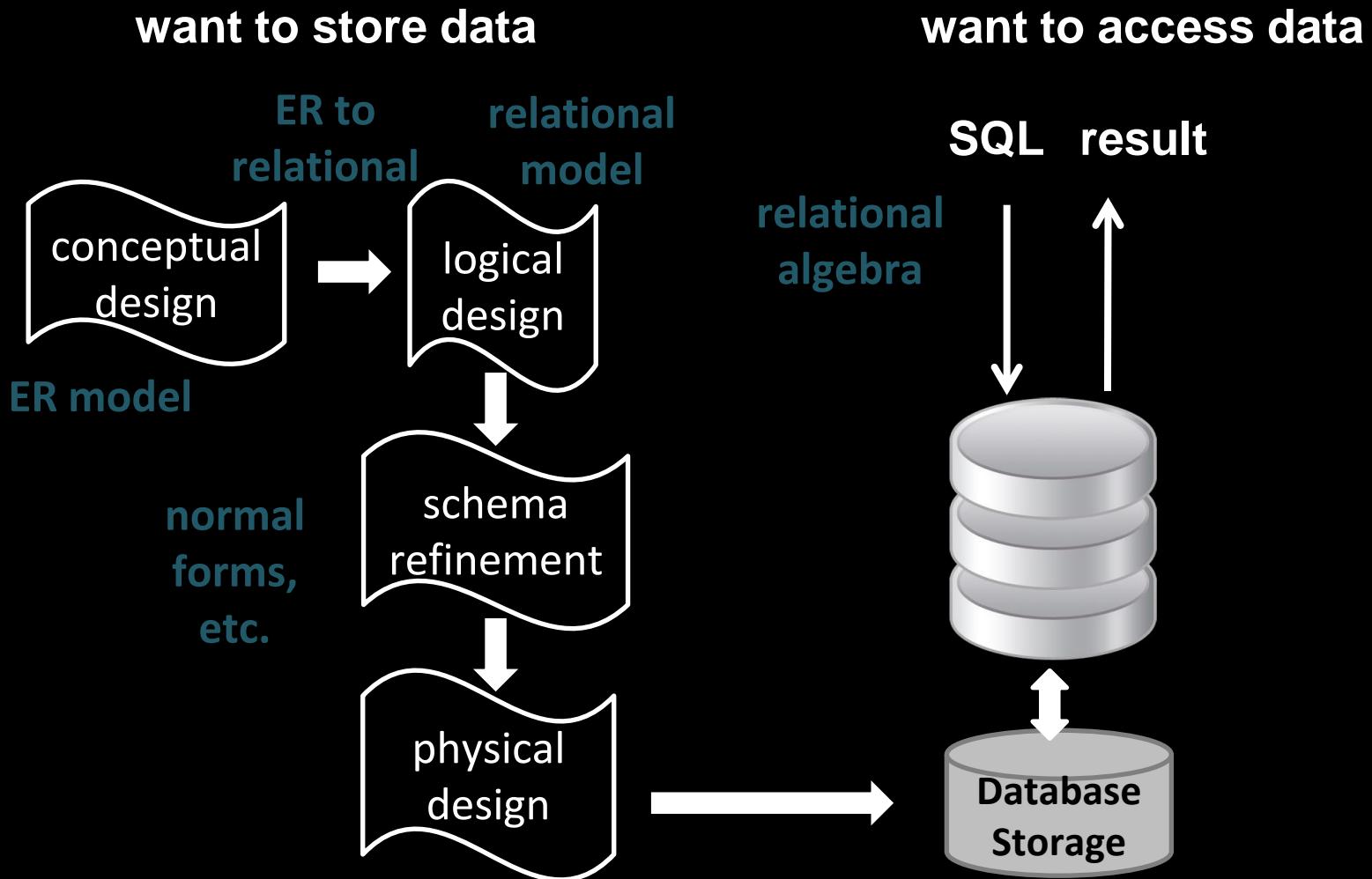
Martin Hentschel

Slide credits: Eleni Tzirita Zacharatou

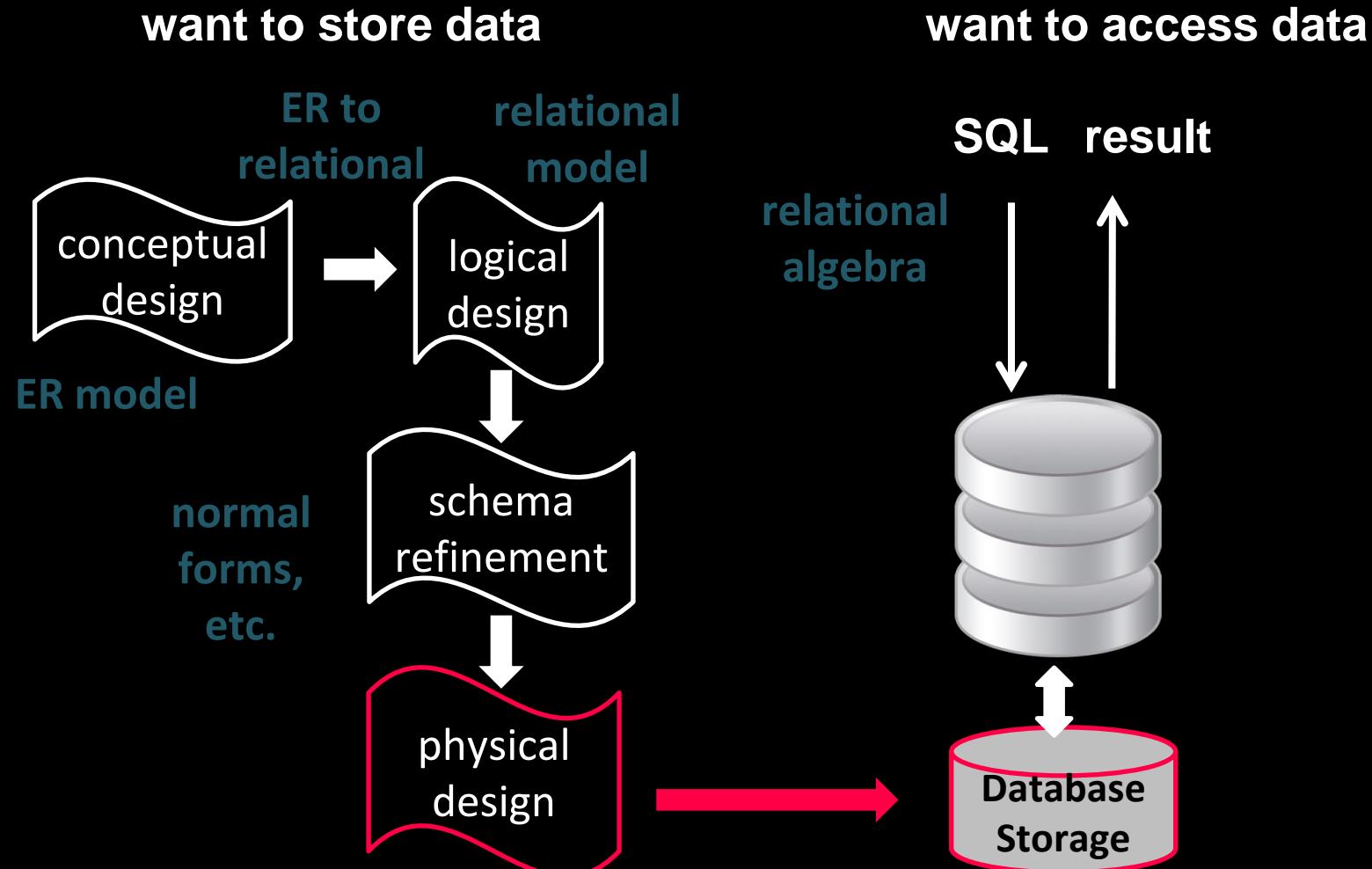
Readings:

PDBM2 + more material on LearnIT

# The Bigger Picture



# The Bigger Picture



**Today we will talk about the lowest physical representation of data in a database**

# Storage Models

---

# Database Workloads

---

## **On-Line Transaction Processing (OLTP)**

→ Fast operations that only read/update a small amount of data each time

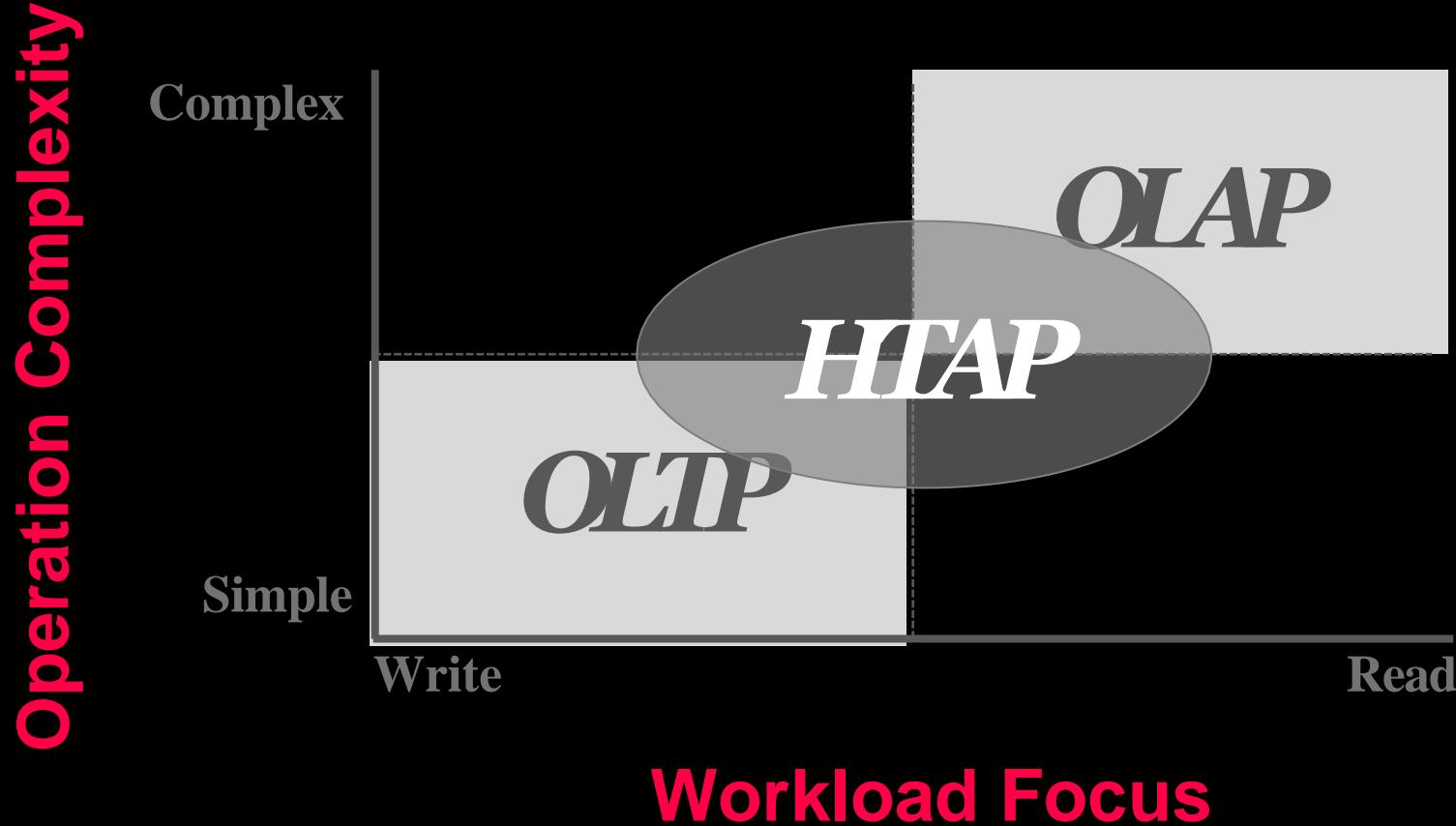
## **On-Line Analytical Processing (OLAP)**

→ Complex queries that read a lot of data to compute aggregates

## **Hybrid Transaction + Analytical Processing (HTAP)**

→ OLTP + OLAP together on the same database instance

# Database Workloads



# OLTP

---

## On-Line Transaction Processing:

→ Simple queries that read/update a small amount of data that is related to a single entity in the database.

This is usually the kind of application that people build first.

```
SELECT P.* , R.*  
FROM pages AS P  
INNER JOIN revisions AS R  
ON P.latest = R.revID  
WHERE P.pageID = ?
```

```
UPDATE useracct  
SET lastLogin = NOW(),  
    hostname = ?  
WHERE userID = ?
```

```
INSERT INTO revisions  
VALUES (?, ?, ?, ?)
```

# OLAP

## **On-Line Analytical Processing:**

→ Complex queries that read large portions of the database spanning multiple entities.

You execute these workloads on the data you have collected from your OLTP application(s).

```
SELECT COUNT(U.lastLogin), EXTRACT(month FROM
    U.lastLogin) AS month
FROM useracct AS U
WHERE U.hostname LIKE '%.gov'
GROUP BY EXTRACT(month FROM U.lastLogin)
```

# Exercise

---

Consider the following relation:

- Sales (ProductID, ProductName, Date, Quantity, Revenue)

## Query A

```
SELECT ProductID, SUM(Quantity)  
AS TotalQuantitySold  
FROM Sales  
WHERE Date BETWEEN '2023-01-01'  
AND '2023-12-31'  
GROUP BY ProductID;
```

## Query B

```
UPDATE Sales  
SET Revenue = Revenue * 1.1  
WHERE ProductID = 12345 AND  
Date = '2023-03-10';
```

# Observation

The relational model *does not* specify that the DBMS must store all a tuple's attributes together on a single page.

This *may not* be the best layout for some workloads.

# Storage Models

A DBMS's storage model specifies how it physically organizes tuples on disk and in memory.

- Can have different performance characteristics based on the target workload (OLTP vs. OLAP).
- Influences the design choices of the rest of the DBMS.

**Choice #1: N-ary Storage Model (NSM)**

**Choice #2: Decomposition Storage Model (DSM)**

**Choice #3: Hybrid Storage Model (PAX)**

# N-ary Storage Model (NSM)

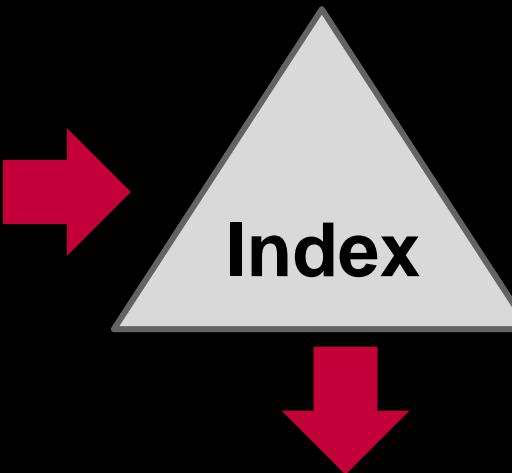
The DBMS stores (almost) all attributes for a single tuple contiguously on a single page.

→ Also known as a "row store"

Ideal for OLTP workloads where queries are more likely to access individual entities and execute write-heavy workloads.

# NSM: OLTP Example

```
SELECT * FROM useracct  
WHERE userName = ?  
AND userPass = ?
```



NSM Disk Page

header	userID	userName	userPass	hostname	lastLogin
header	12345	user1	password1	host1	2023-01-01
header	12345	user2	password2	host2	2023-01-01
header	-	-	-	-	-



Database File

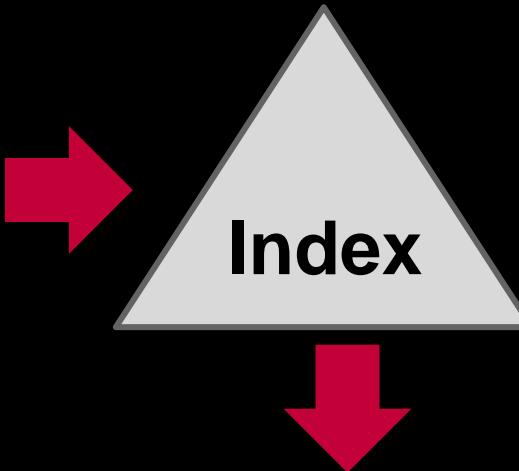


Disk

# NSM: OLTP Example

```
SELECT * FROM useracct  
WHERE userName = ?  
AND userPass = ?
```

```
INSERT INTO useracct  
VALUES ( ?, ?, ...? )
```



NSM Disk Page

<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin



Database File

Disk



# NSM: OLAP Example

```
SELECT COUNT(U.lastLogin),
      EXTRACT(month FROM U.lastLogin) AS month
FROM useracct AS U
WHERE U.hostname LIKE '%.gov'
GROUP BY EXTRACT(month FROM U.lastLogin)
```



Disk

Database File



NSM Disk Page

<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin

# NSM: OLAP Example

```
SELECT COUNT(U.lastLogin),  
       EXTRACT(month FROM U.lastLogin) AS month  
  FROM useracct AS U  
 WHERE U.hostname LIKE '%.gov'  
 GROUP BY EXTRACT(month FROM U.lastLogin)
```



Disk



NSM Disk Page

header	userID	userName	userPass	hostname	lastLogin
header	userID	userName	userPass	hostname	lastLogin
header	userID	userName	userPass	hostname	lastLogin
header	userID	userName	userPass	hostname	lastLogin

# Reflection

---

- Do you see a problem in the previous OLAP example?
- Is NSM a good choice for OLAP queries?
- Can you think of some drawbacks of NSM?

# NSM: OLAP Example

```
SELECT COUNT(U.lastLogin),  
       EXTRACT(month FROM U.lastLogin) AS month  
  FROM useracct AS U  
 WHERE U.hostname LIKE '%.gov'  
 GROUP BY EXTRACT(month FROM U.lastLogin)
```



Disk

Database File



NSM Disk Page

header	userID	userName	userPass	hostname	lastLogin
header	userID	userName	userPass	hostname	lastLogin
header	userID	userName	userPass	hostname	lastLogin
header	userID	userName	userPass	hostname	lastLogin
header	userID	userName	userPass	hostname	lastLogin

Useless Data

# NSM: Summary

## **Advantages**

- Fast inserts, updates, and deletes.
- Good for queries that need the entire tuple (OLTP).
- Can use index-oriented physical storage for clustering.

## **Disadvantages**

- Not good for scanning large portions of the table and/or a subset of the attributes.
- Terrible memory locality in access patterns.
- Not ideal for compression because of multiple value domains within a single page.

# Decomposition Storage Model (DSM)

The DBMS stores a single attribute for all tuples contiguously in a block of data.

→ Also known as a "column store"

Ideal for OLAP workloads where read-only queries perform large scans over a subset of the table's attributes.

DBMS is responsible for combining/splitting a tuple's attributes when reading/writing.

# DSM: Database Example

---

The DBMS stores the values of a single attribute across multiple tuples contiguously in a page.

→ Also known as a "column store".

<b>header</b>	userID	userName	userPass	hostname	lastLogin
<b>header</b>	userID	userName	userPass	hostname	lastLogin
<b>header</b>	userID	userName	userPass	hostname	lastLogin
<b>header</b>	userID	userName	userPass	hostname	lastLogin

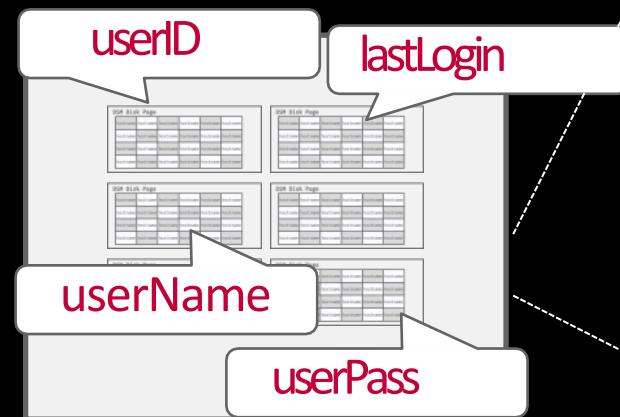
# DSM: Database Example

The DBMS stores the values of a single attribute across multiple tuples contiguously in a page.

→ Also known as a "column store".



Disk



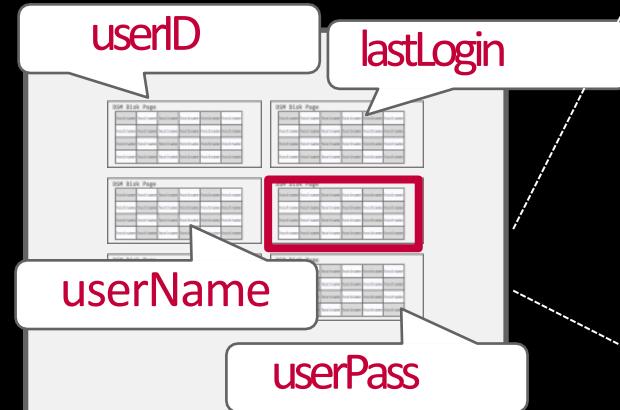
header		hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname	hostname

# DSM: OLAP Example

```
SELECT COUNT(U.lastLogin),  
       EXTRACT(month FROM U.lastLogin) AS month  
FROM useracct AS U  
WHERE U.hostname LIKE '%.gov'  
GROUP BY EXTRACT(month FROM U.lastLogin)
```



Disk



DSM Disk Page

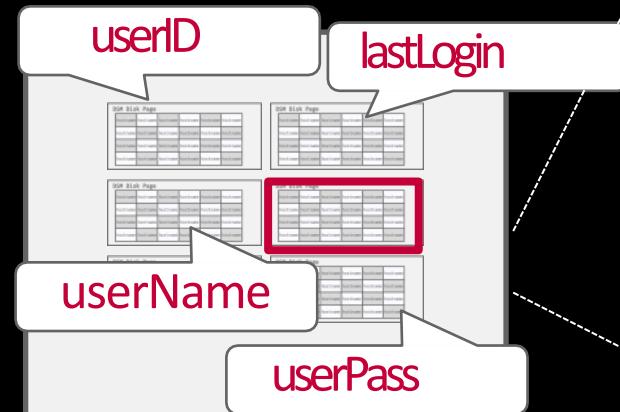
header	hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname

# DSM: OLAP Example

```
SELECT COUNT(U.lastLogin),  
       EXTRACT(month FROM U.lastLogin) AS month  
FROM useracct AS U  
WHERE U.hostname LIKE '%.gov'  
GROUP BY EXTRACT(month FROM U.lastLogin)
```



Disk



DSM Disk Page

header	hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname

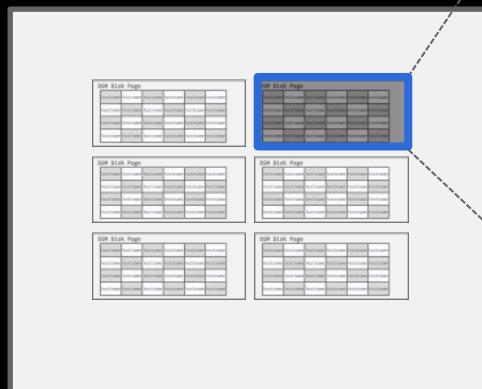
# DSM: OLAP Example

```
SELECT COUNT(U.lastLogin),  
       EXTRACT(month FROM U.lastLogin) AS month  
FROM useracct AS U  
WHERE U.hostname LIKE '%.gov'  
GROUP BY EXTRACT(month FROM U.lastLogin)
```



Disk

Database File



DSM Disk Page

header	lastLogin	lastLogin	lastLogin	lastLogin
lastLogin	lastLogin	lastLogin	lastLogin	lastLogin
lastLogin	lastLogin	lastLogin	lastLogin	lastLogin
lastLogin	lastLogin	lastLogin	lastLogin	lastLogin

# Reflection

---

- Is DSM a good choice for OLTP queries?

```
SELECT * FROM useracct  
WHERE userName = ?  
AND userPass = ?
```

- Can you think of some drawbacks of DSM?

# DSM: Summary

## **Advantages**

- Reduces the amount wasted I/O per query because the DBMS only reads the data that it needs.
- Faster query processing because of increased locality and cached data reuse.
- Better data compression.

## **Disadvantages**

- Slow for point queries, inserts, updates, and deletes because of tuple splitting/stitching/reorganization.

# Optimal Layout

## **NSM**

easy to implement

good if you are accessing  
the whole tuple or  
most columns of a tuple

## **DSM**

have tuple reconstruction cost

good if you are accessing one or  
a few columns of a tuple

**Choice depends on the workload!**

# Systems in the Wild

---

row stores



column stores

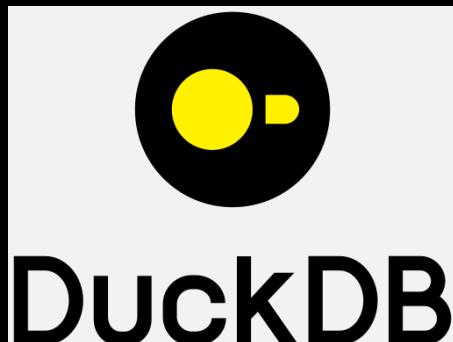


# Systems in the Wild

big vendors offering different products with both row & column stores



mix of row & column stores





# DBMS Architecture

---

## Main-Memory DBMSs

# DBMS Architecture

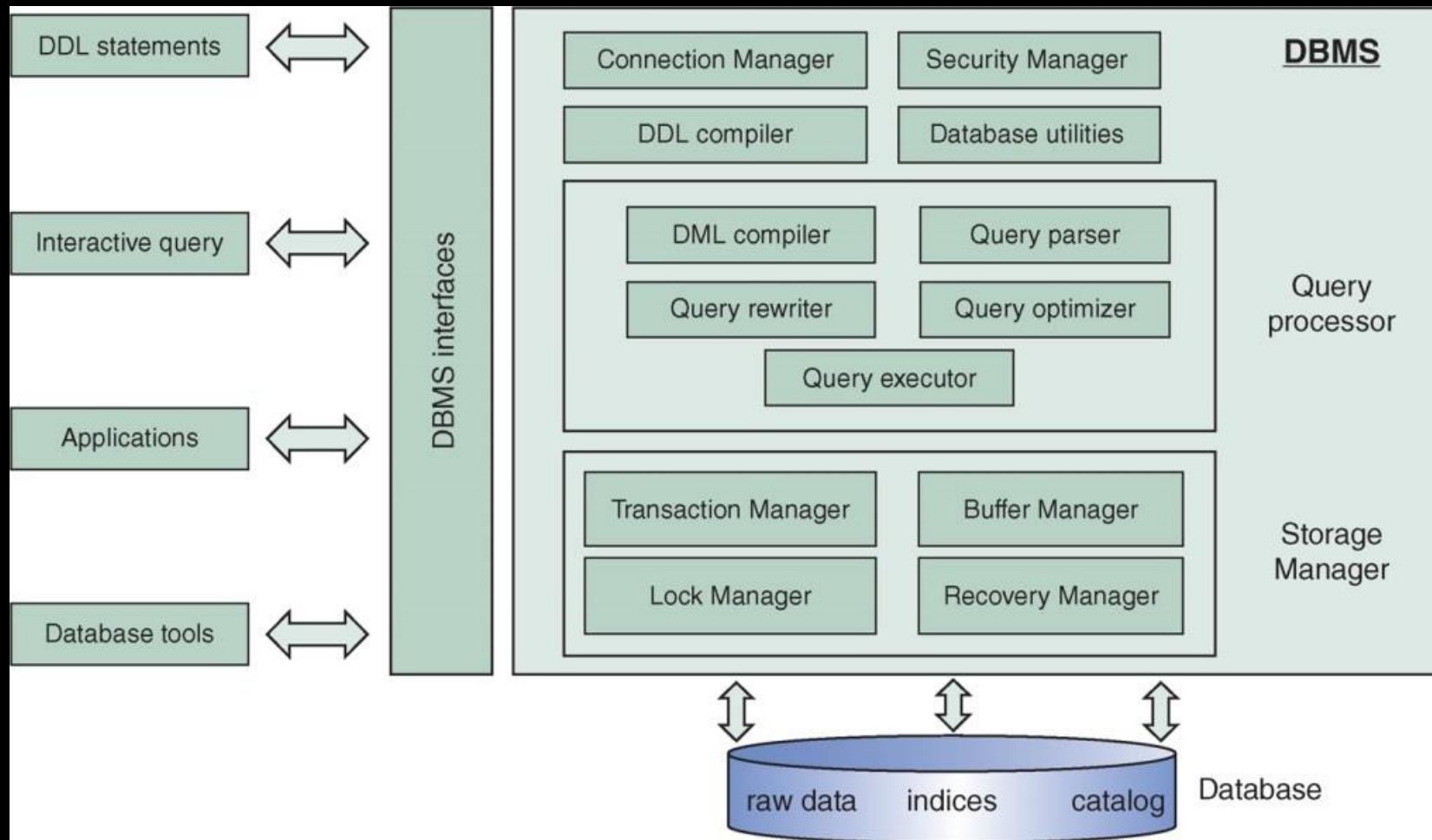
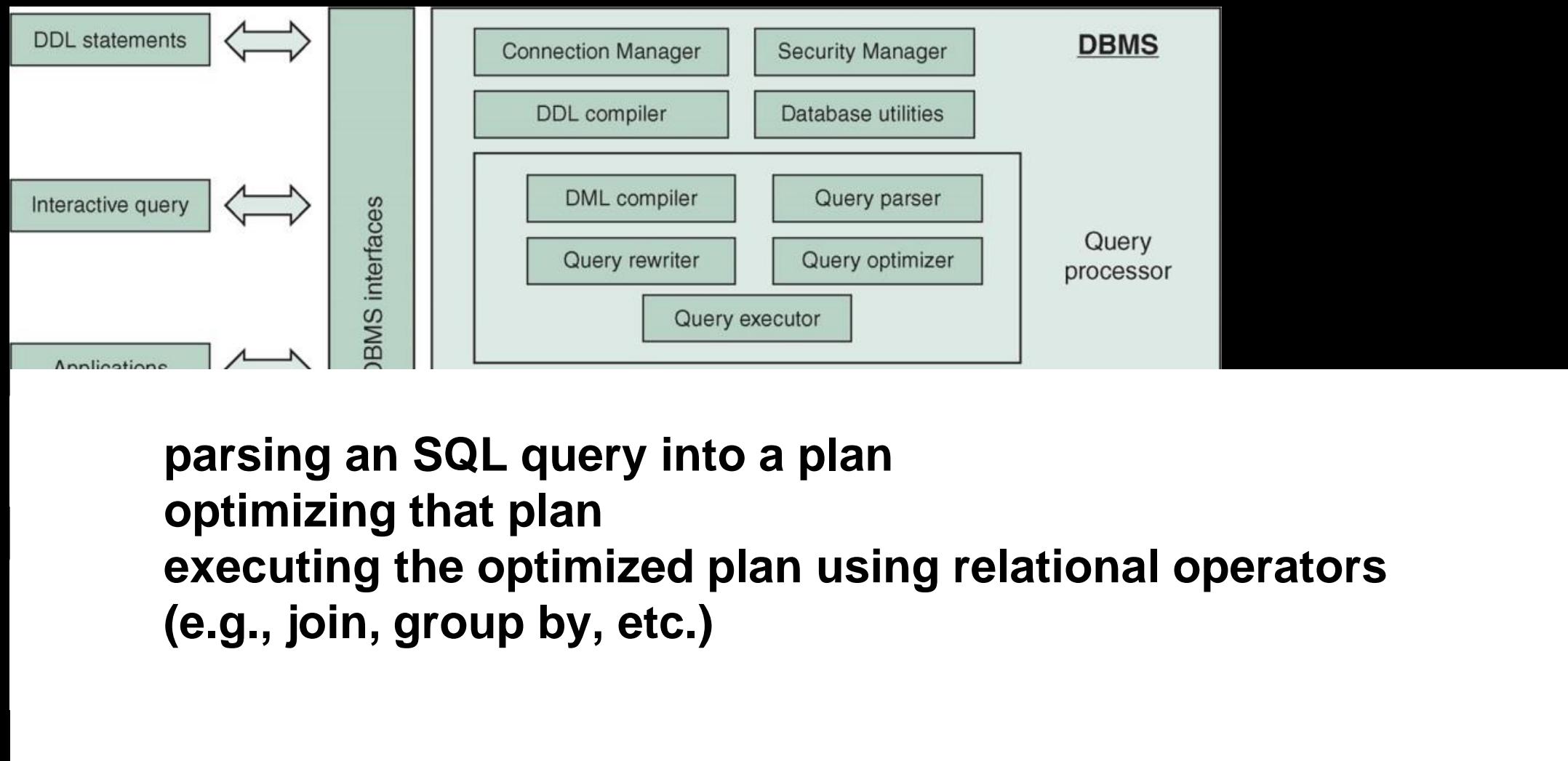


Figure 2.1

# Query Processor

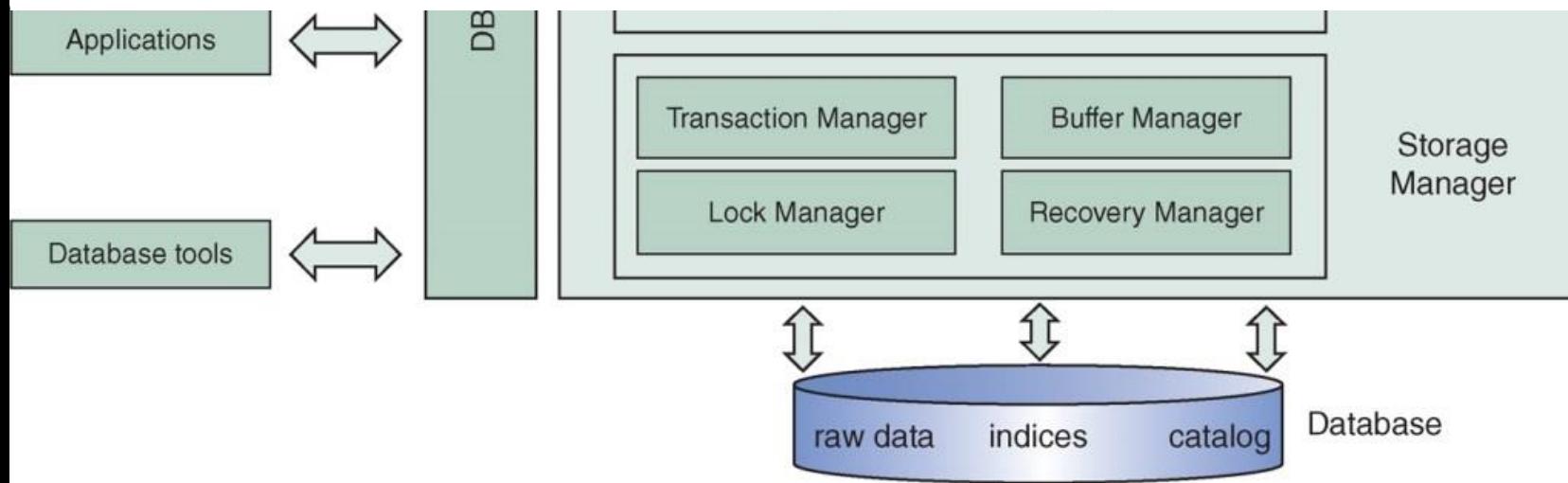


**parsing an SQL query into a plan  
optimizing that plan  
executing the optimized plan using relational operators  
(e.g., join, group by, etc.)**

# Storage Manager

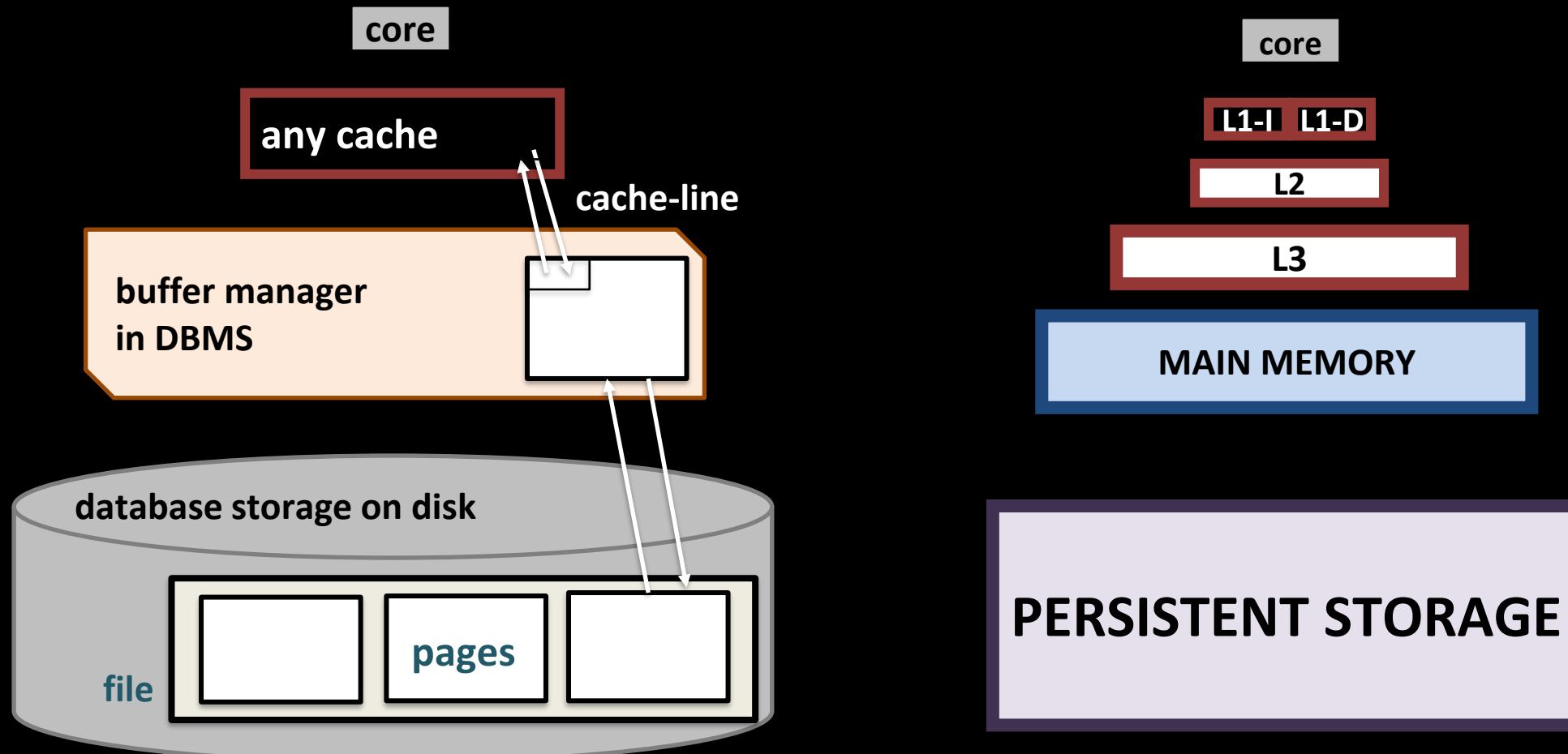
manages & keeps track of the data read from persistent storage into main memory:

- better optimizations because of application knowledge, e.g., replacement policy, prefetching
- you may want to force-flush some data to disk (e.g., log)
- you may not want to ever flush some data (e.g., aborted requests)



# Data Movement through Storage Hierarchy

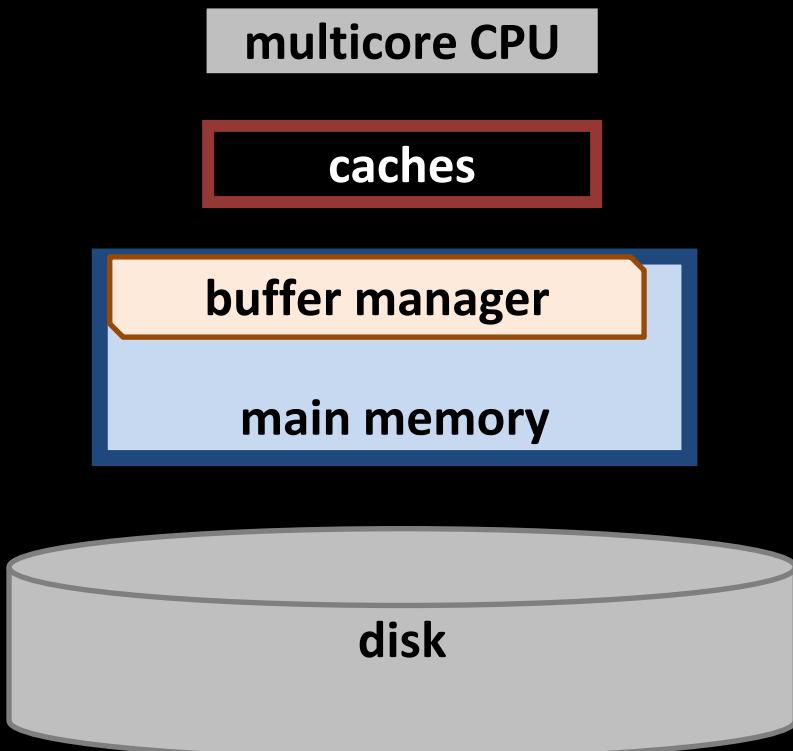
for a traditional database system



# What if data fits in-memory?

---

## Traditional System



## Main-Memory System

- query compilation that generates more efficient code
- no buffer manager
- data organized for better cache utilization/accesses
- no disk use during query execution
- lightweight logging for recovery

# Main-Memory DBMSs

---

- **What if all data is in RAM?**
  - No disk while executing tasks → never need to wait for disk
  - *Note:* There is still disk for persistent storage!
- **Don't have to optimize data accesses for disk**
  - no buffer manager
  - no need for tuples in slotted-pages in-memory,  
have direct access to tuples
  - indexes aren't kept in pages either & usually aren't persisted
  - aim better cache utilization/accesses
    - row-store for OLTP, column-store for OLAP
    - cache-conscious index structures
    - query compilation that generates cache-conscious code

} also  
valid  
for  
disk

# Main-Memory DBMSs: Downsides

---

- **Startups/restarts are expensive**
  - need to load everything in-memory
  - indexes has to be rebuilt
  - recovery takes longer if not done carefully
- **There comes a day where your data doesn't fit in memory anymore**
  - most main-memory systems added support for efficiently accessing cold/old data from disk later  
→ because customers ask for it

# Take Away

- It is important to choose the right storage model for the target workload:
  - OLTP = Row Store
  - OLAP = Column Store
- Goal: maximize sequential access, minimize unnecessary data reads.
- Main-memory DBMS
  - No blocking for disk I/O
  - No buffer management/buffer pool
  - Optimizing for cache accesses
  - Non-blocking concurrency, lightweight logging

# Introduction to Database Systems

## IDBS - Autumn 2024

### Lecture 10 - Transactions in RDBMs

---

ACID Properties

Logging

Locking

---

Readings: PDBM 14

Omar Shahbaz Khan

# Wake Up Task!

- Course Evaluation on LearnIT
- Take 15 min.
- What do you like about the course and what can be done better in the future?

# -- TODO

- **Transactions**
  - ACID Properties
  - Buffer Management
  - Logging
  - Recovery
  - Locking

# -- TODO

- **Transactions**
  - ACID Properties
  - Buffer Management
  - Logging
  - Recovery
  - Locking

# Transactions

Consider the following transactions on the relation accounts(no, balance, type):

Transaction A

```
UPDATE accounts SET balance=balance*1.02 WHERE type='savings';
UPDATE accounts SET balance=balance*1.01 WHERE type='salary' AND balance > 0;
UPDATE accounts SET balance=balance*1.07 WHERE type='salary' AND balance < 0;
```

Transaction B

```
UPDATE accounts SET type='salary' WHERE no=12345;
```

Transaction C

```
UPDATE accounts SET balance=balance-1000 WHERE no=12345;
```

- Assume that account 12345 starts as 'savings' with balance=500.
- What are the possible balance values after running transactions A, B and C?

Consider the following transactions on the relation accounts(no, balance, type):

Transaction A

```
UPDATE accounts SET balance=balance*1.02 WHERE type='savings';
UPDATE accounts SET balance=balance*1.01 WHERE type='salary' AND balance > 0;
UPDATE accounts SET balance=balance*1.07 WHERE type='salary' AND balance < 0;
```

Transaction B

```
UPDATE accounts SET type='salary' WHERE no=12345;
```

Transaction C

```
UPDATE accounts SET balance=balance-1000 WHERE no=12345;
```

Assume that account 12345 starts as 'savings' with balance=500

A→B→C: -490

A→C→B: -490

B→A→C: -495

B→C→A: -535

C→A→B: -510

C→B→A: -535

Consider the following transactions on the relation accounts(no, balance, type):

Transaction A

```
UPDATE accounts SET balance=balance*1.02 WHERE type='savings';
UPDATE accounts SET balance=balance*1.01 WHERE type='salary' AND balance > 0;
UPDATE accounts SET balance=balance*1.07 WHERE type='salary' AND balance < 0;
```

Transaction B

```
UPDATE accounts SET type='salary' WHERE no=12345;
```

Transaction C

```
UPDATE accounts SET balance=balance-1000 WHERE no=12345;
```

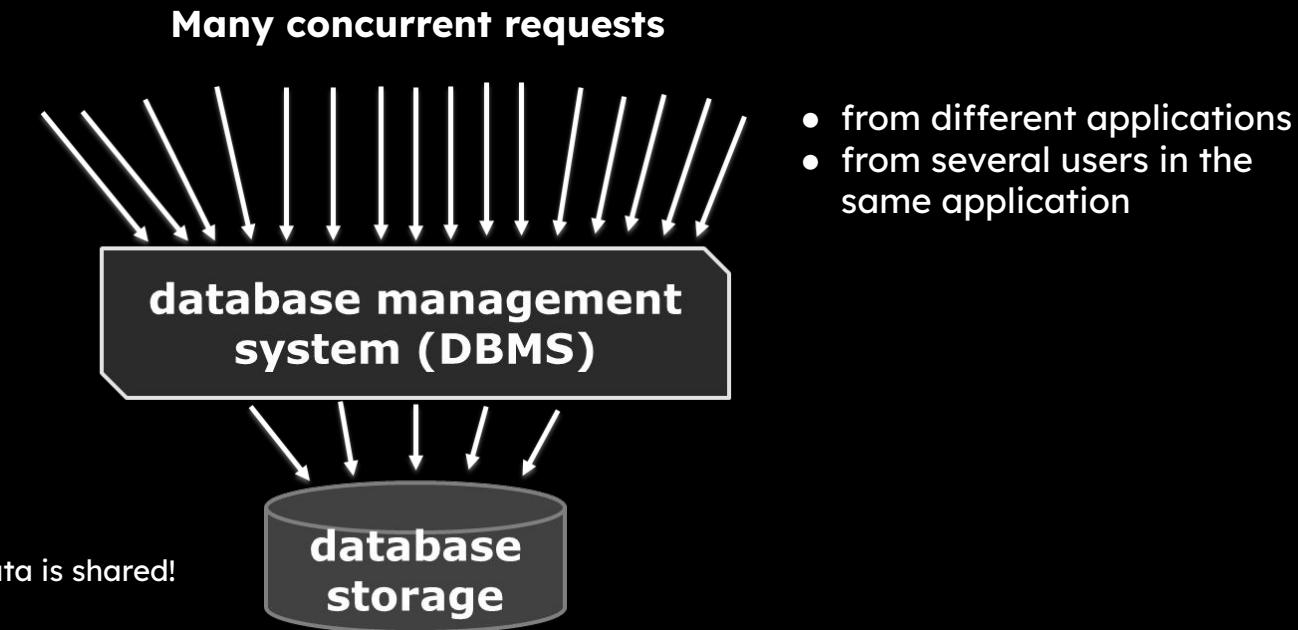
Assume that account 12345 starts as 'savings' with balance=500 and they are interleaving

- A.1: 510
- B: 510 (salary)
- A.2: 515.10
- C: -484.90
- A.3: -518.843

# Why do we need Transactions?

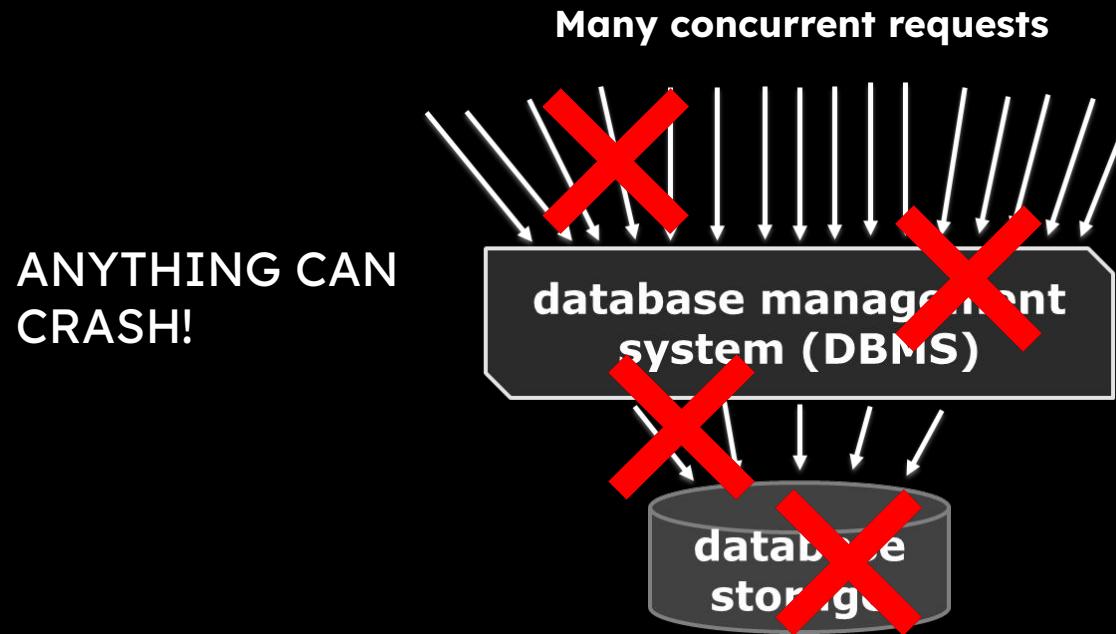
- Transactions group a set of database operations
  - Atomic, all operations succeed or none do
- May require shared database resources (rows, columns, tables)
  - Can lead to conflicts with multiple concurrent transactions
- Can be run sequentially or concurrently
  - Trade-off between consistency (data integrity) and performance (latency)

# Why do we need Transactions?



**DBMS must ensure reliable operations over shared data  
despite many concurrent accesses**

# Why do we need Transactions?



**DBMS must ensure reliable operations over shared data  
despite many concurrent accesses**

To the end user everything is fine,  
thanks to transactions



Recovery

Constraints, Triggers

Concurrency  
Control

Recovery

# A<sub>TOMICITY</sub>

A Transaction is  
“one operation”

# C<sub>ONSISTENCY</sub>

Each Transaction  
moves the DB from one  
consistent state to  
another

# I<sub>SOLATION</sub>

Each Transaction  
is alone in the  
world

# D<sub>URABILITY</sub>

Persistence of  
successful  
transactions even  
through system failure

# ACID Properties of Transactions

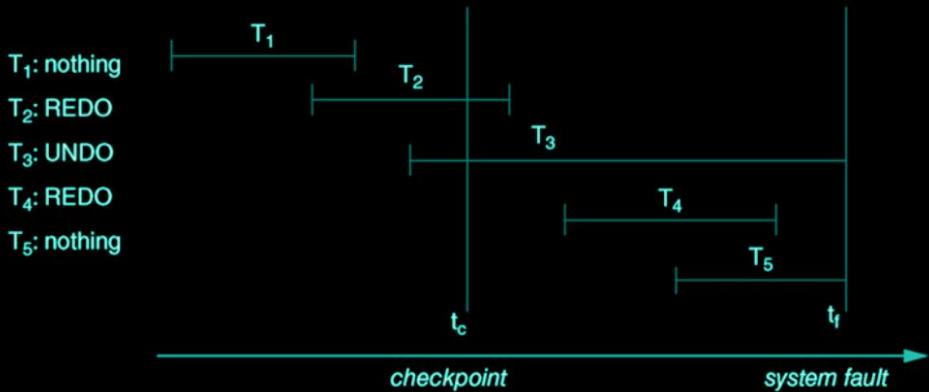
- **Atomicity:** Each transaction runs to completion or has no effect at all.
- **Consistency:** After a transaction completes, the integrity constraints are satisfied.
- **Isolation:** Transactions executed in parallel have the same effect as if they were executed sequentially.
- **Durability:** The effect of a committed transaction remains in the database even if the system crashes.

# How to Implement Transactions?

- Consistency ~= satisfying constraints
  - Use indexes for primary and foreign key, triggers, ...
- Atomicity and Durability = tracking changes
  - Logging “before” values to UNDO changes
  - Logging “after” values to REDO changes
- Isolation = preventing corrupting changes
  - Pessimistic: Locking to prevent conflicts
  - Optimistic: Time stamps to detect conflicts

# Atomicity and Durability Issues

- Atomicity
  - Transactions abort
  - Systems crash
- Durability
  - Systems crash
- Upon restart
  - Want to see effects of T1, T2, T4
  - Want to remove the effects of T3, T5



# Buffer Management (RAM v Disk)

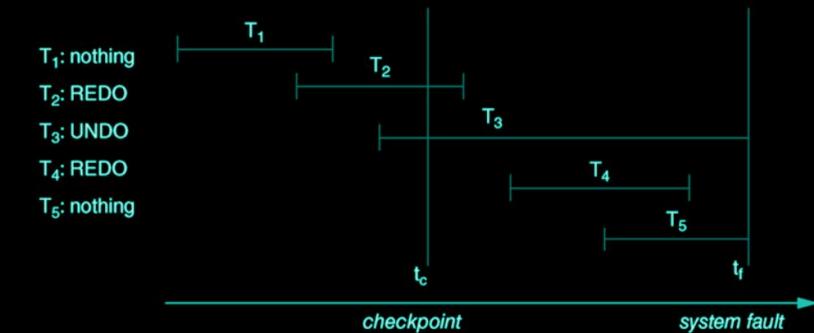
- Once a transaction completes, it performs a COMMIT
- A COMMIT ≠ changes are on disk
- Buffer Management Policies determine when changes are moved from RAM to disk
  - The transaction log is always updated / written to disk
  - FORCE / NO FORCE policies affect data pages of committed transactions
  - STEAL / NO STEAL policies affect data pages of uncommitted transactions

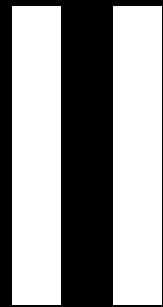
# FORCE / NO STEAL

- **FORCE** = write changed pages to disk at COMMIT
  - Book: “immediate update” policy
  - Ensures Durability (assuming writes are atomic)
  - Increases response time
  - Will we FORCE changes to disk at COMMIT? NO
- **NO STEAL** = allow updated pages to be replaced
  - Book: NO STEAL = “deferred update” policy
  - Ensures Atomicity (can simply discard at abort)
  - Increases response time
  - Will we guarantee NO STEAL of dirty pages? NO

# NO FORCE / STEAL

- NO FORCE: Changes in RAM after COMMIT
  - What if system crashes?
  - Need to remember the **new** value of P to be able to **REDO** the changes
- STEAL: Changes to disk before COMMIT
  - What if transaction aborts? system crashes?
  - Need to remember the **old** value of P to be able to **UNDO** the changes





# Write-Ahead Logging (WAL Protocol)

- Write-Ahead Logging
  1. Before any changes are written to disk  
we force the corresponding log record to disk
  2. Before a transaction is committed  
we force all log records for the transaction to disk
- #1 ensures Atomicity
- #2 ensures Durability

# Key Concept: The Log

- Write REDO and UNDO info to log
  - Ordered list of REDO/UNDO info
  - Think of an infinite file with append only!
- Log processing must be fast – why? and how?
  - Write minimal info to log (diff)
    - <xid, pageID, offset, length, old\_data, new\_data> + control info
    - Many log entries per page
  - Ensure sequential writes!
    - Writing a log entry  $\diamond$  writing all previous entries
    - **Put log on its own disk!**

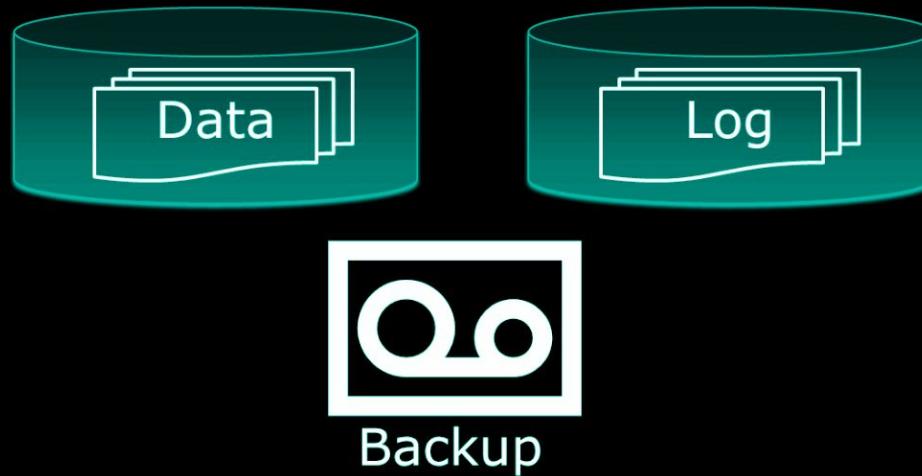
# Transactions in MMDBMSs

- ACI...D
  - Are we ok with losing all data?
  - How do we make changes persistent?
    - Transaction Log on Disk
    - Snapshots on Disk (Copy of the database state)

# Restart Recovery

1. Analyze information about transactions from the last checkpoint
2. REDO the changes of committed transactions that did not make it to disk
3. UNDO the changes of uncommitted transactions that accidentally made it to disk

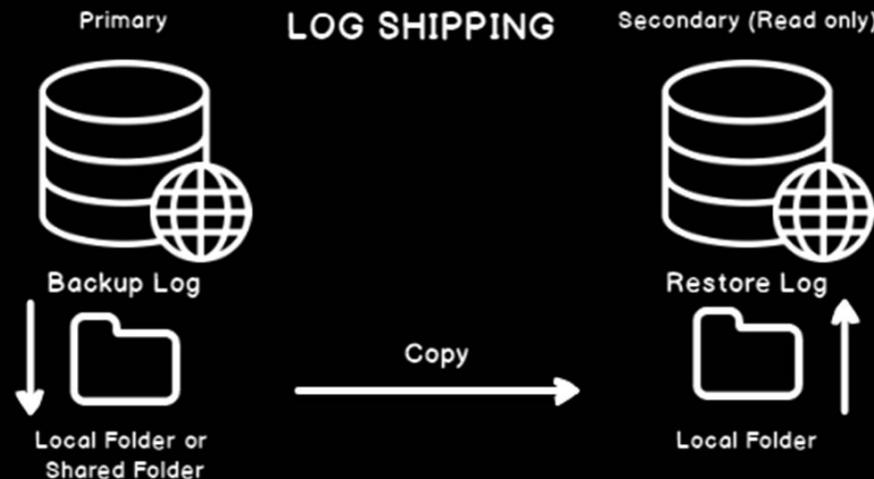
# Database Setup for DBAs



- On crash:
  - Data + Log = Recovery succeeds
  - Backup + Log = Recovery succeeds
  - Data + Backup (No Log) = Recovery FAILS!

# High Availability

- Typical approach: Second (failover) server
  - Takes over in case the primary fails
- Transaction log used to keep it up to date



# How to Implement Transactions?

- Consistency ~= satisfying constraints
  - Use indexes for primary and foreign keys, triggers, ...
- Atomicity and Durability = tracking changes
  - Logging “before” values to undo changes
  - Logging “after” values to redo changes
- Isolation = preventing corrupting changes
  - **Pessimistic: Locking to prevent conflicts**
  - Optimistic: Time stamps to detect conflicts

# Isolation and Serializability

- Want transactions to satisfy *serializability*:
  - The state of the database should always look as if the committed transactions ran in some *serial schedule*
- The scheduler of the DBMS is allowed to choose the order of transactions:
  - It is not necessarily the transaction that is started first, which is first in the serial schedule

# A Simple Scheduler

- A simple scheduler would maintain a queue of transactions and carry them out in order
- Problems:
  - Transactions must *wait* for each other, even if unrelated (e.g. requesting data on different disks)
    - Some transactions may take very long, e.g. when external input or remote data is needed during the transaction
  - Possibly smaller throughput (Why?)

# A Simple Scheduler

- A simple scheduler would maintain a queue of transactions and carry them out in order
- Some believe this is fine for *transaction processing*, especially for MMDBs

## The End of an Architectural Era (It's Time for a Complete Rewrite)

Michael Stonebraker  
Samuel Madden  
Daniel J. Abadi  
Stavros Harizopoulos

Nabil Hachem  
AvantGarde Consulting, LLC  
[nhachem@agdba.com](mailto:nhachem@agdba.com)

Pat Helland  
Microsoft Corporation  
[phelland@microsoft.com](mailto:phelland@microsoft.com)

# Interleaving Schedulers

- Most DBMSs still have schedulers that allow the actions of transactions to interleave
- However, the result should be **as if** some serial schedule was used
  - A non-serial schedule that yields the same outcome as a serial schedule is known as a serializable schedule
- We will now study a mechanism that *enforces* “serializability”: **Locking**
- Other methods exist: Time stamping / optimistic concurrency control
  - Out of scope for this course

# Locks

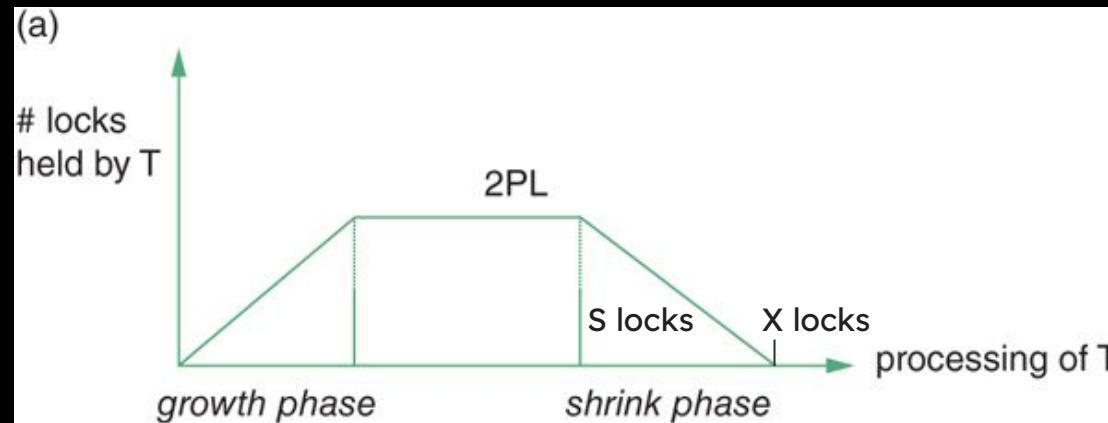
- In its simplest form, a lock is a right to perform operations on a database element
- Only one transaction may hold a lock on an element at any time
- Locks must be requested by transactions and granted by the locking scheduler
- Typically, two types of locks: Read/Shared or Write/Exclusive

# Two-Phase Locking

- **Growing Phase:** Acquire locks
  - When reading a database resource, get a shared (S) lock
  - When writing a database resource, get an exclusive (X) lock
- **Shrinking Phase:** No new locks can be acquired, and start releasing locks
- **Multiple variations:**
  - Static 2PL: All locks acquired before running the transaction
  - Strict 2PL: Release only shared locks during shrinking phase
  - Rigorous 2PL: Release locks at COMMIT/ABORT
- **Commonly implemented, since:**
  - Simple to understand and works well in practice
  - It makes transaction rollback easier to implement
- **But:** Optimistic approaches are gaining in popularity

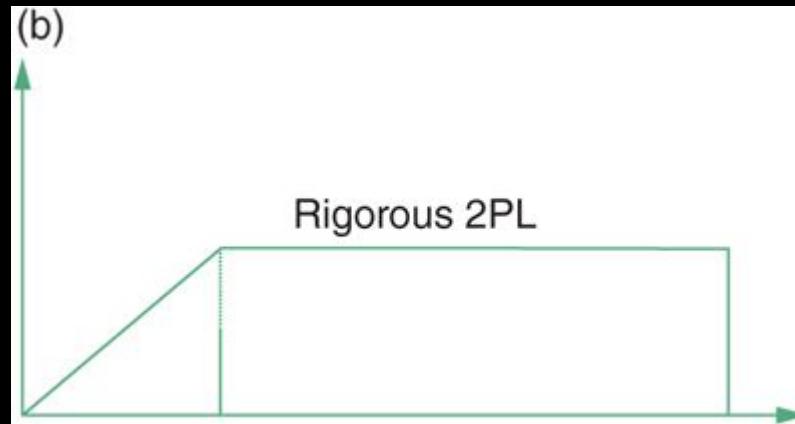
# Strict Two-Phase Locking

- Strict 2PL protocol:
  1. Before reading a record/page, get a shared (S) lock  
Before writing a record/page, get an exclusive (X) lock
  2. A record/page cannot have an X lock at the same time as any other lock
  3. Release S locks during shrinking phase and X locks on COMMIT/ABORT



# Rigorous Two-Phase Locking

- Rigorous 2PL protocol:
  1. Before reading a record/page, get a shared (S) lock  
Before writing a record/page, get an exclusive (X) lock
  2. A record/page cannot have an X lock at the same time as any other lock
  3. Release all locks on COMMIT/ABORT



# Locks and Deadlocks

- The DBMS sometimes must make a transaction wait for another transaction to release a lock.
- This can lead to deadlock if e.g. A waits for B, and B waits for A.
- In general, we have a deadlock exactly when there is a cycle in the waits-for graph.
- Deadlocks are resolved by aborting some transaction involved in the cycle.

# Avoiding Deadlocks

- Upgrade requests can also deadlock

```
SELECT :x=counter  
FROM table  
WHERE <condition>  
:x=:x+1  
  
UPDATE table  
SET counter = :x  
WHERE <condition>
```

```
UPDATE table  
SET counter = counter+1  
WHERE <condition>  
  
SELECT :x=counter  
FROM table  
WHERE <condition>
```

```
SELECT FOR UPDATE :x=counter  
FROM table  
WHERE <condition>  
:x=:x+1  
  
UPDATE table  
SET counter = :x  
WHERE <condition>
```

- Order Matters:
  - With consistent order of access, deadlocks are avoided
  - Why do B+ tree accesses not deadlock? - Same traversal order
  - Optimizer may not allow control over order!

# Phantom Tuples

- Suppose we lock tuples where  $A=1$  in a relation, and subsequently another tuple with  $A=1$  is inserted.
- For some transactions this may result in unserializable behaviour, i.e., it will be clear that the tuple was inserted during the course of a transaction.
- Such tuples are called phantoms.

# Phantom Example

```
CREATE TABLE B (x INTEGER, y INTEGER);
INSERT INTO B VALUES (1, 2);
```

```
BEGIN;
SELECT MIN(y) FROM B WHERE x = 1;
```

```
BEGIN;
INSERT INTO B VALUES (1, 1);
COMMIT;
```

-- Repeat the SAME read!

```
SELECT MIN(y) FROM B WHERE x = 1;
COMMIT;
```

# Avoiding Phantoms

- Phantoms can be avoided by putting an exclusive lock on a relation before adding tuples.
  - However, this leads to poor concurrency.
- A technique called “index locking” can be used to prevent other transactions from inserting phantom tuples but allow most non-phantom insertions.
- In SQL, the programmer may choose to either allow phantoms in a transaction or insist they should not occur.

# Isolation Levels in Modern Systems

- **READ UNCOMMITTED**  
a transaction can read uncommitted changes
- **READ COMMITTED**  
a transaction only reads committed data,  
some other transaction may overwrite this data
- **REPEATABLE READ**  
a transaction only reads committed data,  
other transactions cannot overwrite this data,  
but phantoms are possible
- **SERIALIZABLE**  
ensures serializable schedule with no anomalies

May violate the “I” in ACID

Ensures “I” in ACID

Database	Default Isolation	Maximum Isolation
Actian Ingres 10.0/10S	S	S
Aerospike	RC	RC
Akiban Persistit	SI	SI
Clustrix CLX 4100	RR	?
Greenplum 4.1	RC	S
IBM DB2 10 for z/OS	CS	S
IBM Informix 11.50	Depends	RR
MySQL 5.6	RR	S
MemSQL 1b	RC	RC
MS SQL Server 2012	RC	S
NuoDB	CR	CR
Oracle 11g	RC	SI
Oracle Berkeley DB	S	S
Oracle Berkeley DB JE	RR	S
Postgres 9.2.2	RC	S
SAP HANA	RC	SI
ScaleDB 1.02	RC	RC
VoltDB	S	S
Legend	<i>RC: read committed, RR: repeatable read, S: serializability, SI: snapshot isolation, CS: cursor stability, CR: consistent read</i>	

The entire world doesn't run on  
ACID!

But an important part of it does!

# -- TODO -> DONE

- ✓ Transactions
- ✓ ACID Properties
- ✓ Buffer Management
- ✓ Logging
- ✓ Recovery
- ✓ Locking

# Takeaways

You're banished to  
the Phantom Zone

## ACID:

- **Atomicity:**
  - All or nothing
  - COMMIT/ABORT/ROLLBACK
  - Write Ahead Logging (WAL)
- **Consistency:**
  - Constraints
  -
- **Isolation:**
  - Locking (Pessimistic)
  - Isolation levels
- **Durability:**
  - Write Ahead Logging (WAL)
  - Backups and recovery protocols

# Next Time in IDBS...

# Introduction to Database Systems

## IDBS - Autumn 2024

### Lecture 11 - Scale and Cloud

---

Scaling-Out

NoSQL

Eventual Consistency

CAP Theorem

---

Readings: PDBM 11

Omar Shahbaz Khan

# Introduction to Database Systems

## IDBS - Spring 2024

### Lecture 11 - Scale & NoSQL

---

Scaling Up/Out  
NoSQL  
Eventual Consistency  
CAP Theorem

---

Readings: PDBM 11

Omar Shahbaz Khan

# General Info

## HOMEWORK 4 - OUT NOW!

- Deadline: December 2, 2024 23:59
- Remember: 3/4 Homework Assignments need to be passed to be eligible for the exam!

## FEEDBACK FOR HOMEWORK 3

- Will be out ASAP

# Last Time in IDBS...

# -- TODO -> DONE

- ✓ Course Evaluation!
- ✓ Transactions
  - ✓ ACID Properties
  - ✓ Buffer Management
  - ✓ Logging
  - ✓ Recovery
  - ✓ Locking

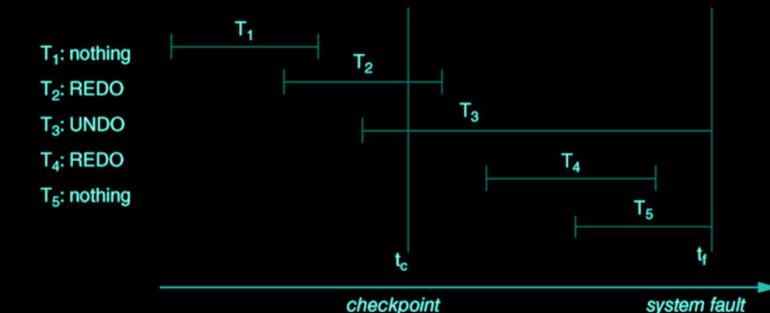
# Transactions

- Atomicity & Durability
  - Logging
  - Recovery
- Consistency
  - Constraints
  - Triggers
- Isolation
  - Locking
  - Isolation Levels

# Transactions

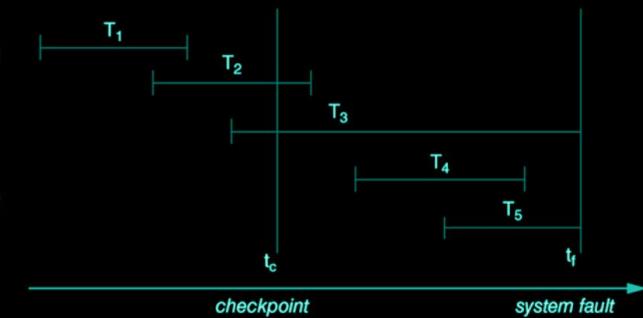
- Atomicity & Durability
  - Logging
  - Recovery
- Consistency
  - Constraints
  - Triggers
- Isolation
  - Locking
  - Isolation Levels

- Write-Ahead-Logging (WAL)
  1. Before any changes are written to disk we force the corresponding log record to disk
    - Is this necessary for MMDBs?
  2. Before a transaction is committed we force all log records for the transaction to disk



# Transactions

- Atomicity & Durability
    - Logging
    - Recovery
  - Consistency
    - Constraints
    - Triggers
  - Isolation
    - Locking
    - Isolation Levels
  - Recovery
    - 1. Analyze Log
    - 2. REDO Committed Transactions
    - 3. UNDO Uncommitted Transactions
- Is this necessary for MMDBs?



# Transactions

- Atomicity & Durability
  - Logging
  - Recovery
- Consistency
  - Constraints
  - Triggers
- Isolation
  - Locking
  - Isolation Levels
- Constraints
  - Primary Key, Foreign Key
  - NOT NULL, UNIQUE, CHECK<sup>1</sup>
  - IDENTITY<sup>2</sup>, SERIAL
  - In DDL
- Triggers
  - Functions
  - Complex Requirements
  - Beyond DDL

1: <https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-check-constraint/>

2: <https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-identity-column/>

# Transactions

- Atomicity & Durability
  - Logging
  - Recovery
- Consistency
  - Constraints
  - Triggers
- Isolation
  - Locking
  - Isolation Levels
- Serializability
- Two-Phase Locking
  - Growing Phase: Acquire Locks
  - Shrinking Phase: Release Locks
  - S2PL: Release only Shared Locks during shrinking phase
  - R2PL: Release locks at COMMIT/ABORT/ROLLBACK
- Deadlocks
  - Detected through wait-for-graphs
  - Resolved by picking a victim aka. aborting one of the transactions
- Phantoms

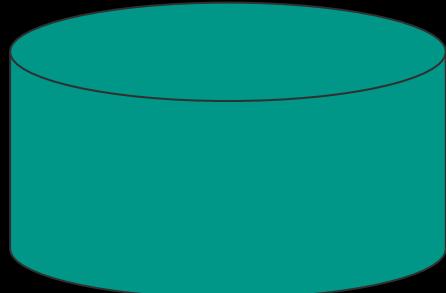
# Transactions

- Atomicity & Durability
  - Logging
  - Recovery
- Consistency
  - Constraints
  - Triggers
- Isolation
  - Locking
  - Isolation Levels
- Isolation Levels
  - Read Uncommitted (RU)
  - Read Committed (RC)
  - Repeatable Read (RR)
  - Serializable (S)
  - Snapshots (Si)

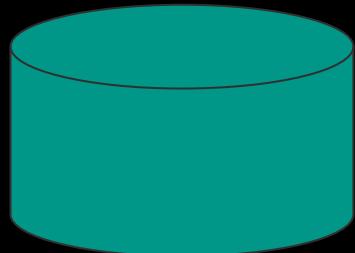
# This Time...

# -- TODO

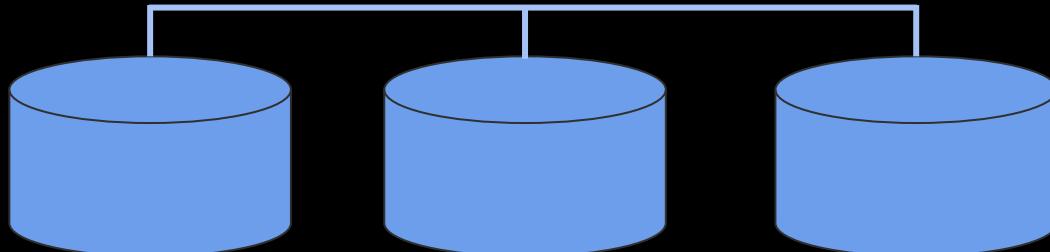
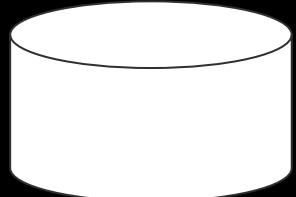
- Scaling Up/Out
- NoSQL
  - Data Model
  - Distributed Architecture
  - Consistency/Availability trade-off
    - Eventual Consistency
    - CAP Theorem



**SCALING UP**



**SCALING OUT**



# The End of an Architectural Era (It's Time for a Complete Rewrite)

Michael Stonebraker

Samuel Madden

Daniel J. Abadi

Stavros Harizopoulos

MIT CSAIL

{stonebraker, madden, dna,  
stavros}@csail.mit.edu

Nabil Hachem

AvantGarde Consulting, LLC

nhachem@agdba.com

Pat Helland

Microsoft Corporation

phelland@microsoft.com

## ABSTRACT

In previous papers [SC05, SBC+07], some of us predicted the end of “one size fits all” as a commercial relational DBMS paradigm. These papers presented reasons and experimental evidence that showed that the major RDBMS vendors can be outperformed by 1-2 orders of magnitude by specialized engines in the data warehouse, stream processing, text, and scientific database markets.

Assuming that specialized engines dominate these markets over time, the current relational DBMS code lines will be left with the business data processing (OLTP) market and hybrid markets where more than one kind of capability is required. In this paper we show that current RDBMSs can be beaten by nearly two orders of magnitude in the OLTP market as well. The experimental evidence comes from comparing a new OLTP prototype, H-Store, which we have built at M.I.T., to a popular RDBMS on the standard transactional benchmark, TPC-C.

We conclude that the current RDBMS code lines, while attempting to be a “one size fits all” solution, in fact, excel at nothing. Hence, they are 25 year old legacy code lines that should be retired in favor of a collection of “from scratch” specialized engines. The DBMS vendors (and the research community) should start with a clean sheet of paper and design systems for tomorrow’s requirements, not continue to push code lines and architectures designed for yesterday’s needs.

All three systems were architected more than 25 years ago, when hardware characteristics were much different than today. Processors are thousands of times faster and memories are thousands of times larger. Disk volumes have increased enormously, making it possible to keep essentially everything, if one chooses to. However, the bandwidth between disk and main memory has increased much more slowly. One would expect this relentless pace of technology to have changed the architecture of database systems dramatically over the last quarter of a century, but surprisingly the architecture of most DBMSs is essentially identical to that of System R.

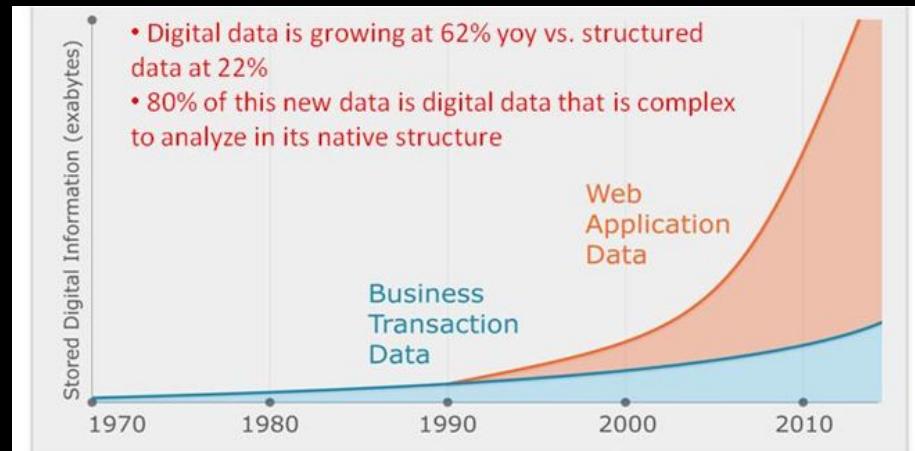
Moreover, at the time relational DBMSs were conceived, there was only a single DBMS market, business data processing. In the last 25 years, a number of other markets have evolved, including data warehouses, text management, and stream processing. These markets have very different requirements than business data processing.

Lastly, the main user interface device at the time RDBMSs were architected was the dumb terminal, and vendors imagined operators inputting queries through an interactive terminal prompt. Now it is a powerful personal computer connected to the World Wide Web. Web sites that use OLTP DBMSs rarely run interactive transactions or present users with direct SQL interfaces.

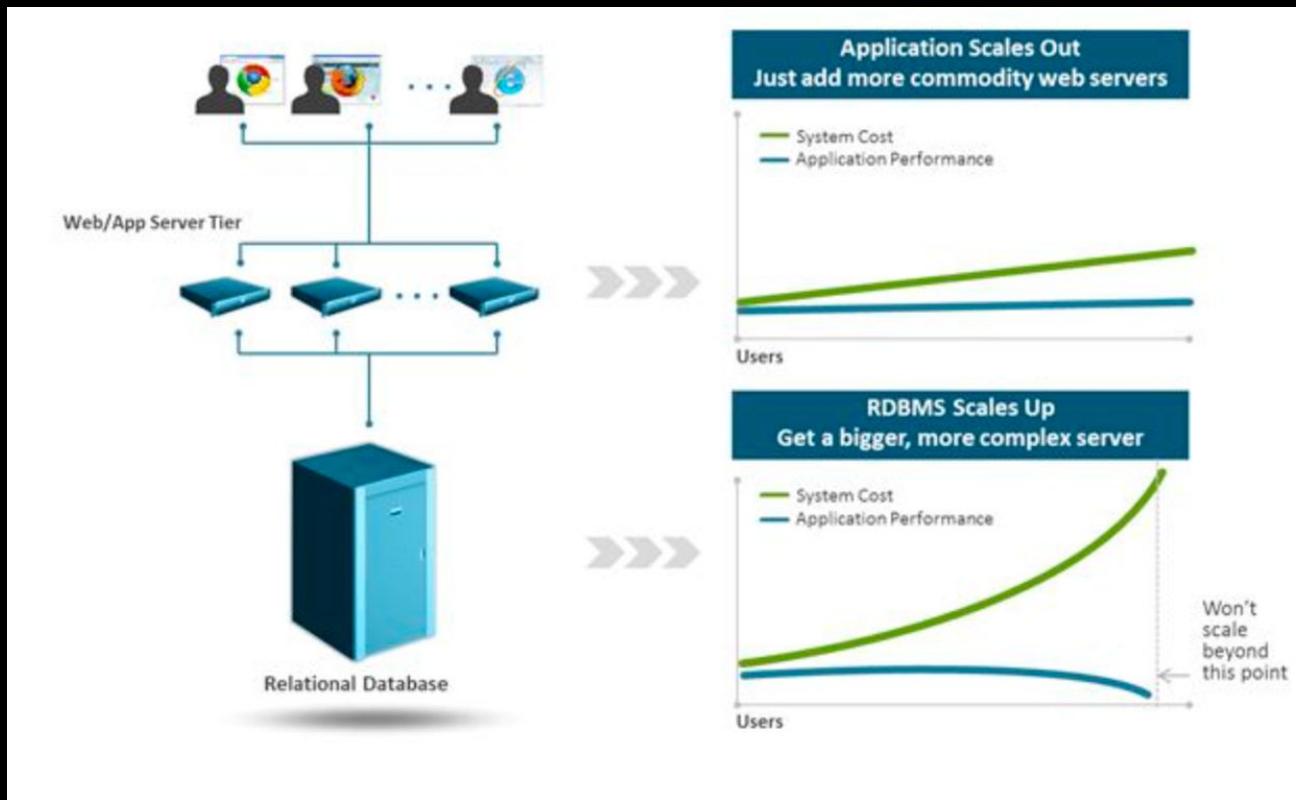
In summary, the current RDBMSs were architected for the business data processing market in a time of different user

# Major Trends

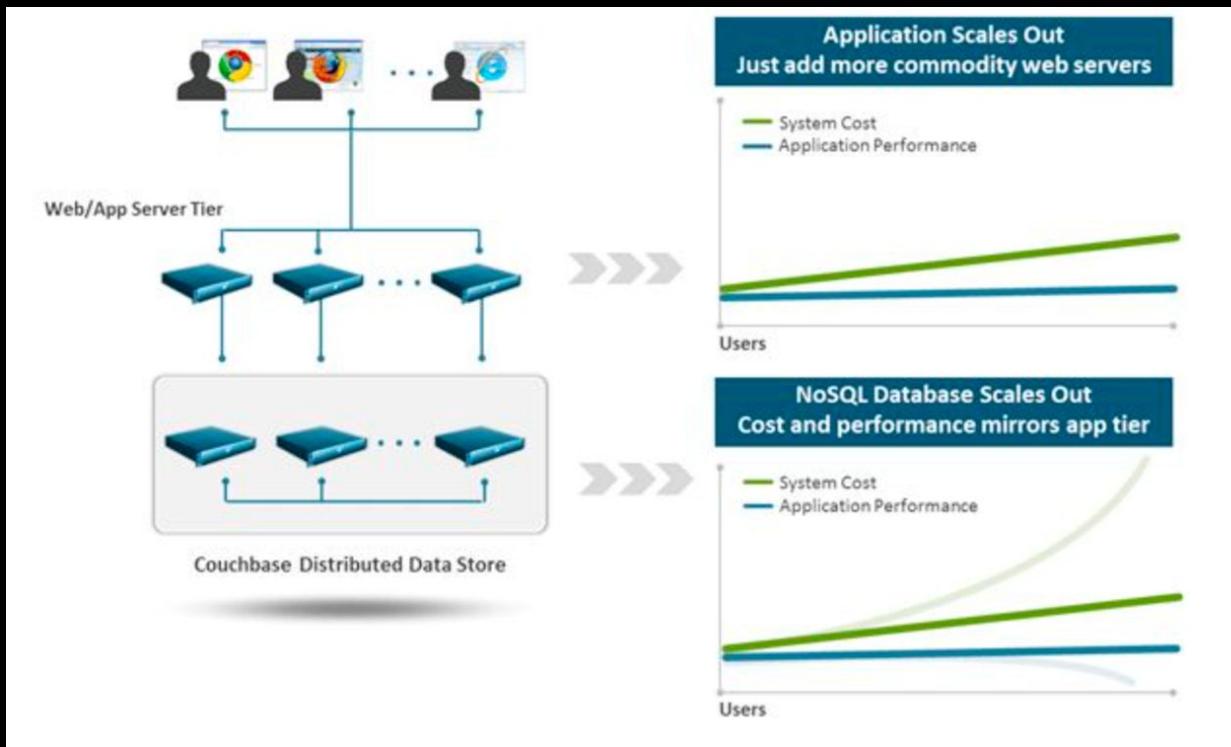
- More hardware
  - Better servers
  - More servers → Cloud
- More data
  - More quantity of data
  - More types of data
  - Still want fast systems
  - Less structure → less need for complexity

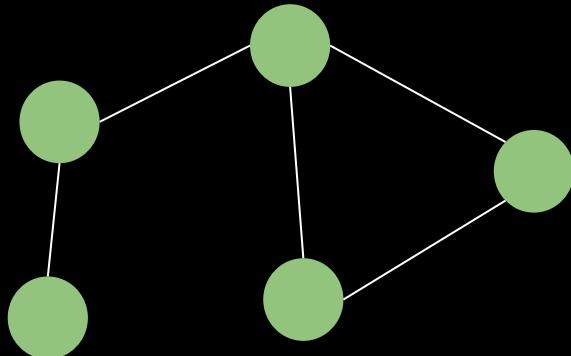
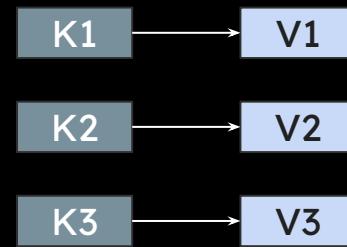
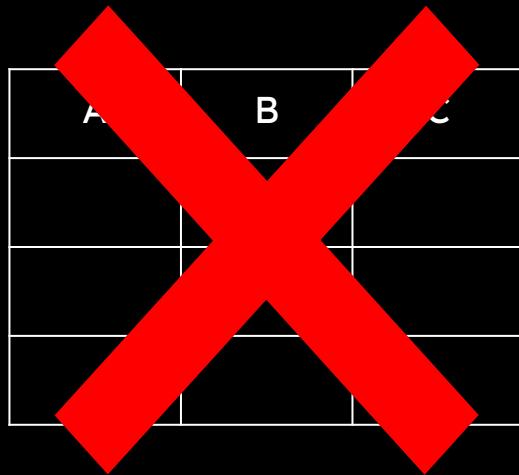


# Scaling UP



# Scaling OUT



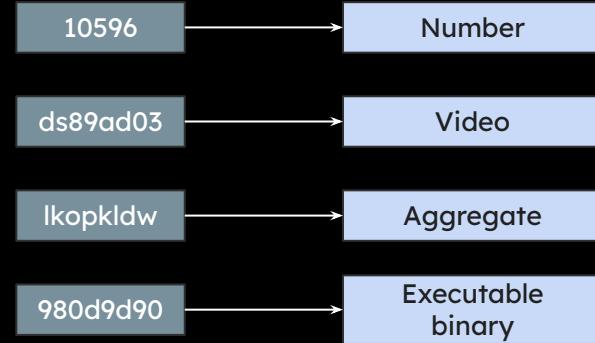


# NoSQL in a Nutshell

- Data model
  - Not relational
  - No formally described schema
- Interface
  - Not only SQL (→ NoSQL name)
  - Proprietary, REST, etc.
- Architecture
  - Usually distributed
- Mostly not ACID compliant
  - Consistency/Availability trade-off (**CAP theorem**)
- Mostly open source

# Key-Value Stores

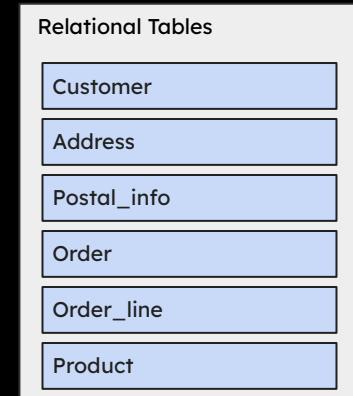
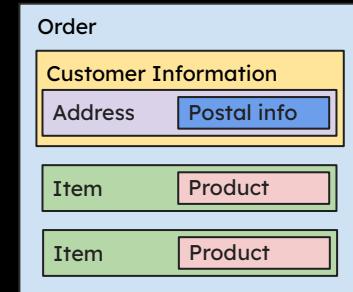
- Associative Array
  - Unique key points to a value
  - Value contents unknown
- Can not be queried
  - GET / PUT only
  - Value can be an aggregate structure
- Examples
  - Riak, Apache Cassandra, Redis



# Document Stores

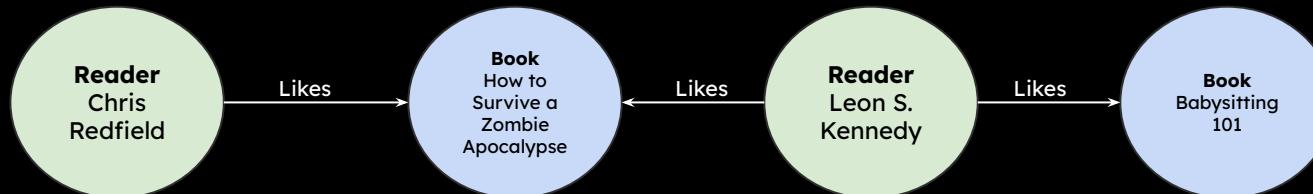
- Each value is a document
  - Most often JSON
  - Unique keys used for retrieval
- You can query into the document
  - More transparent than key-value stores
- The document is an aggregate structure
- Examples: MongoDB, Couchbase, Amazon DocumentDB

```
{  
  "order_id": 3294,  
  "customer": {  
    "ssn": "123456789",  
    "name": "John Doe",  
    "address": "123 Random Street",  
    "postal_code": "98765"  
  },  
  "line-items": [  
    {"product": "Sofa", "price": 2500},  
    {"product": "Table", "price": 1000},  
  ]  
}
```



# Graph Stores

- Nodes = Entities
- Edges = Relationships, directional
- Properties = Entity descriptors
- Examples - neo4j, Allegro, InfiniGraph, OrientDB



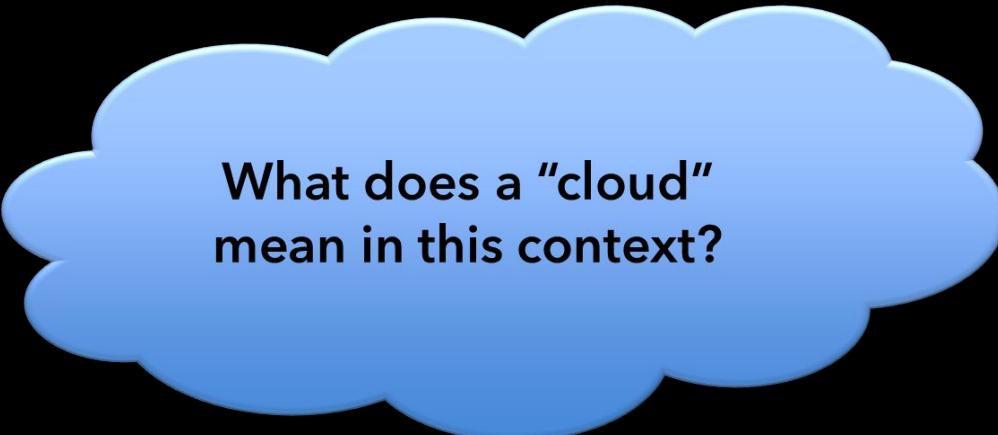
# -- TODO

✓ Scaling Up/Out

- NoSQL
  - ✓ Data Model
    - Distributed Architecture
    - Consistency/Availability trade-off
      - Eventual Consistency
      - CAP Theorem

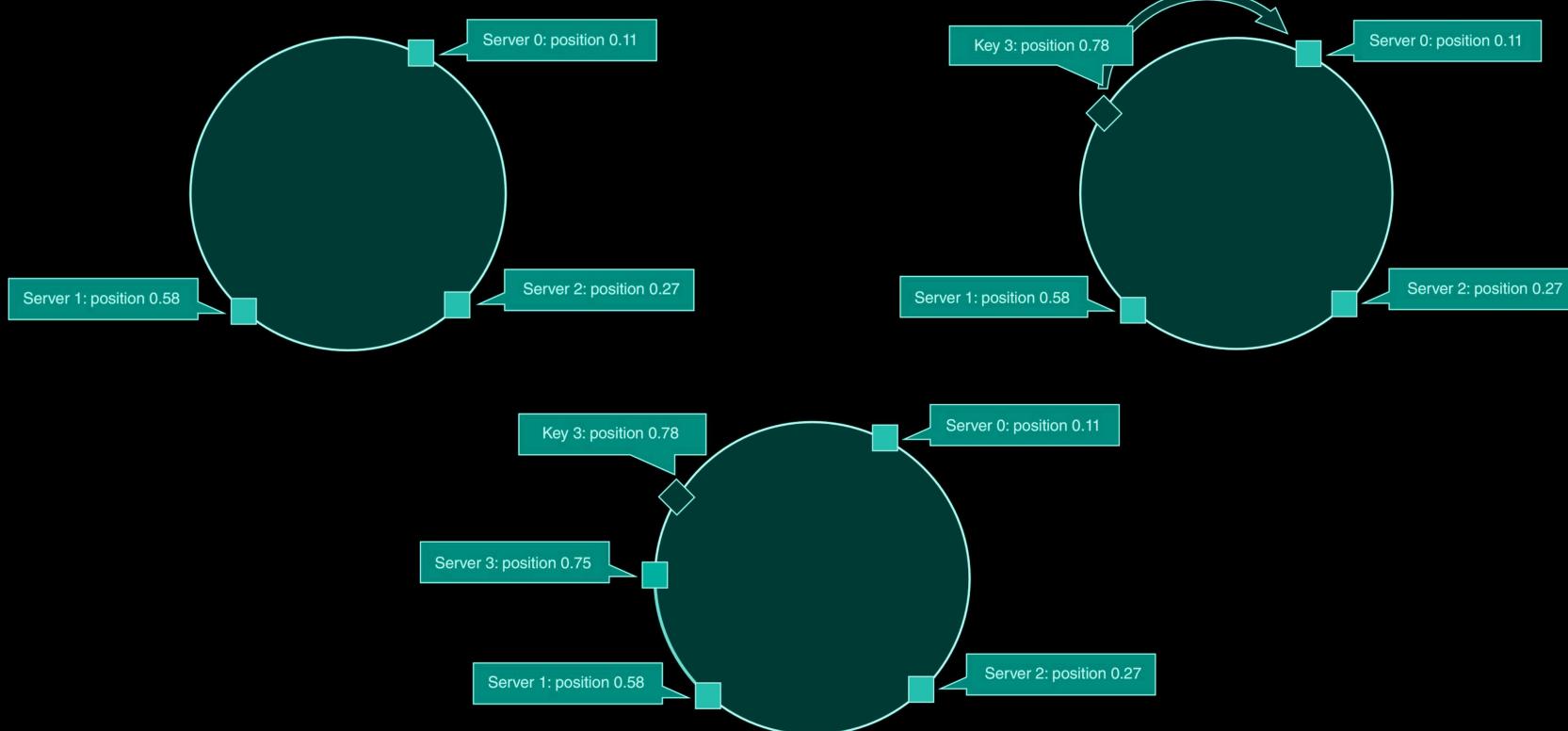
# Goals of a Distributed Storage

- Workload sharing – balance
- Redundancy (replicas) – failure handling



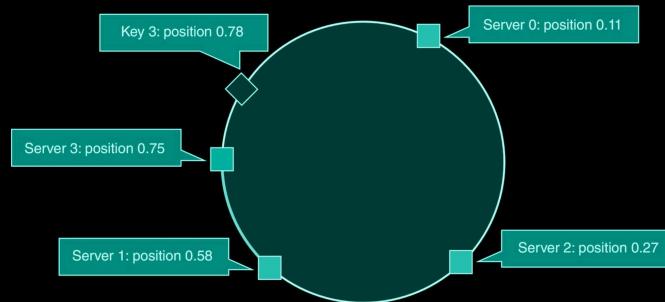
What does a “cloud” mean in this context?

# Consistent Hashing: Balance



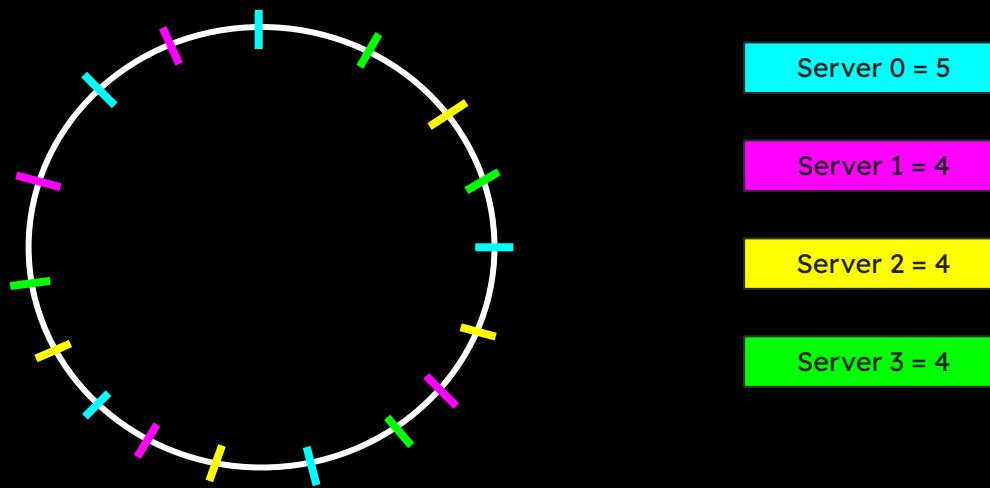
# Consistent Hashing: Problems

- Random assignments can lead to skewed distributions
- Does not reflect a server's capabilities
  - Not all servers have the same hardware
- Adding new server only takes high load from 1 server

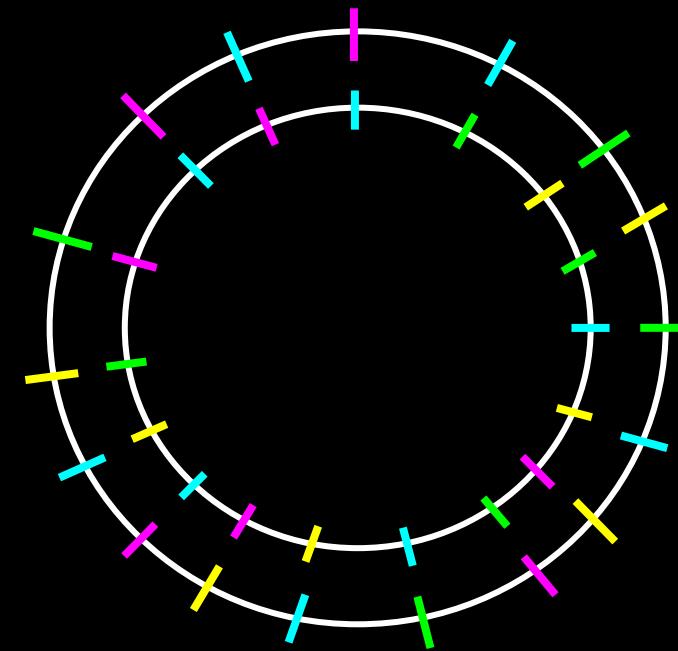


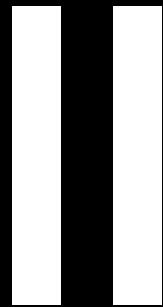
# Consistent Hashing: Virtual Servers

- 1 Server = multiple smaller (virtual) servers
- Random assignments of virtual servers



# Consistent Hashing: Redundancy





# -- TODO

## ✓ Scaling Up/Out

- NoSQL
  - ✓ Data Model
  - ✓ Distributed Architecture
    - Consistency/Availability trade-off
      - Eventual Consistency
      - CAP Theorem

# Replica Consistency

- Sequential (or strong) consistency: All updates are seen by all processes in the same order. As a result, the effects of an update are seen by all observers. There is no inconsistency.
  - Roughly the same as Isolation in ACID
- Weak consistency: Observers might see inconsistencies among replicas
- Eventual consistency: A form of weak consistency, where at some point, in case there is no failure, all replicas will reflect the last update.

# Tunable Consistency

- Not a binary system
  - N replicas, R read quorum, W write quorum
- $R = W = 1$  gives eventual consistency
- $R + W > N$  gives strong consistency
- Example:

If the replication factor is 3, then the consistency level of the reads and writes combined must be at least 4 to achieve strong consistency.  
As in, read operations use 2 out of 3 replicas to verify the value and write operations use 2 out of 3 replicas to verify the value.

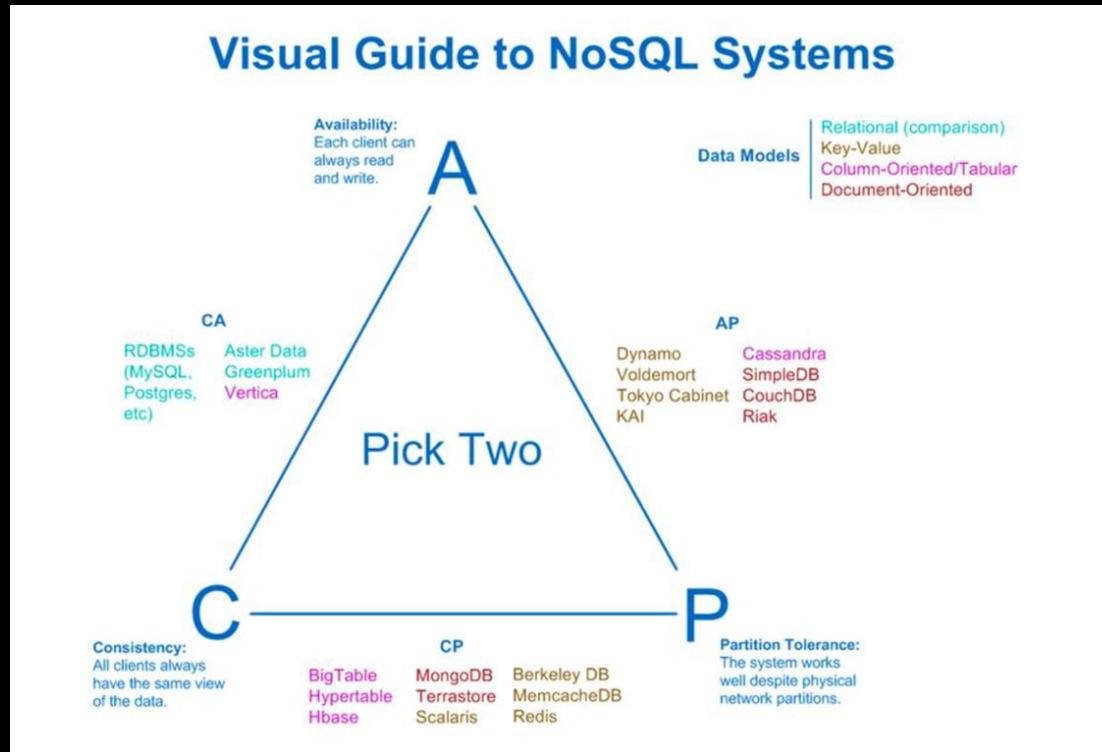
# CAP Theorem

- **C = Consistency**
  - Readers read the most recent update
- **A = Availability**
  - A valid answer is returned, even if one or more nodes are down
- **P = Partition Tolerance**
  - Partition: The network becomes disconnected
  - A distributed system works despite the network failure

# Incorrect (but Typical) Formulation

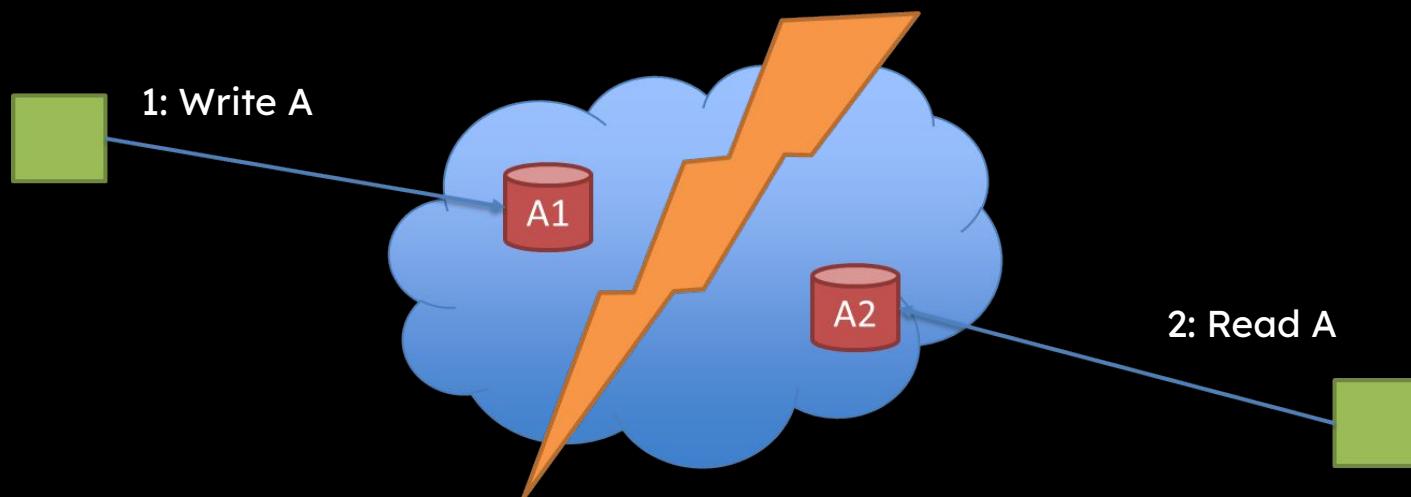
- You can only get **two** of  
**Consistency**,  
**Availability**, and  
**Partition Tolerance**

# Resulting Classification



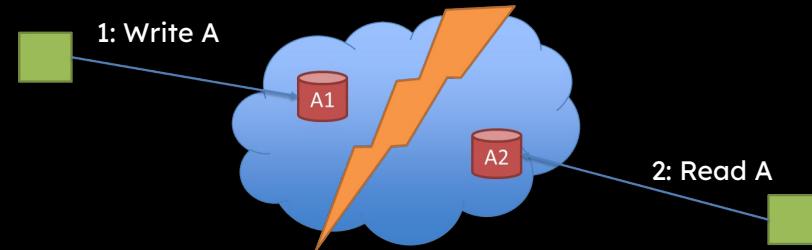
# Correct (but Useless) Formulation

- In a **partitioned** network, choose between **Consistency** and **Availability**
- Proof: Simple thought experiment

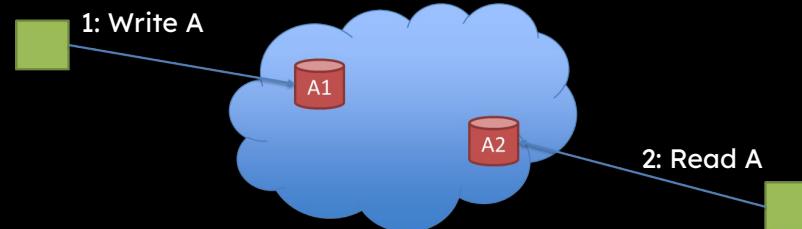


# PACELC (useful!) Formulation

- In a **partitioned** network, choose between **Availability and Consistency**



- Else (regular operation), choose between **Latency and Consistency**



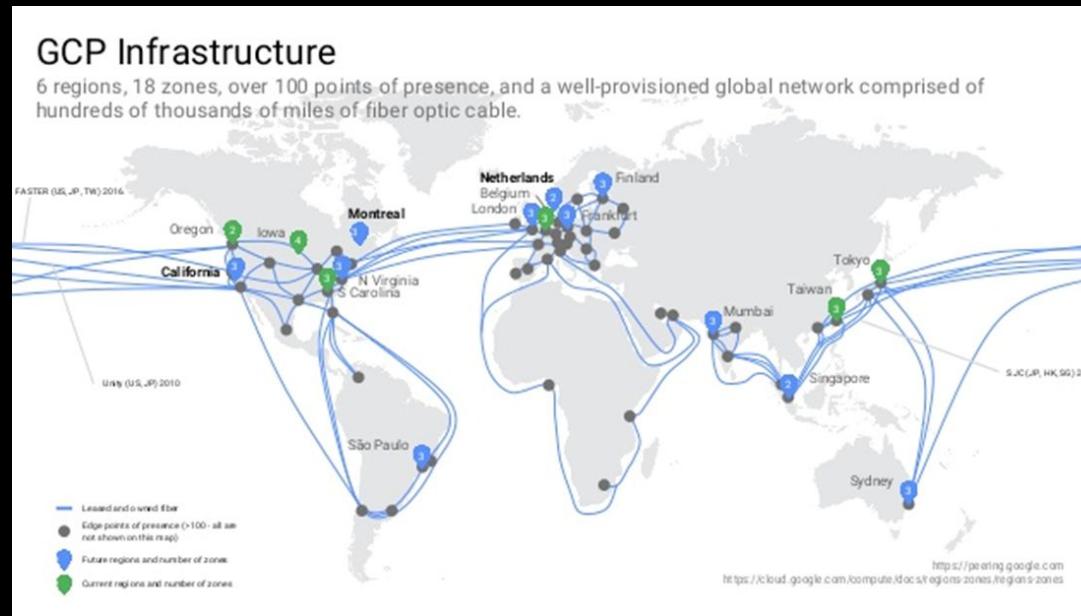
# 2012: Google Spanner

Cloud Spanner: The best of the relational and non-relational worlds

	CLOUD SPANNER	TRADITIONAL RELATIONAL	TRADITIONAL NON-RELATIONAL
Schema	✓ Yes	✓ Yes	✗ No
SQL	✓ Yes	✓ Yes	✗ No
Consistency	✓ Strong	✓ Strong	✗ Eventual
Availability	✓ High	✗ Failover	✓ High
Scalability	✓ Horizontal	✗ Vertical	✓ Horizontal
Replication	✓ Automatic	⟳ Configurable	⟳ Configurable

- Does it break the CAP theorem?

# What is High Availability?



- “Cloud Spanner [...] serves data with low latency while maintaining transactional consistency and industry-leading 99.999% (five 9s) availability - 10x less downtime than four nines (<5 minutes per year).”

# 2012: Google Spanner

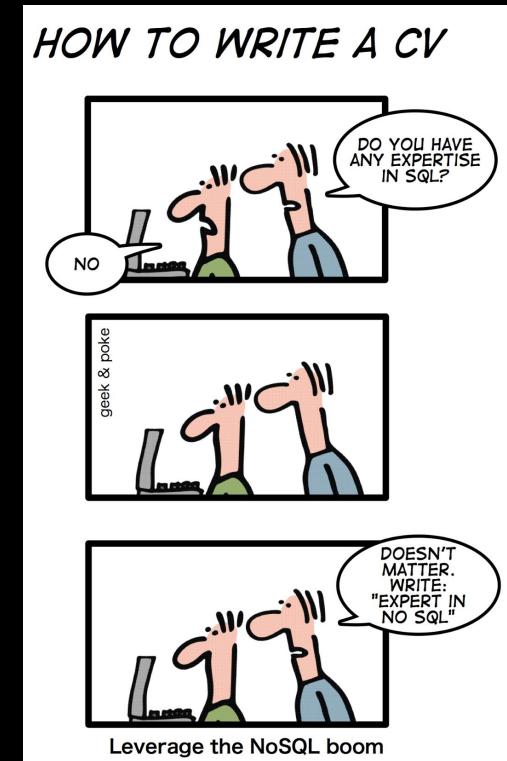
Cloud Spanner: The best of the relational and non-relational worlds

	CLOUD SPANNER	TRADITIONAL RELATIONAL	TRADITIONAL NON-RELATIONAL
Schema	✓ Yes	✓ Yes	✗ No
SQL	✓ Yes	✓ Yes	✗ No
Consistency	✓ Strong	✓ Strong	✗ Eventual
Availability	✓ High	✗ Failover	✓ High
Scalability	✓ Horizontal	✗ Vertical	✓ Horizontal
Replication	✓ Automatic	⟳ Configurable	⟳ Configurable

- Does it break the CAP theorem? **NO!**

# NoSQL

- A result of two trends:
  - More hardware
  - More data
- Data model
  - Not relational
  - No formally described schema
- Architecture
  - Usually distributed
- Mostly not ACID compliant
  - CAP = Consistency/Availability trade-off



# -- TODO

- ✓ Scaling Up/Out
- ✓ NoSQL
  - ✓ Data Model
  - ✓ Distributed Architecture
  - ✓ Consistency/Availability trade-off
    - ✓ Eventual Consistency
    - ✓ CAP Theorem

# Takeaways

Why do NoSQL devs eat alone?

...

They don't know how to join  
tables.

## **Scale Up vs Scale Out**

### NoSQL

- Non-relational Data Models
- Horizontal Scaling
- MongoDB, Redis, DynamoDB, Neo4j, etc.

### CAP Theorem

- Consistency
- Availability
- Partitions
- Pick between Consistency and Availability in case of network partitions

# Next Time in IDBS...

# Introduction to Database Systems

## IDBS - Autumn 2024

### Lecture 12 - Big Data

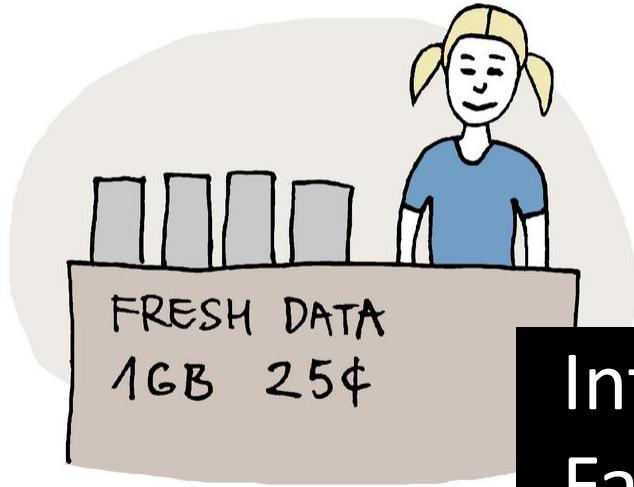
Big Data Analytics  
Distributed Computing Frameworks

---

Readings: PDBM 19.1-19.2, 19.4, 20.1-20.3

Martin Hentschel

# MONETIZING BIG DATA



## Introduction to Database Systems Fall 2024, Lecture 12: Big Data Management

 Dataedo /cartoon

# Lecture Outline

Learning Outcome 6: Discuss the pros and cons of different classes of data systems for modern analytics and data science applications

- Characteristics of Big Data
- Classes of Big Data Processing Systems
- Example: MapReduce, Spark

# Lecture Outline

Learning Outcome 6: Discuss the pros and cons of different classes of data systems for modern analytics and data science applications

- **Characteristics of Big Data**
- Classes of Big Data Processing Systems
- Example: MapReduce, Spark

## Online Transactional Processing (OTLP)

- Build great interactive applications
- “Small” data

## This class: beyond small data

- Keep track of all the history
  - Keep track of all interactions, also low-level
  - Keep track of all data: media, user input, logs
- “Big Data”

# Characteristics of Big Data

Five Vs of Big Data describe key dimensions of big data

Volume



Velocity



Variety



Veracity



Value





**Definition:** The amount of data generated, stored and processed.

**Significance:** Data management system must be scale to given data volume.

# Volume

VIEWERS WATCH  
**43 YEARS**  
OF STREAMING  
**CONTENT**

**3,720**  
USERS DOWNLOAD  
**INSTAGRAM**  
THREADS

**CHATGPT**  
USERS SEND  
**6,944** PROMPTS

**FACEBOOK**  
USERS LIKE

IT UNIVERSITY OF COPENHAGEN

# AMAZON

SHOPPERS SPEND  
**\$455K**

X  
USERS SEND  
**360K**  
TWEETS

# 6.3M

SEARCHES  
HAPPEN ON  
**GOOGLE**

# WHATSAPP

USERS SEND  
**41.6M**  
MESSAGES

# LINKEDIN

USERS SUBMIT  
**6,060**  
RESUMES

# CYBER-CRIMINALS

LAUNCH **30** DDOS ATTACKS

# INSTAGRAM

USERS SEND  
**694K** REELS VIA DM

# DOORDASH

DINERS PLACE

EVERY MINUTE  
**01:00**  
OF THE DAY

PRESENTED BY



Source: <https://www.domo.com/learn/infographic/data-never-sleeps-11>

Rivian electric vehicles generate multiple TB of data per day



CERN: „In June 2022, we had 424 PB of data on tapes”

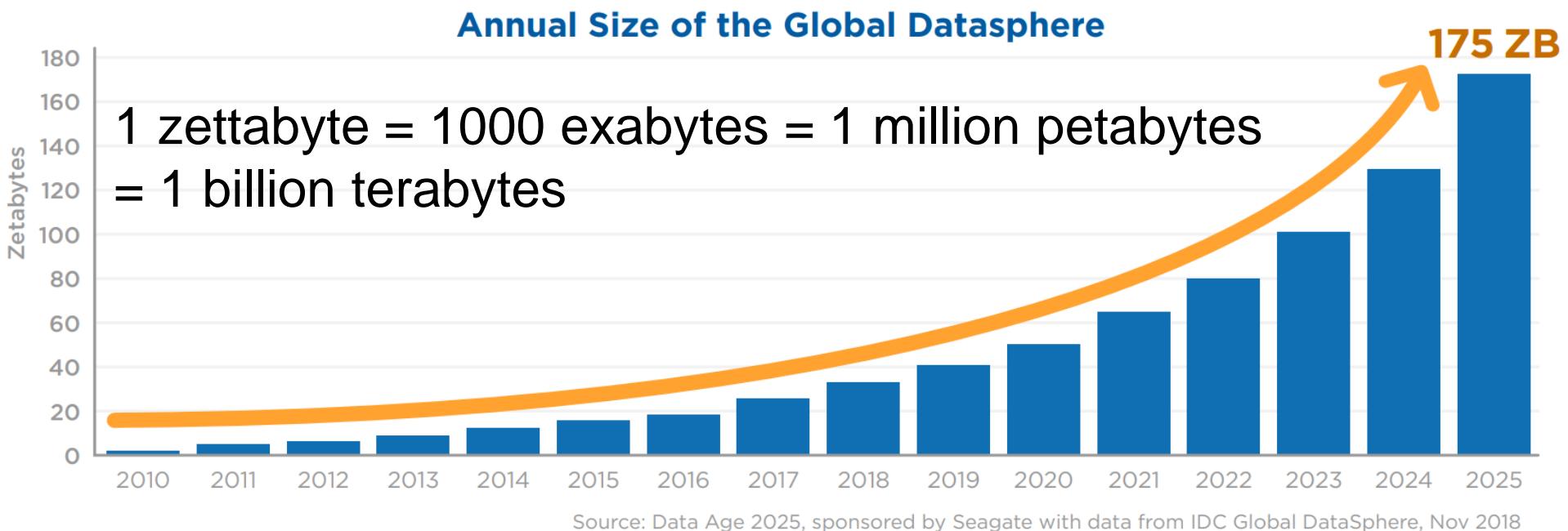


301,722 Datasets Available

# 15

## YEARS OF DATA.GOV

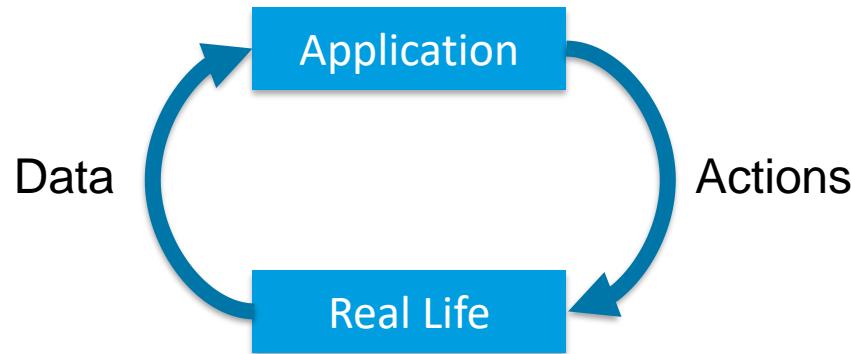
# Growth of Data Globally



Source: [Reinsel, Gantz, Rydning; The Digitization of the World from Edge to Core; IDC White Paper 2018](#)



**Definition:** The speed at which data is generated, collected, and processed.



**Significance:** Requirement on the system to process data at given speed.



**Definition:** The diversity of data types and sources. Data can come in **structured, semi-structured, or unstructured** formats.

**Significance:** Challenge of managing different types of data. Choose data management system for your use case.



## Structured data

- Highly regular structure, repeating patterns
- Example: relational data

ID	Name
1	Pinar
2	Veronika
3	Dovile
4	Zoi
5	Martin
6	Eleni



## Semi-structured data

- Some structure, changes over time
- Examples: logs, comment threads, graphs

Common Log Format\*

```
127.0.0.1 alice Alice [06/May/2021:11:26:42 +0200] "GET / HTTP/1.1" 200 3477
```

Same log line in JSON

```
{  
    "ip_address": "127.0.0.1",  
    "user_id": "alice",  
    "username": "Alice",  
    "timestamp": "06/May/2021:11:26:42 +0200",  
    "request_method": "GET",  
    "request_url": "/",  
    "protocol": "HTTP/1.1",  
    "status_code": 200,  
    "response_size_bytes": 3477  
}
```



## PHONES OUT

### Advantages of JSON over Common Log Format

<https://www.menti.com/al2n6oqco3vf>

Common Log Format

```
127.0.0.1 alice Alice [06/May/2021:11:26:42 +0200] "GET / HTTP/1.1" 200 3477
```

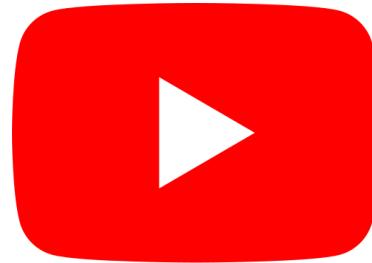
Same log line in JSON

```
{  
    "ip_address": "127.0.0.1",  
    "user_id": "alice",  
    "username": "Alice",  
    "timestamp": "06/May/2021:11:26:42 +0200",  
    "request_method": "GET",  
    "request_url": "/",  
    "protocol": "HTTP/1.1",  
    "status_code": 200,  
    "response_size_bytes": 3477  
}
```



## Unstructured data

- Little structure
- Examples: photos, videos, music





**Definition:** The accuracy and trustworthiness of data.

**Significance:** Data quality varies, and not all collected data can be trusted. It's essential to filter, clean, and validate data to ensure accurate insights.



# Only 3% of Companies' Data Meets Basic Quality Standards

by Tadhg Nagle, Thomas C. Redman, and David Sammon

September 11, 2017

Source: [Only 3% of Companies' Data Meets Basic Quality Standards](#), Harvard Business Review



A 2016 study by IBM is even more eye-popping. IBM found that poor data quality strips **\$3.1 trillion from the U.S. economy annually** due to lower productivity, system outages and higher maintenance costs, to name only a handful of the bad outcomes that result from poor data quality.

Source: [Flying Blind: How Bad Data Undermines Business](#), Forbes



**Definition:** The accuracy and trustworthiness of data.

**Significance:** Data quality varies, and not all collected data can be trusted. It's essential to filter, clean, and validate data to ensure accurate insights.



**Definition:** The ability to turn data into insights for decision making and creating business value.

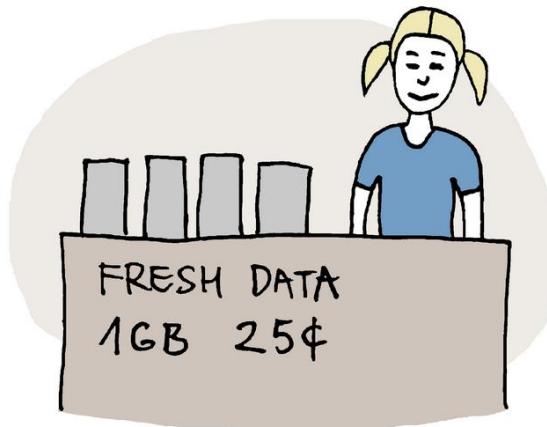
**Significance:** Big data is only beneficial if it delivers insights.

- Businesses: more effective operations, stronger customer relationships, \$\$\$
- Science: new discoveries
- Public sector: better serving people



Only half a joke: data marketplaces are a thing!

## MONETIZING BIG DATA





## Snowflake Data Sharing & Marketplace

### DATA SHARING<sup>1</sup>

**36%**

of customers<sup>1</sup> have at  $\geq 1$  stable edge<sup>1</sup>

### MARKETPLACE LISTINGS<sup>1</sup>

**2,946**

26% Y/Y Growth

Source: [Snowflake Investor Presentation](#), October 31, 2024

# Characteristics of Big Data

Five Vs of Big Data describe key dimensions of big data

Volume



Velocity



Variety



Veracity



Value

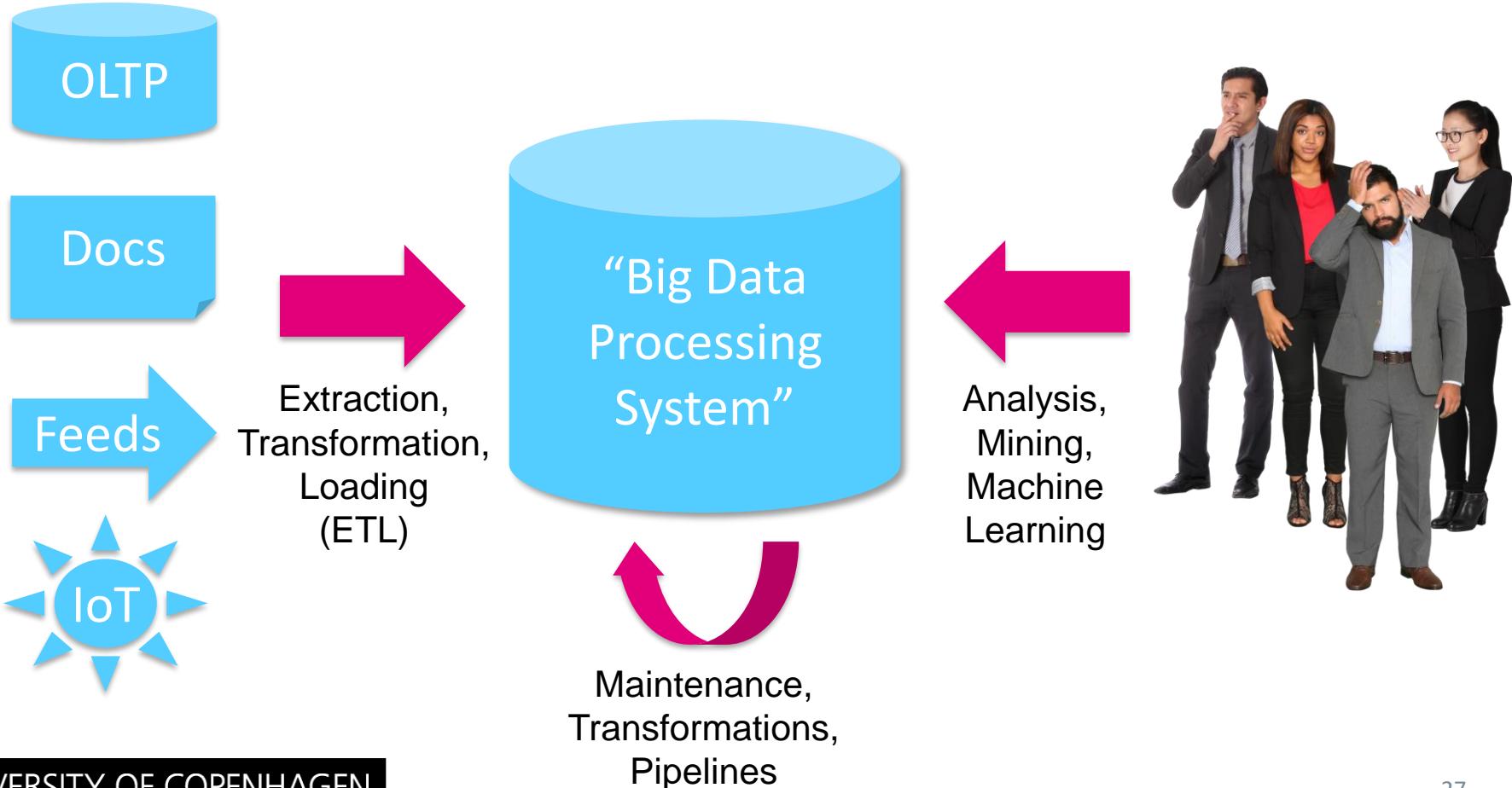


# Lecture Outline

Learning Outcome 6: Discuss the pros and cons of different classes of data systems for modern analytics and data science applications

- Characteristics of Big Data
- **Classes of Big Data Processing Systems**
- Example: MapReduce, Spark

# Big Data Processing Overview



# Access Patterns to Big Data

Data collections typically consists of many millions of files of data

- Too large for a single server → Distributed storage
- Reading large amounts of data → Sequential scans

## Online Analytical Processing (OLAP)

# Processing Requirements

Apply filters to reduce data quantity

- Like SQL execution with predicates

Run complex processing pipelines

- Periodic tasks & triggered tasks upon new data
- User-defined functions (UDFs)
- Server-side applications (Python, ML)

Too large for a single server → **Distributed processing**



## OLAP Database System

- Distributed storage for exabytes of data
- Distributed compute, typically using large, sequential scans
- Large ecosystem of tools for complex data analysis tasks

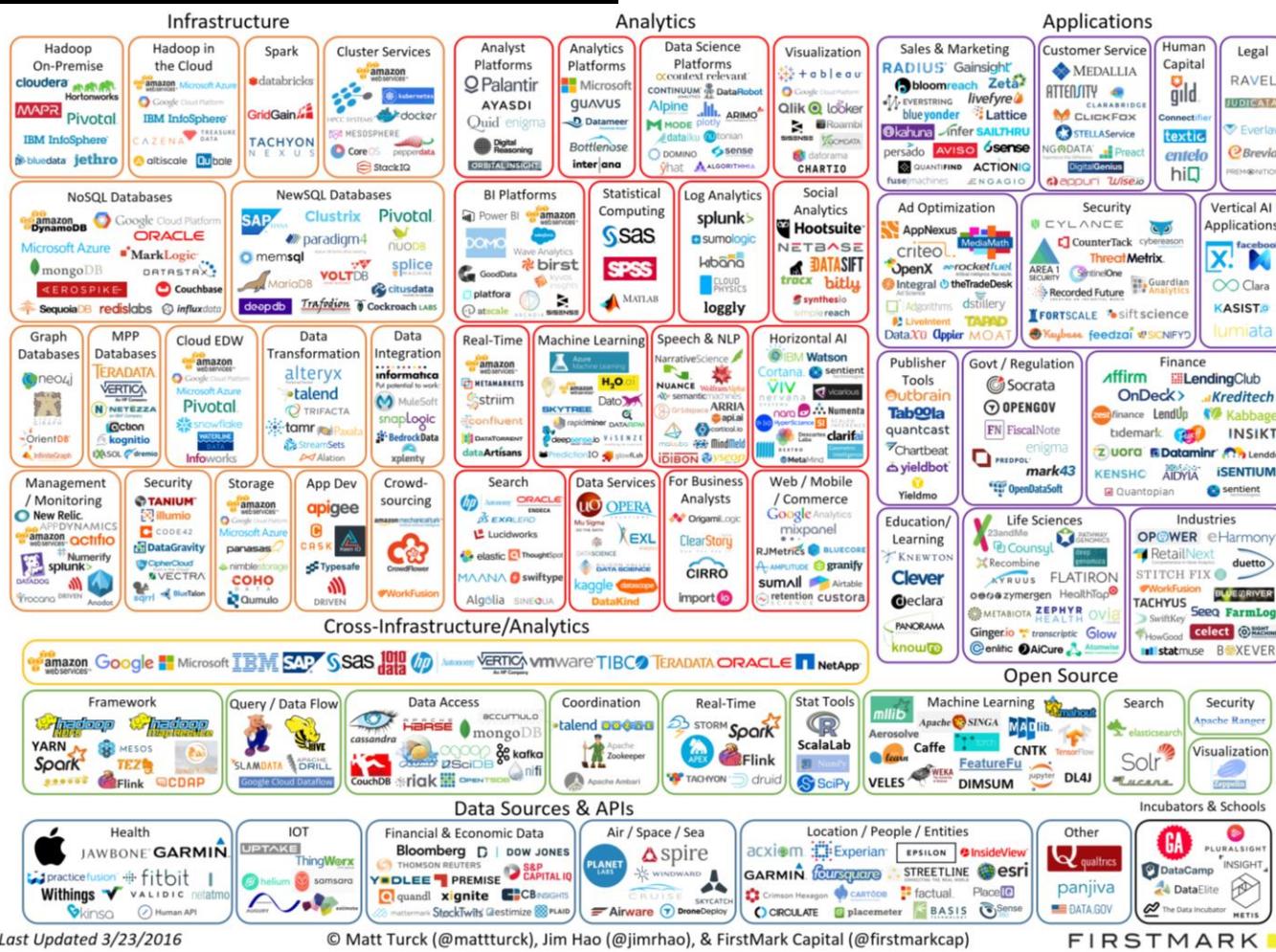
# Big Data Landscape 2012



© Matt Turck (@mattturck) and ShivonZilis (@shivonz)

Source: [A chart of the big data ecosystem – Matt Turck](#)

# Big Data Landscape 2016

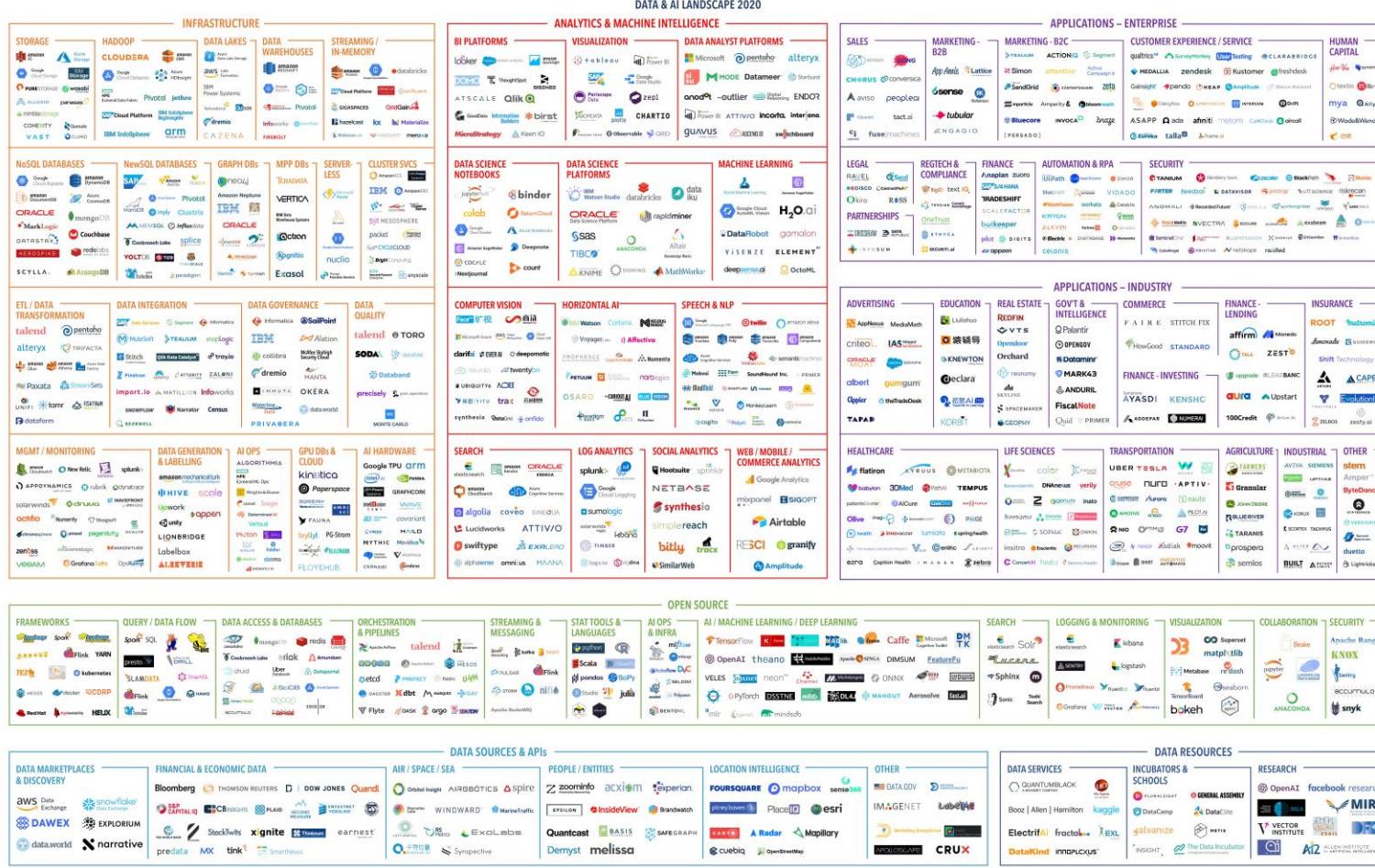


Last Updated 3/23/2016

© Matt Turck (@mattturck), Jim Hao (@jimrhao), & FirstMark Capital (@firstmarkcap)

FIRSTMARK

# Big Data Landscape 2020



Version 1.0 - September 2020

© Matt Turck (@mattturck) & FirstMark (@firstmarkcap)

mattturck.com/data2020

FIRSTMARK

Source: Resilience and Vibrancy: The 2020 Data & AI Landscape – Matt Turck

# Big Data Landscape 2023

THE 2023 MAD (MACHINE LEARNING, ARTIFICIAL INTELLIGENCE & DATA) LANDSCAPE



# Conclusions:

- No product does it all
  - Typically, you use a set of products
    - e.g. ETL → Storage & Processing → UI, Reporting, Serving
  - Interesting space for software engineers, sales engineers, sales

# Classes of Big Data Processing Systems

## Relational Database Systems

### Strengths

- SQL programming API
- Huge SQL ecosystem
- ACID transactions
- Complex query processing

### Weaknesses

- Little support for unstructured data
- Little support for machine learning training and serving



Amazon Redshift

teradata.

# Classes of Big Data Processing Systems

## MapReduce Systems

### Strengths

- Distributed storage
- Great scalability
- Able to process great variety of data sets (structured, semi-structured, unstructured)

### Weaknesses

revealed later



**CLOUDERA**



# Classes of Big Data Processing Systems

## Key-value Stores

### Strengths

- Distributed storage
- Great scalability
- Short latency for single items
- Quick, “out of the box” way to store data

### Weaknesses

- Limited storage model (only key, value pairs)
- Limited query interface (no SQL)
- No means to scan and filter data



Amazon DynamoDB

Redis



# Classes of Big Data Processing Systems

## Document Databases

### Strengths

- Store objects / XML / JSON in hierarchical form
- Good integration with object-oriented languages and JavaScript

### Weaknesses

- Limited query interface (no SQL)
- No ACID guarantees
- Not designed for scans (very related to key-value stores)



# Classes of Big Data Processing Systems

## Graph Databases

### Strengths

- Capture graph relationships, e.g. knowledge graphs, social networks
- Fast at traversing edge chains, no joins needed



### Weaknesses

- Specific to graph applications
- Not many such use cases
- Relational databases outperform graph databases these days



# Classes of Big Data Processing Systems

## Data Lakehouses

### Strengths

- Data storage in open data formats (Apache Parquet + Apache Iceberg) in the cloud
- Great ML support, e.g. training, but also Python-based notebooks

### Weaknesses

- Typical API is Apache Spark, less versatile and supported than SQL
- However: Lakehouses are moving to SQL



**databricks**

# Lecture Outline

Learning Outcome 6: Discuss the pros and cons of different classes of data systems for modern analytics and data science applications

- Characteristics of Big Data
- Classes of Big Data Processing Systems
- **Example: MapReduce, Spark**

## Early tools to deal with Big Data

- MapReduce by Google in 2003
- Open-source Apache Hadoop based on MapReduce

## Interface

- Map       $(k_1, v_1) \rightarrow \text{list } (k_2, v_2)$
- Reduce     $(k_2, \text{list}(v_2)) \rightarrow (k_2, v_3)$

## Framework

- Read lots of data (e.g. text documents)
- **Map:** process a data item
- Sort and shuffle
- **Reduce:** aggregate
- Write results

Map and Reduce functions are user-specified

# MapReduce Phases

## Map Phase

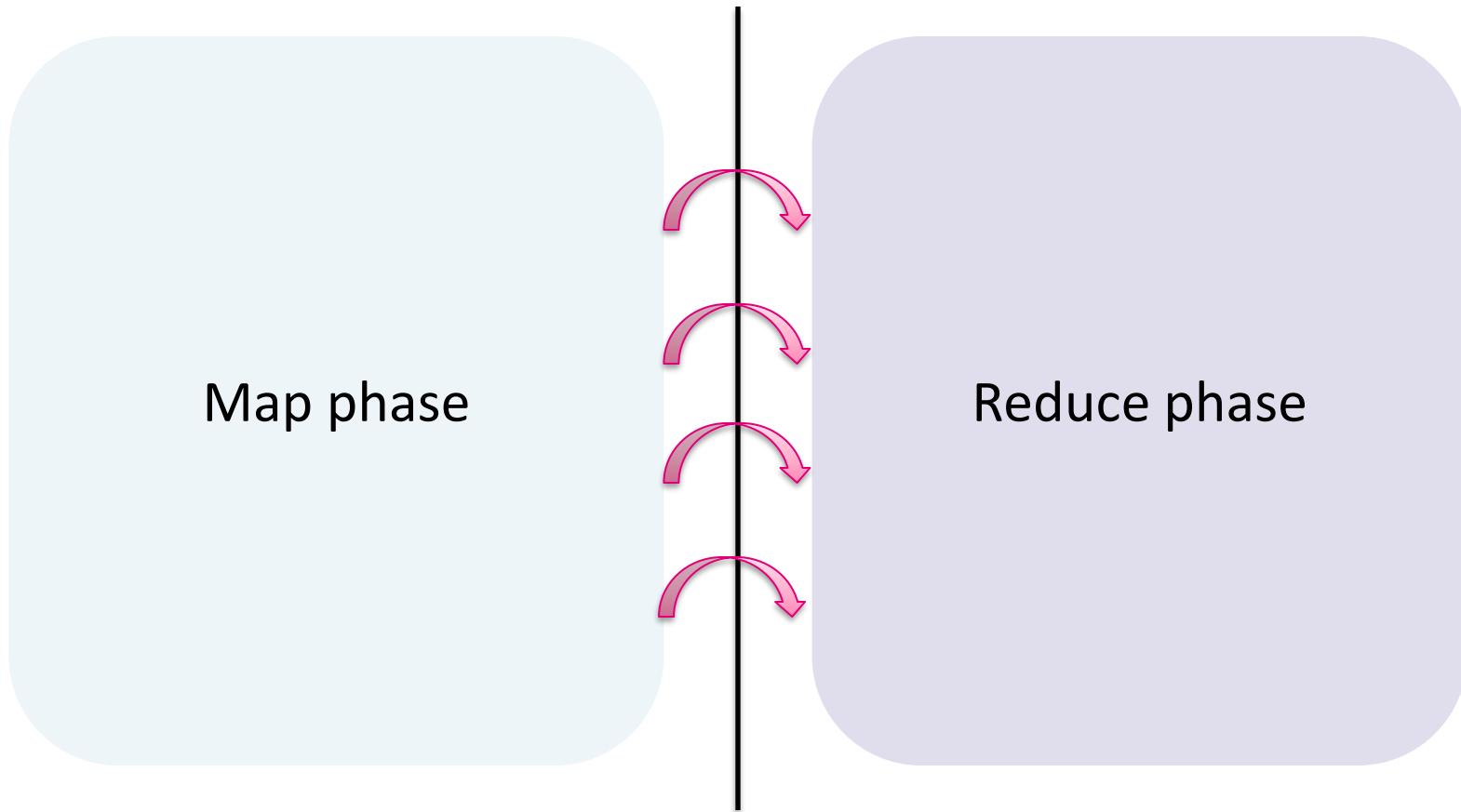
- Input:  $(k_1, v_1)$
- Output: list  $(k_2, v_2)$
- Independent for every key-value pair  
→ mapping processes run in parallel

**Shuffle Phase:** intermediate results are shuffled through the network

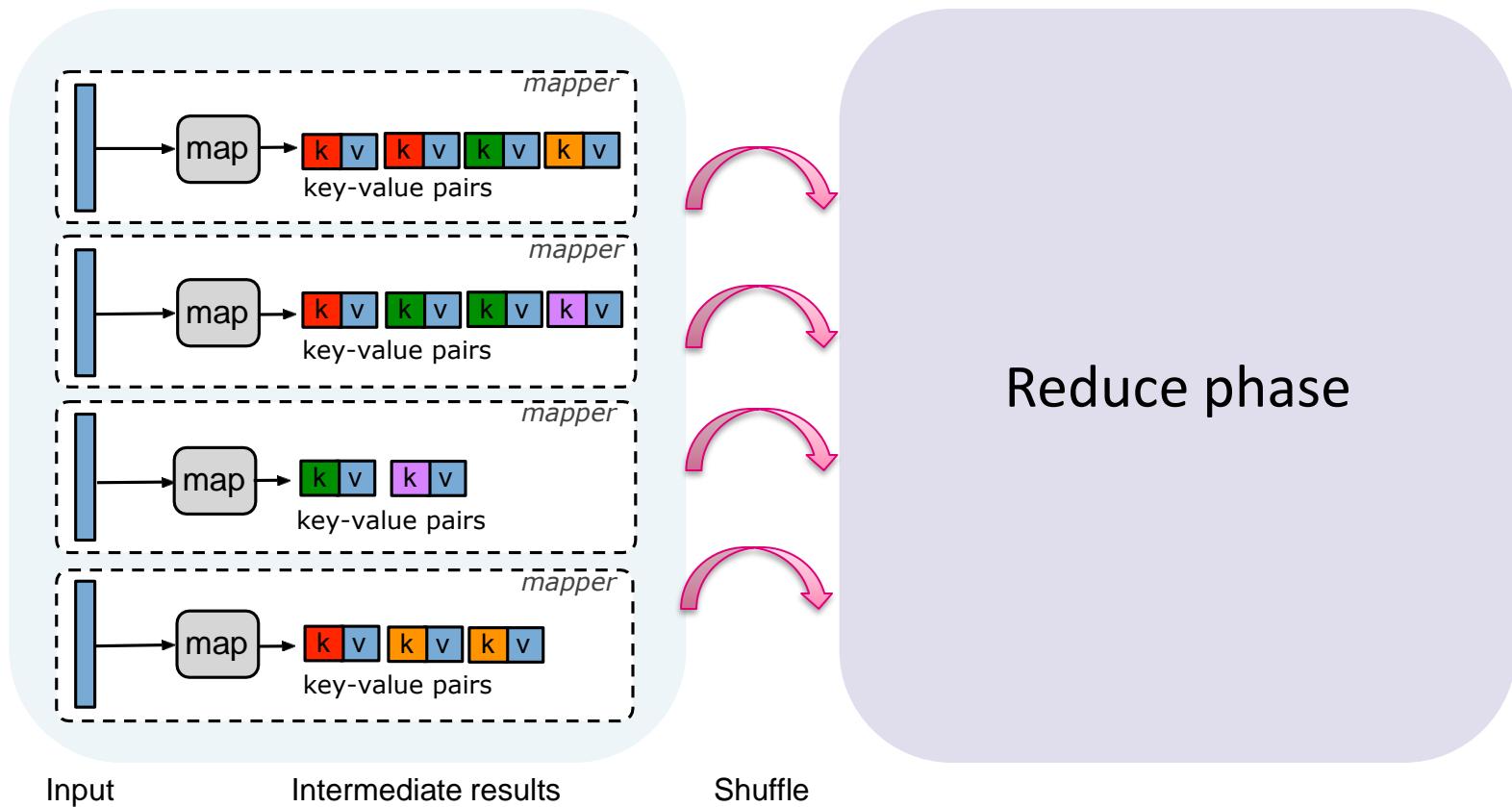
## Reduce Phase

- Input:  $(k_2, \text{list}(v_2))$
- Output:  $(k_2, v_3)$
- Independent per group  
→ reducer processes can run in parallel (per group)

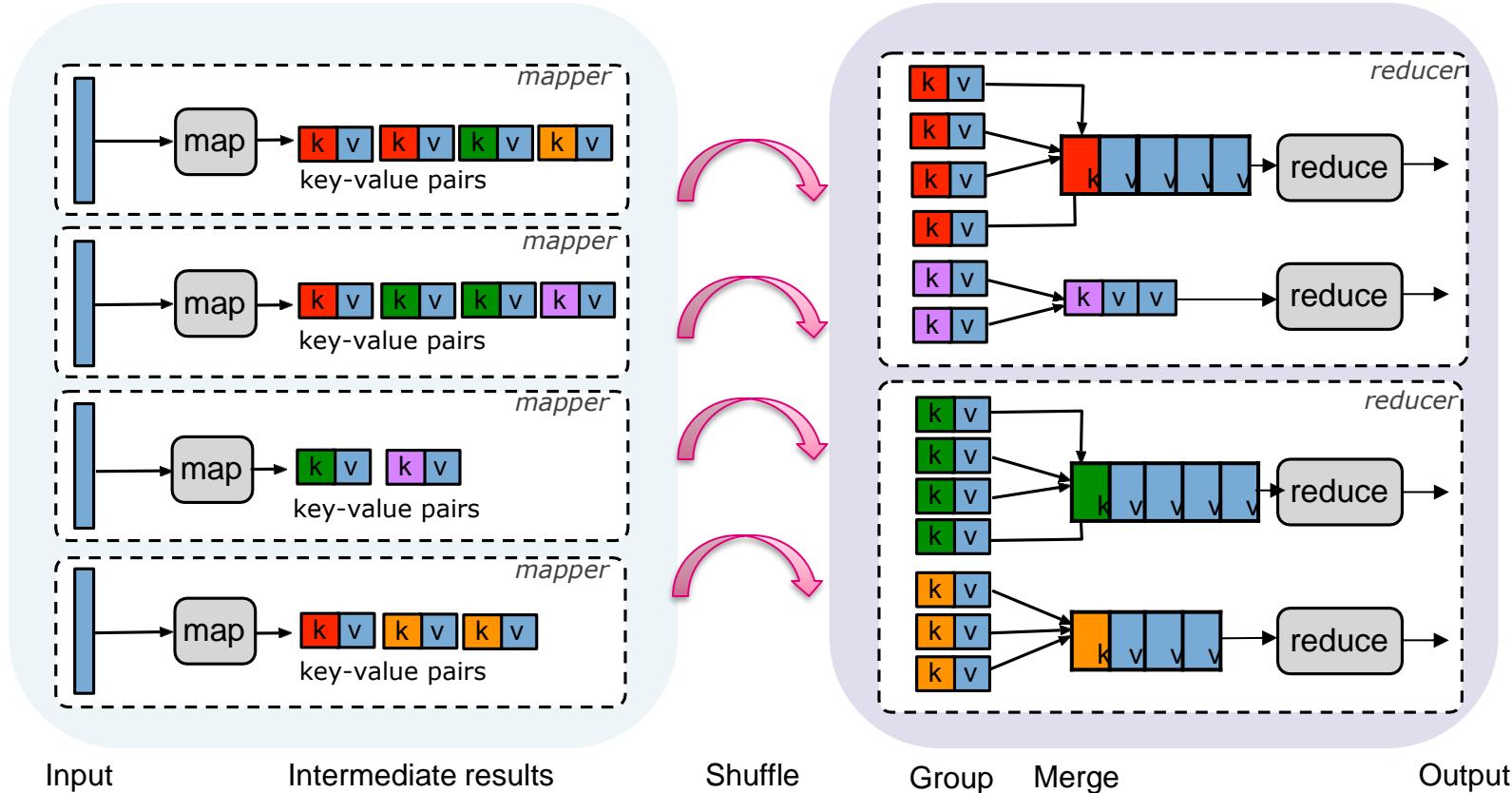
# MapReduce Illustration



# MapReduce Illustration



# MapReduce Illustration



# MapReduce Wordcount Example

## Wordcount Example

- Given a text document, output the count for each word.

## MapReduce interface

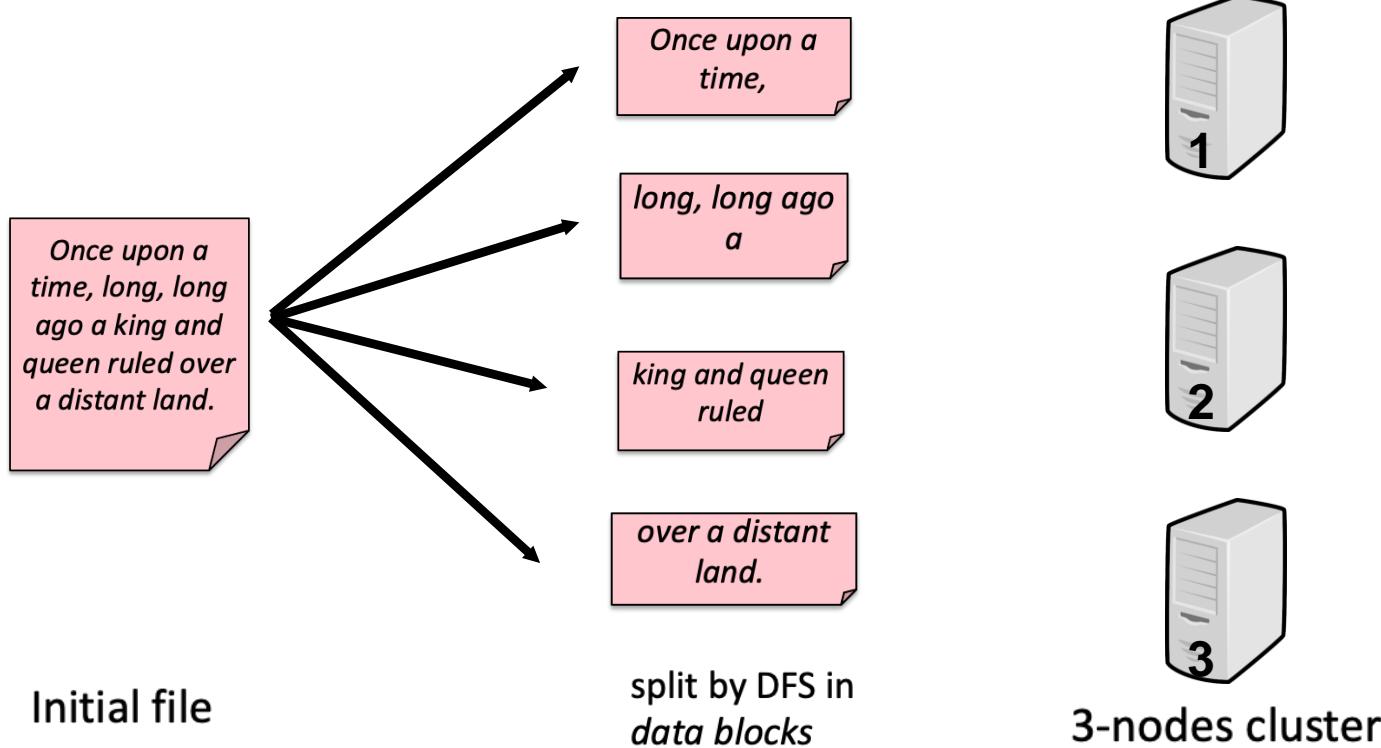
- Map  $(k_1, v_1) \rightarrow \text{list } (k_2, v_2)$
- Reduce  $(k_2, \text{list}(v_2)) \rightarrow (k_2, v_3)$

# MapReduce Wordcount Example

## MapReduce implementation

- Map       $(\text{filename}, \text{text}) \rightarrow \{\text{word}, 1\}$
- Reduce     $(\text{word}, [1,1,\dots]) \rightarrow (\text{word}, \text{count})$

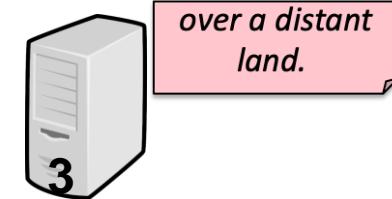
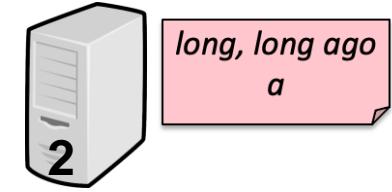
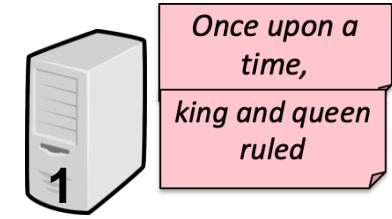
# MapReduce Wordcount Example



# MapReduce Wordcount Example

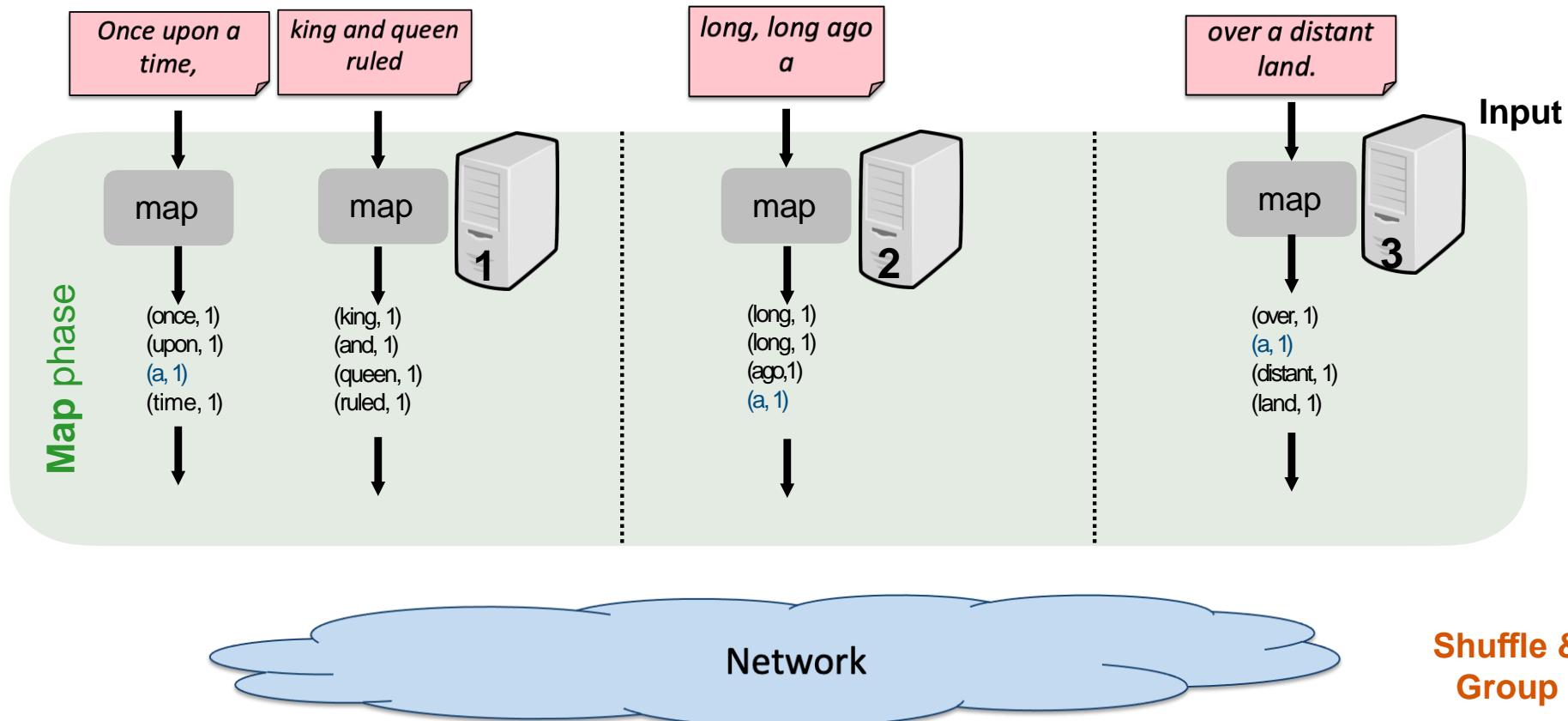
*Once upon a time, long, long ago a king and queen ruled over a distant land.*

Initial file

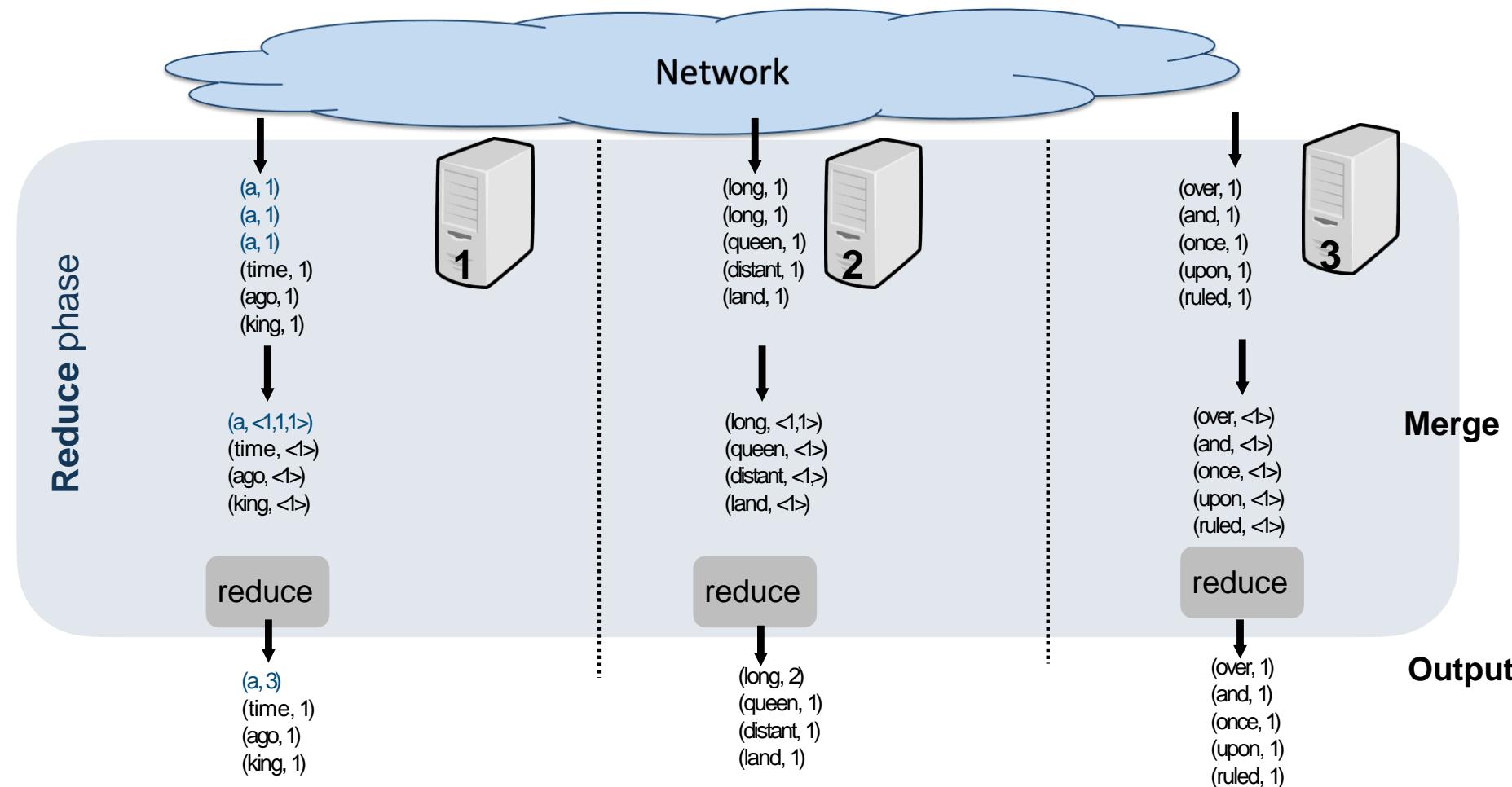


3-nodes cluster

# Map Phase



# Reduce Phase



# Pseudocode

```
map(String key, String value):
```

?

```
reduce(String key, Iterator values):
```

?

## MapReduce

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

Source: [Dean, Ghemawat; MapReduce: Simplified Data Processing on Large Clusters; OSDI'2004](#)

## SQL

```
select word, count(*) from docs
group by word
```

## Early tool to operate on Big Data

- Invented by Google in 2003
- Initially great tool for distributed processing of large amounts of data
- Used to build the web page index, many other applications
- Google moved away in 2010, killed it in 2014
- Apache Hadoop's market share on significant decline

# Classes of Big Data Processing Systems

## MapReduce Systems

### Strengths

- Distributed storage
- Great scalability
- Able to process great variety of data sets (structured, semi-structured, unstructured)

### Weaknesses



<https://www.menti.com/al2n6oqco3vf>



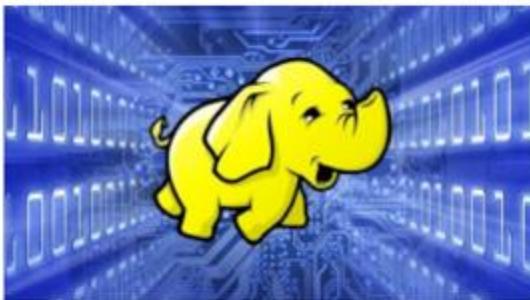
**CLOUDERA**



March 13, 2017

## Hadoop Has Failed Us, Tech Experts Say

Alex Woodie



The Hadoop dream of unifying data and compute in a distributed manner has all but failed in a smoking heap of cost and complexity, according to technology experts and executives who spoke to *Datanami*.

"I can't find a happy Hadoop customer. It's sort of as simple as that," says Bob Muglia, CEO of [Snowflake Computing](#), which develops and runs a cloud-based relational data warehouse offering. "It's very clear to me, technologically, that it's not the technology base the world will be built on going forward."

# Why bother?

Teaches foundational concepts in distributed systems

- **Parallel computing:** MapReduce divides data processing tasks into small, parallelizable units
- **Data processing:** Easy to understand data flow through the system
- **Fault tolerance:** internal mechanism to recover from failures
- **Scalability:** demonstrates how to scale “horizontally” (across many servers)

# Lecture Outline

Learning Outcome 6: Discuss the pros and cons of different classes of data systems for modern analytics and data science applications

- Characteristics of Big Data
- Classes of Big Data Processing Systems
- **Example:** MapReduce, **Spark**



Improvement over MapReduce, developed in 2012

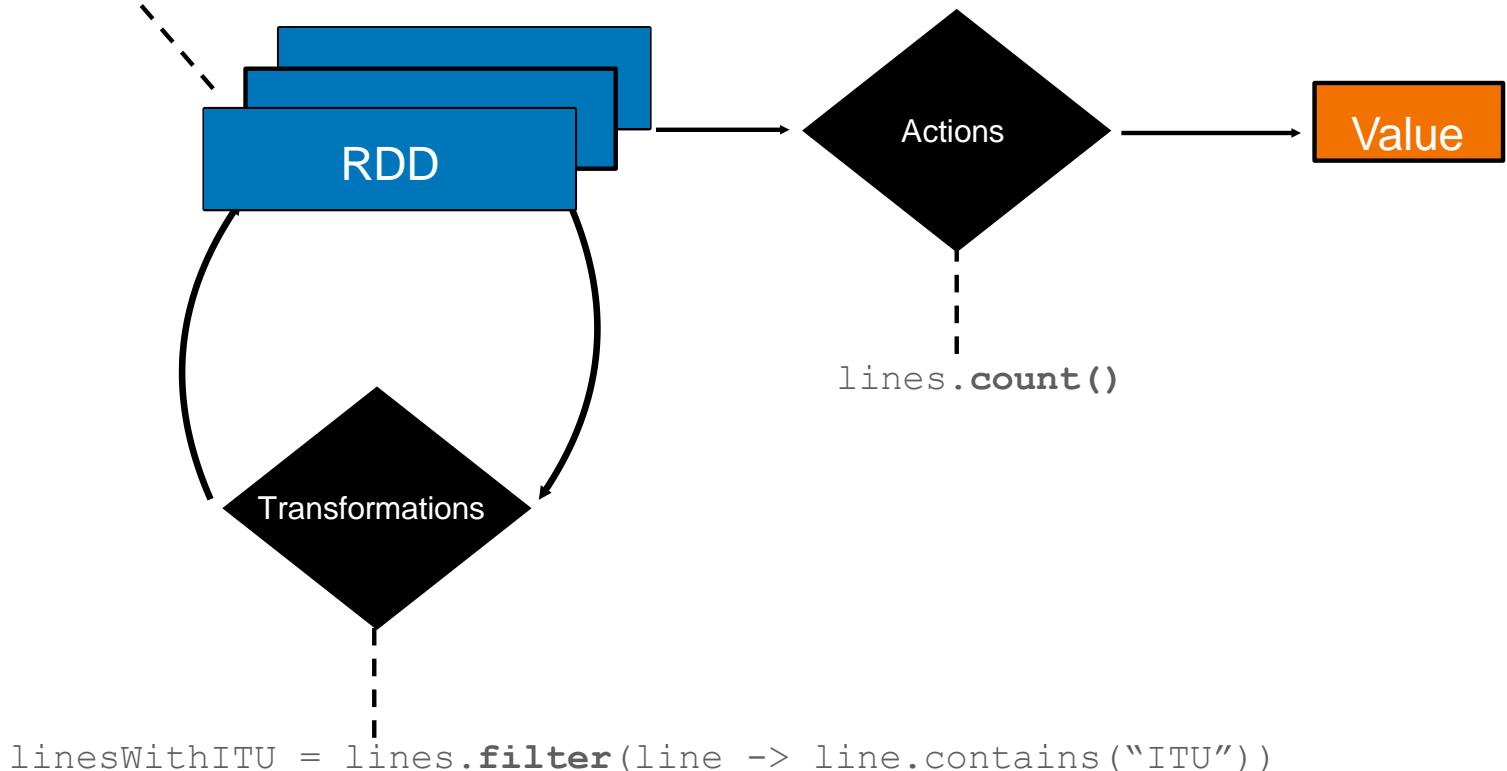
Data model: Resilient Distributed Datasets (RDDs)

- Transform one RDD to another via operators
- Lazy execution – optimizations

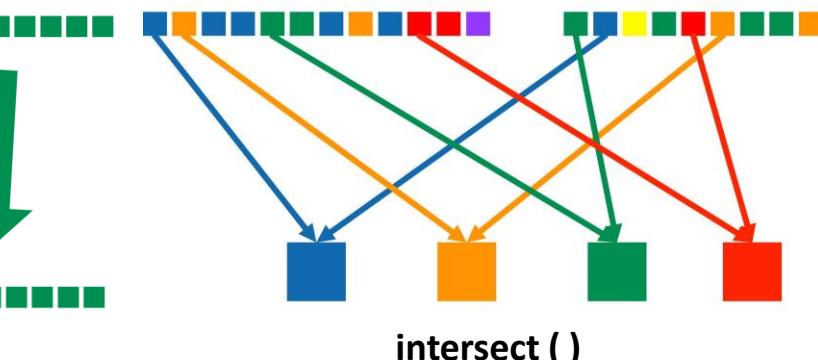
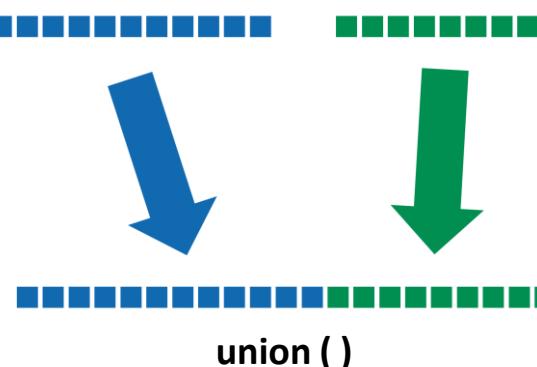
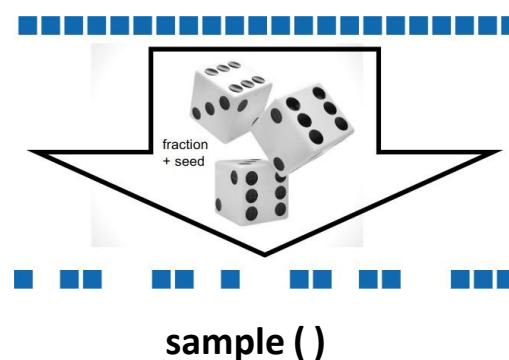
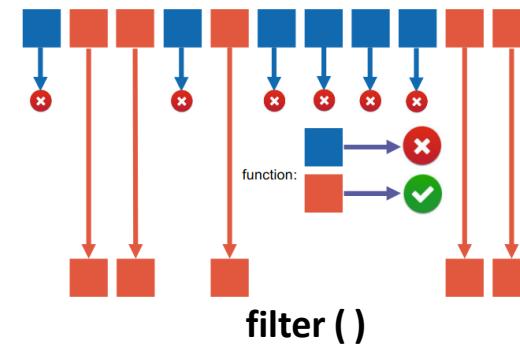
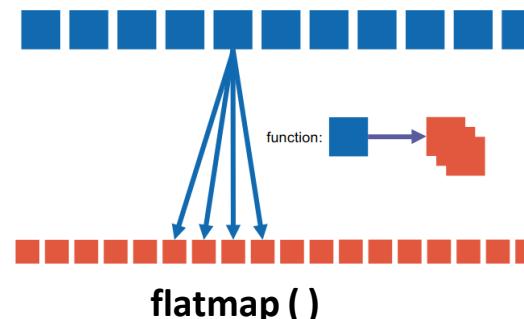
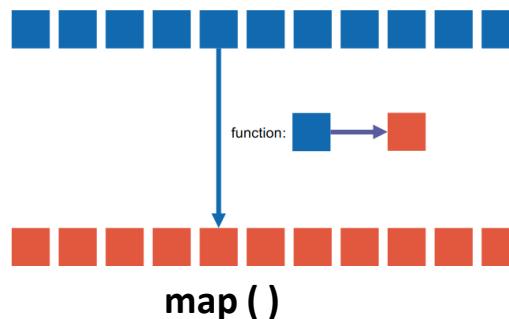
Write programs in terms of distributed datasets and operations on them

# Working with RDDs

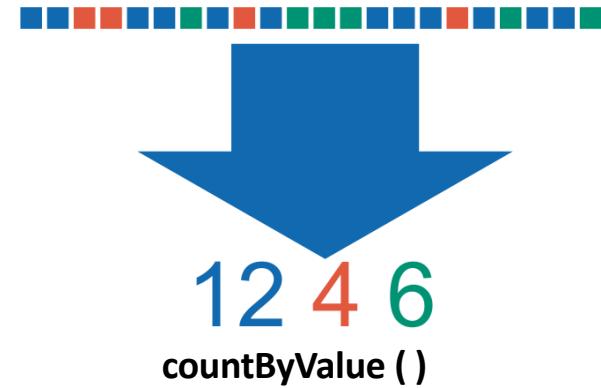
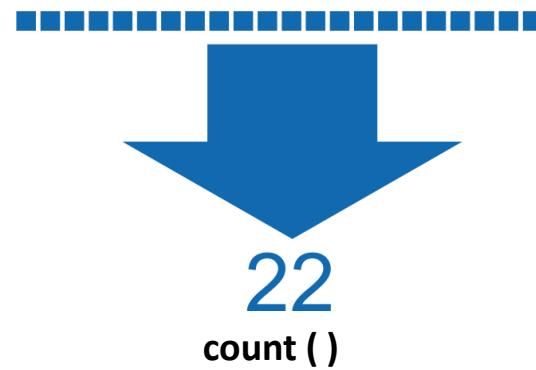
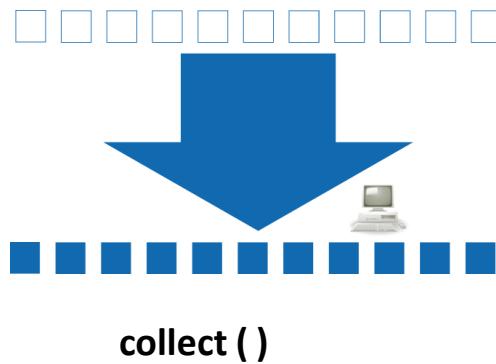
```
lines = sc.textFile("hdfs://data.txt")
```



# Example Transformations

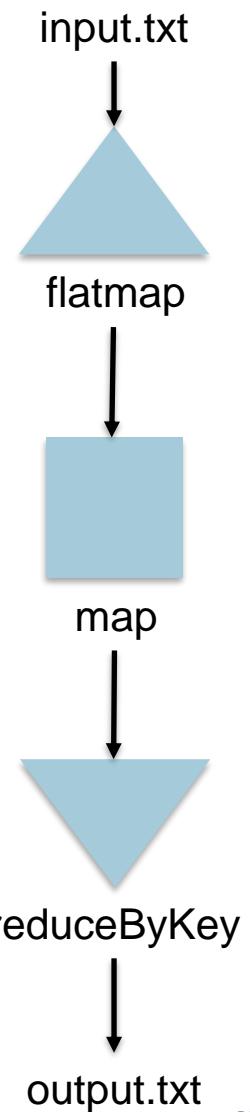


# Example Actions



# Spark Wordcount Example

```
file = sc.textFile("hdfs://...")  
counts = file.flatMap(lambda line: line.split(" " ))  
    .map(lambda word: (word, 1))  
    .reduceByKey(lambda a, b: a + b)  
counts.saveAsTextFile("hdfs://...")
```



# Take Aways

**Learning Outcome 6:** Discuss the pros and cons of different classes of data systems for modern analytics and data science applications

- Data sizes and requirements on systems continue to grow
- Many systems to process big data initially built to solve difficult problems (e.g. MapReduce for large, distributed data manipulation)
- SQL systems mostly caught up, pushed specialized systems out
- Pattern repeating (LLM training, vector databases)

## Recommended reading

- Michael Stonebraker, Andrew Pavlo, What Goes Around Comes Around... And Around..., SIGMOD Record, June 2024

<https://db.cs.cmu.edu/papers/2024/whatgoesaround-sigmodrec2024.pdf>