

CS2102: Database Systems

Lecture 1 — Introduction & Relational Model

Overview

- **Why Database Management Systems (DBMS)?**
 - Challenges for data-intensive applications
 - From file-based data management to DBMS
 - Core concepts of DBMS (transactions, data abstraction)
- **Relational Database Model**
 - Motivation & history
 - Core concepts: relation, domain, schema, etc.
 - Integrity constraints
- **Summary**

Common Challenges for Data-Intensive Applications

- Fast access to information in huge volumes of data
→ **Efficiency**
- "All-or-nothing" changes to data
(e.g. bank transfer: debit + credit)
→ **Transactions**
- Parallel access and changes to data
→ **Data Integrity**



5,000 tps*



100,000 tps*



544,000 tps*

Common Challenges for Data-Intensive Applications

- Fast and reliable handling of failures
(e.g., HDD/SDD/system crash, power outage, network disruption)

→ Recovery

- Fine-grained data access rights

→ Security

Only HR & Management

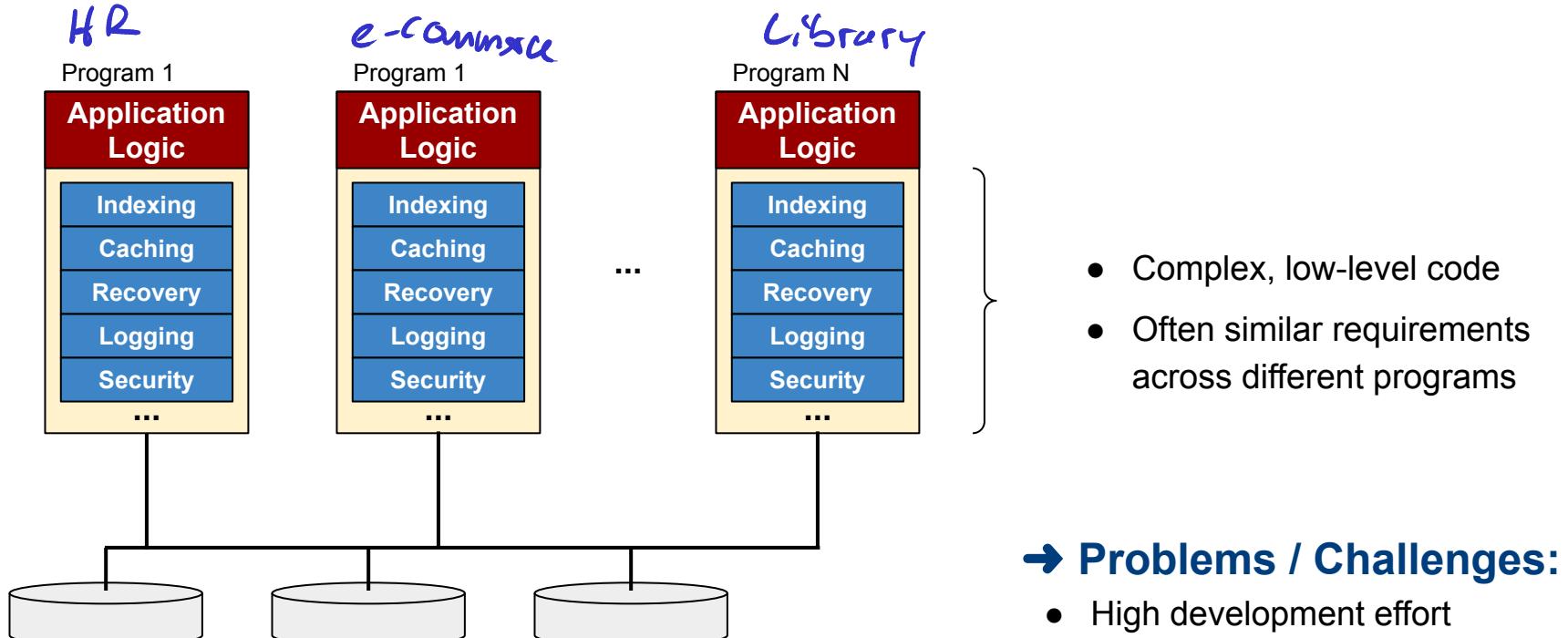
EmplID	Name	Office	Phone	DOB	Salary
1	Alice	02-05	4520	10-08-1988	7,500
2	Bob	02-10	4530	06-11-2001	4,800
3	Carol	01-06	4540	25-02-1995	5,500

All employees

Overview

- **Why Database Management Systems (DBMS)?**
 - Challenges for data-intensive applications
 - **From file-based data management to DBMS**
 - Core concepts of DBMS (transactions, data abstraction)
- **Relational Database Model**
 - Motivation & history
 - Core concepts: relation, domain, schema, etc.
 - Integrity constraints
- **Summary**

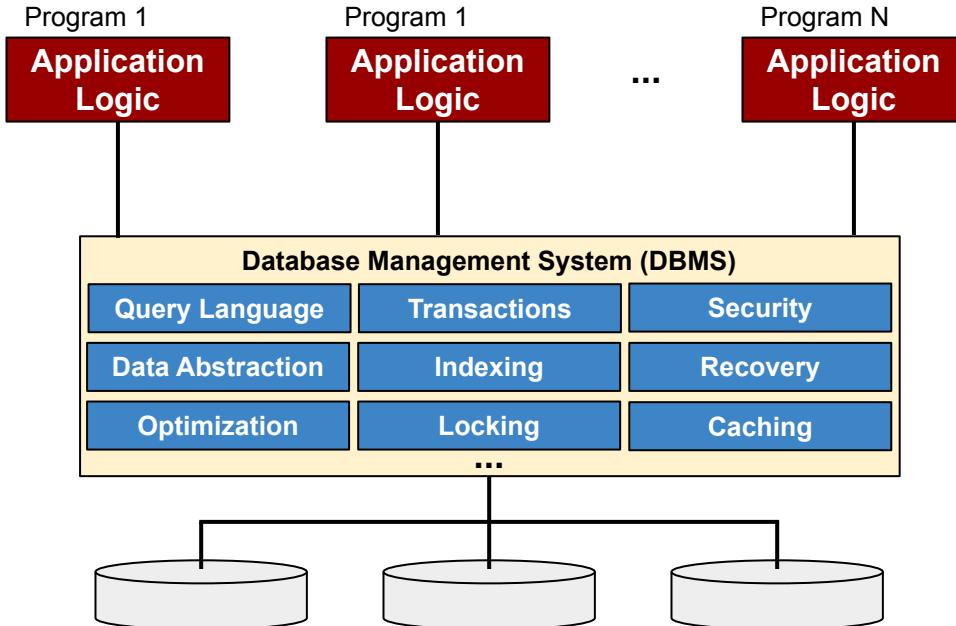
File-Based Data Management



→ Problems / Challenges:

- High development effort
- Long development times
- Higher risk of (critical) errors

Data Management with DBMS



- Complex, low-level code moved from application logic to DBMS
- DBMS = set of universal and powerful functionalities for data management

→ Benefits:

- Faster application development
- Increased productivity
- Higher stability / less errors

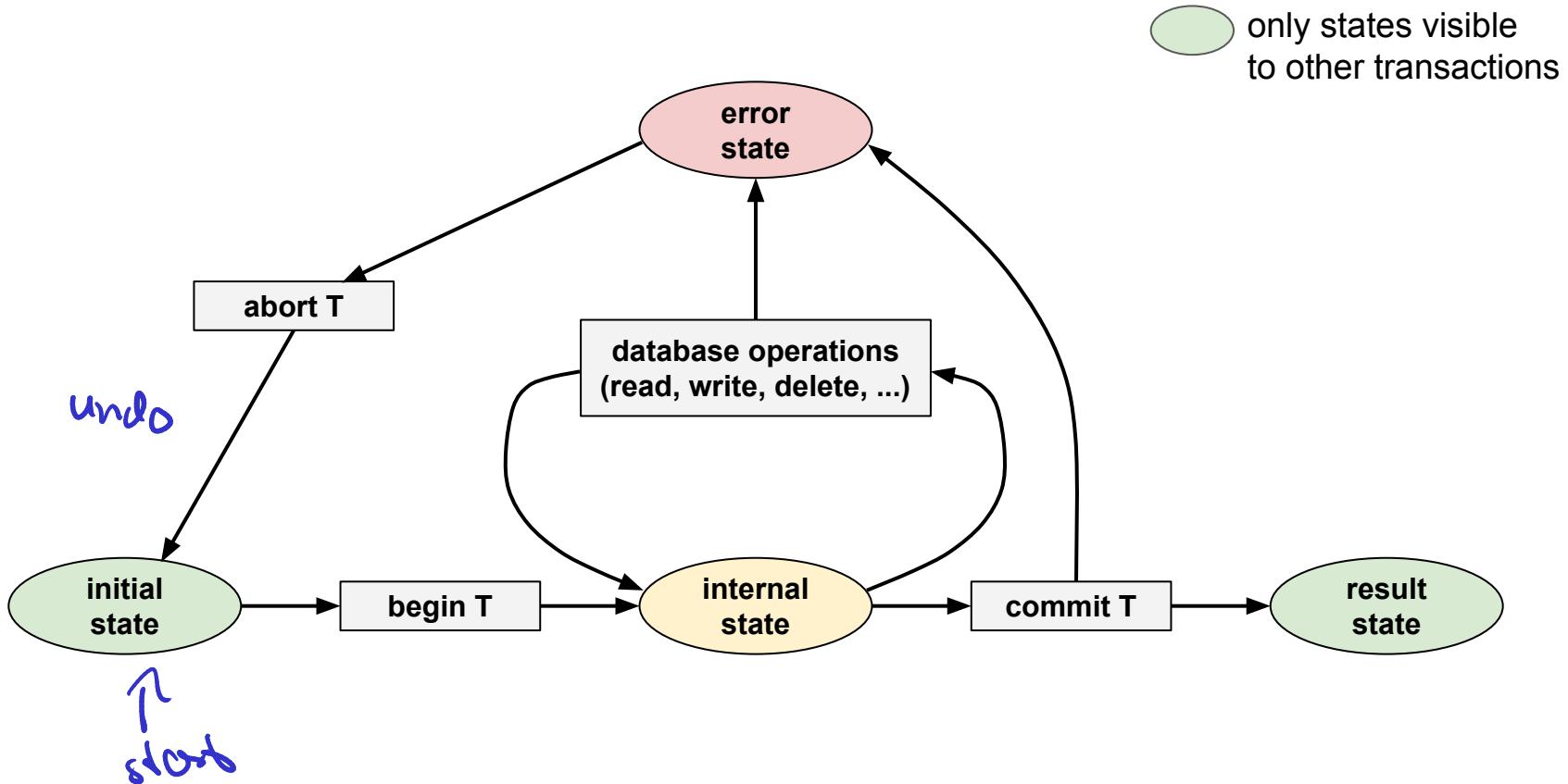
Overview

- **Why Database Management Systems (DBMS)?**
 - Challenges for data-intensive applications
 - From file-based data management to DBMS
 - **Core concepts of DBMS (transactions, data abstraction)**
- **Relational Database Model**
 - Motivation & history
 - Core concepts: relation, domain, schema, etc.
 - Integrity constraints
- **Summary**

Separating "Files Only" from DBMS: Transactions

- **Transaction**
 - Finite sequence of database operations (reads and/or writes)
 - Smallest logical unit of work from an application perspective
- Each transaction T has the following properties:
 - **Atomicity**: either all effects of T are reflected in the database or none ("*all or nothing*")
 - **Consistency**: the execution of T guarantees to yield a correct state of the database
 - **Isolation**: the execution of T is isolated from the effects of concurrent transactions
 - **Durability**: after the commit of T, its effects are permanent even in case of failures
- **ACID** properties of transactions

Transition Graph of a Transaction T



Transactions — Example: Update Bank Account Balance

- Very simple transaction

Transaction update(X, amount)

```
begin:  
    read(X)  
    X = X + amount  
    write(X)  
commit
```

- Assume 2 transactions

(initial balance B: 1,000)

- T₁(B, 500)
 - T₂(B, 100)
- 
- 1,600

Serial execution of T₁ and T₂

T ₁ (B, 500)	T ₂ (B, 100)
begin	
read(B)	
B = B + 500	
write(B)	
commit	
	begin
	read(B)
	B = B + 100
	write(B)
	commit

+ Correct final result (by definition)

- Less resource utilization and low throughput

Concurrent Execution — Common Problems

$T_1(B, 500)$	$T_2(B, 100)$
begin	
read(B)	
$B = B + 500$	
begin	
read(B) $\rightarrow 1,000$	
$B = B + 100 \rightarrow 1,100$	
write(B)	
commit	
	write(B)
	commit

Final balance $B = 1,100$
(effect of T_1 overwritten)

→ Lost Update

$T_1(B, 100)$	$T_2(B, 500)$
begin	
read(B)	
$B = B + 500$	
write(B)	
begin	
read(B) $\rightarrow 1,500$	
$B = B + 100 \rightarrow 1,600$	
write(B)	
commit	
abort	

Final balance $B = 1,600$
(when it should be $1,100$)

→ Dirty Read

$T_1(B, 100)$	$T_2(B, 500)$
begin	
read(B) $\rightarrow 1,000$	
begin	
read(B)	
$B = B + 500$	
write(B)	
commit	
read(B) $\rightarrow 1,100$	
...	

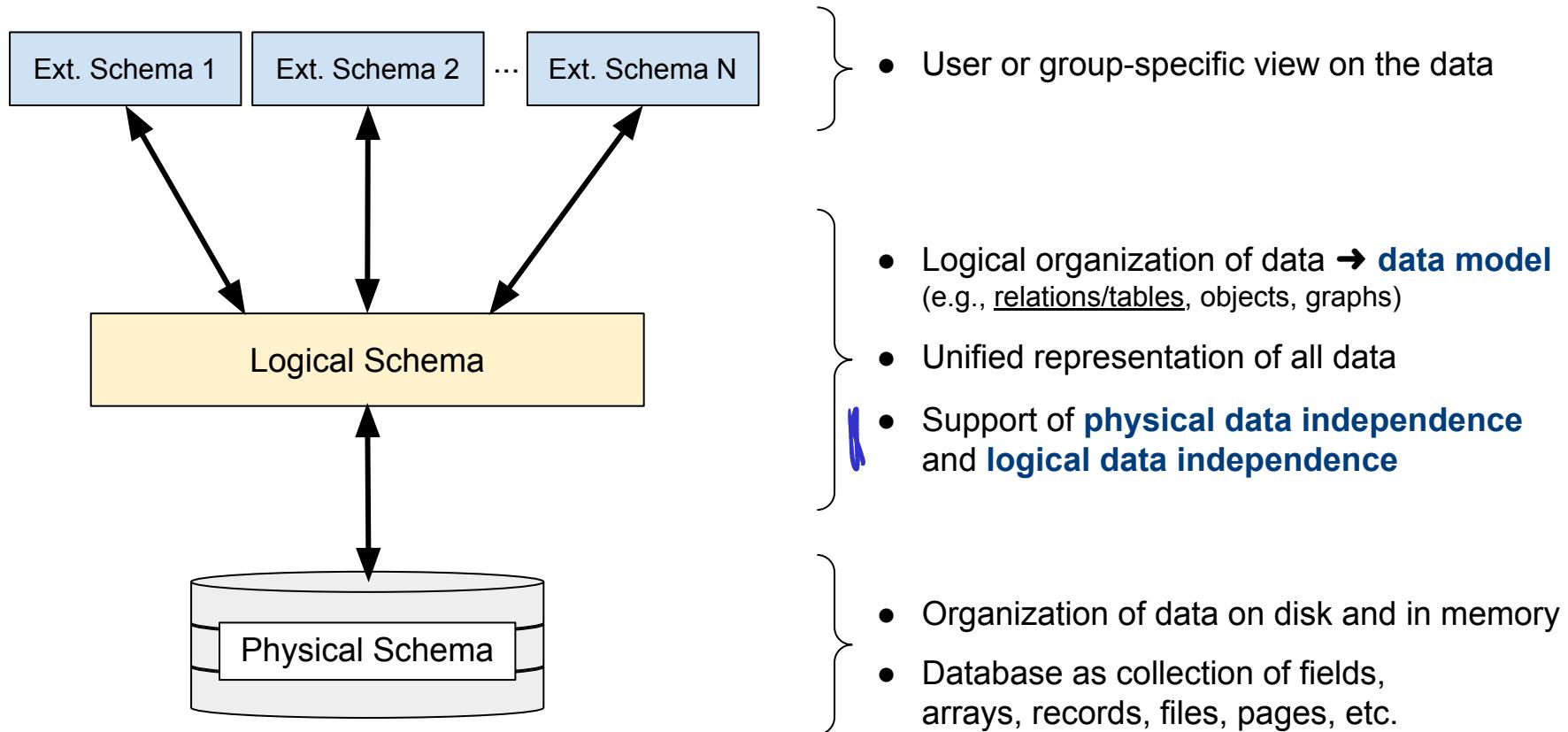
Balance B is retrieved twice
but the values differ

→ Unrepeatable Read

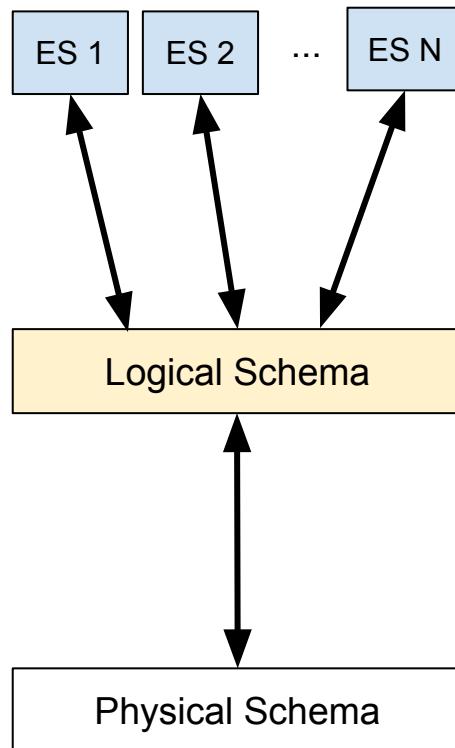
Requirement for Concurrent Transactions: Serializability

- **Serializable transaction execution**
 - A concurrent execution of a set of transactions is **serializable** if this execution is equivalent to some serial execution of the same set of transactions
 - Two executions are equivalent if they have the same effect on the data
- **Core tasks of DBMS**
 - **Support concurrent executions** of transactions to optimize performance
 - **Enforce serializability** of concurrent executions to ensure integrity of data

3-Tier Architecture of DBMS — Levels of Data Abstraction



Data Independence



Logical data independence (with limit)

- Ability to change logical schema without affecting external schemas (e.g., adding/deleting/updating attributes, changing data types, changing data model)

Physical data independence ✓

- Representation of data independent from physical scheme
- Physical schema can be changed without affecting logical schema (e.g., creating indexes, new caching strategies, different storage devices)

Study of DBMS — Scope of CS2102

- **Database design**

- How to model the data requirements
- How to organize data using a DBMS

Topics covered in CS2102

{
 Relation Model
 ER Model
 Schema Refinement

- **Database programming**

- How to create, query and update a database
- How to specify data constraints
- How to use SQL in applications

{
 Relational Algebra
 SQL

- **DBMS implementation**

- How to build a DBMS?

Describing Data in a DBMS

- **Data Model**

- Set concepts for describing the data
- Framework to specify structure of a DB

- **Schema**

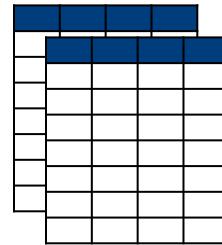
- Description of the structure of a DB using the concepts provided by the data model

- **Schema Instance**

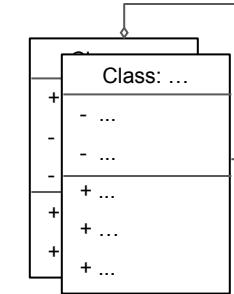
- Content of a DB at a particular time

RDBMS

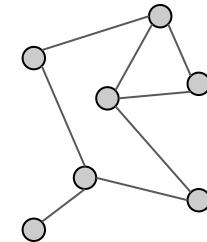
Tables



Objects



Graphs



Employees (id: **integer**, name: **text**, dob: **date**, salary: **numeric**)

Table "Employees"

ID	Name	DOB	Salary
1	Alice	10-08-1988	7,500
2	Bob	06-11-2001	4,800
3	Carol	25-02-1995	5,500

Overview

- Why Database Management Systems (DBMS)?
 - Challenges for data-intensive applications
 - From file-based data management to DBMS
 - Core concepts of DBMS (transactions, data abstraction)
- Relational Database Model
 - Motivation & history
 - Core concepts: relation, domain, schema, etc.
 - Integrity constraints
- Summary

Timeline of DBMS (Regarding the Supported Data Model)

- "Historical" models
 - Hierarchical model
 - Network model

- Relational Model

(early:prototypes 1970+, commercial products: 1980+)

- Commercial RDBMS
- Open-source RDBMS

- Object-oriented model

- Native OO model (e.g., Objectstore, 1988)
- Object-relational model
(now supported by most RDBMS)

- More recent development

- NoSQL models, in-memory DBMS
(e.g.. Cassandra, 2008; MongoDB, 2009; Redis, 2009)

Commercial systems*



Open-source systems



* some vendors offer free versions with limited functionalities

RDBMS (still) Reign Supreme

424 systems in ranking, June 2025

Rank			DBMS	Database Model	Score		
Jun 2025	May 2025	Jun 2024			Jun 2025	May 2025	Jun 2024
1.	1.	1.	Oracle	Relational, Multi-model 	1230.38	+3.82	-13.70
2.	2.	2.	MySQL	Relational, Multi-model 	953.57	-11.41	-107.77
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model 	776.75	+1.86	-44.81
4.	4.	4.	PostgreSQL 	Relational, Multi-model 	680.65	+6.34	+44.41
5.	5.	5.	MongoDB 	Document, Multi-model 	402.85	+0.33	-18.23
6.	6.	↑ 8.	Snowflake	Relational	174.49	+2.48	+44.13
7.	7.	↓ 6.	Redis	Key-value, Multi-model 	151.72	-0.47	-4.22
8.	8.	↑ 9.	IBM Db2	Relational, Multi-model 	125.13	-1.27	-0.77
9.	9.	↓ 7.	Elasticsearch	Multi-model 	121.28	-2.53	-11.55
10.	10.	10.	SQLite	Relational	117.03	-0.74	+5.63

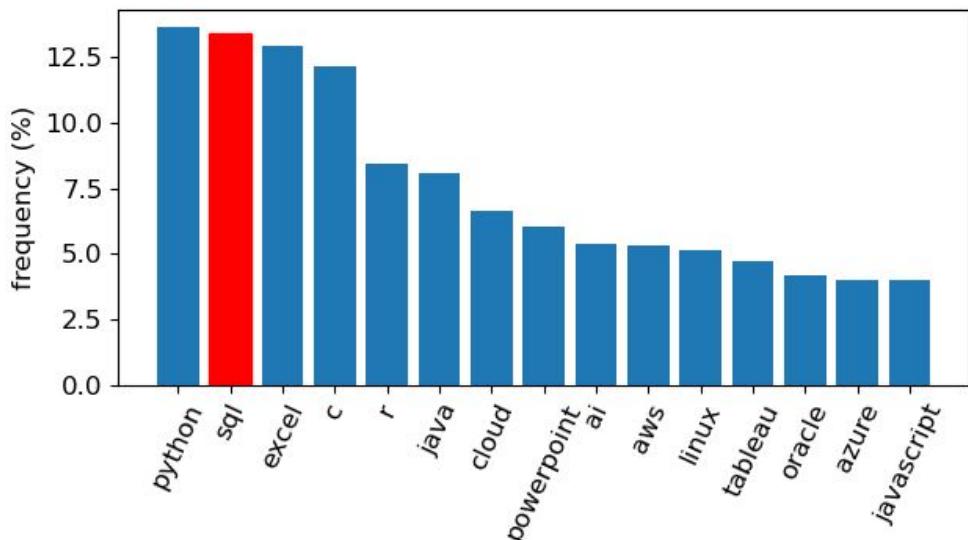
RDBMS (still) Reign Supreme

Java, SQL and Python are the most in-demand digital skills

Key Skill: SQL, Because Companies are Obsessed with Data

Want a Job in Data? Learn SQL.

- Analysis of job descriptions
 - 15k+ job offers from JobStreet
(data analyst, data engineer, data scientist)
 - Quick-&-dirty keyword extraction
 - ...but check for yourself! :)



Overview

- Why Database Management Systems (DBMS)?
 - Challenges for data-intensive applications
 - From file-based data management to DBMS
 - Core concepts of DBMS (transactions, data abstraction)
- Relational Database Model
 - Motivation & history
 - **Core concepts: relation, domain, schema, etc.**
 - Integrity constraints
- Summary

The Relational Model

- Proposed by Edgar F. Codd in 1970
- Basic concept: relations
("tables" with rows and columns)

Table "Employees"

id	name	dob	salary
1	Alice	10-08-1988	7,500
2	Bob	06-11-2001	4,800
3	Carol	25-02-1995	5,500

- Relation schema: definition of a relation
 - Specifies attributes (columns) and data constraints (e.g., domain constraints)

Employees (id: **integer**, name: **text**, dob: **date**, salary: **numeric**)

A Relational Model of Data for Large Shared Data Banks

E. F. CODD

IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.

The Relational Model

- **Domain** — set of atomic values (e.g., integer, numeric, text)
 - $\text{dom}(A_i)$ domain of attribute A_i = set of possible values of A_i
 - Each value v of attribute A_i : $v \in \text{dom}(A_i)$ or $v = \text{null}$
 - null — special value indicating the v is not known or not specified
↳ pain!
- **Relation** — set of tuples (or records)
 - $R(A_1, A_2, \dots, A_n)$ — relation schema with name R and n attributes A_1, A_2, \dots, A_n
 - Each instance of schema R is a relation which is a subset of
$$\{(a_1, a_2, \dots, a_n) \mid a_i \in \text{dom}(A_i) \cup \{\text{null}\}\}$$

Example

- Relational schema: Courses(code, mc, exam) with
 - $\text{dom}(\text{code}) = \{\text{cs2102}, \text{cs3223}, \text{cs4221}\}$
 - $\text{dom}(\text{mc}) = \{2, 4\}$
 - $\text{dom}(\text{exam}) = \{\text{yes}, \text{no}\}$

- Each instance of "Courses" is a subset of

$\{\text{cs2102}, \text{cs3223}, \text{cs4221}, \text{null}\} \times \{2, 4, \text{null}\} \times \{\text{yes}, \text{no}, \text{null}\}$

$$\underbrace{4}_{\text{4}} \times \underbrace{3}_{\text{3}} \times \underbrace{3}_{\text{3}} = \text{max. 36 tuples}$$

code	mc	exam
cs2102	2	yes ↗
cs2102	2	no
cs2102	4	yes
cs2102	4	no
cs3223	2	yes
...
null	4	no
null	null	no
null	null	null

Quick Quiz

- Assume a relation $R(A, B)$ with
 - $\text{dom}(A) = \{x, y, z\}$
 - $\text{dom}(B) = \{1, 2, 3, 4\}$

Which tuples in the table on the right **violate** the definition of relation R ?

	A	B
1:	x	4
2:	z	4
3:	null	2
4:	null	0
5:	y	1
6:	y	null
7:	null	null
8:	x	4
9:	x	y
10:	z	1
11:	x	1

Annotations on the table:
Row 4: A blue wavy line under "null" and "0" with the handwritten note "0 ∉ dom(B)".
Row 9: A blue wavy line under "x" and "y" with the handwritten note "y ∉ dom(A)".

The Relational Model

- **Relational database schema** — set of relation schemas + data constraints

Movies (id: **integer**, title: **text**, genre: **text**, opened: **date**)

Cast (movie_id: **integer**, actor_id: **integer**, role: **text**)

Actors (id: **integer**, name: **text**, dob: **date**)

Instance of Schema

- **Relational database** — collection of tables

Table "Movies"

id	title	genre	opened
101	Aliens	action	1986
102	Logan	drama	2017
103	Heat	crime	1995
104	Terminator	action	1984
...	

Table "Cast"

movie_id	actor_id	role
101	20	Ellen Ripley
101	23	Private Hudson
101	54	Corporal Hicks
102	21	Logan
104	23	Punk Leader
...

Table "Actors"

id	name	dob
20	Sigourney Weaver	08-10-1949
21	Hugh Jackman	12-10-1968
22	Tom Hanks	09-07-1956
23	Bill Paxton	17-05-1955
...

Challenge: Ensuring Data Integrity

- The definition $R(A_1, A_2, \dots, A_n) \subseteq \{(a_1, a_2, \dots, a_n) \mid a_i \in \text{dom}(A_i) \cup \{\text{null}\}\}$ allows:

Table "Movies"

id	title	genre	opened
101	Aliens	action	1986
101	Logan	drama	2017
103	Heat	crime	1995
104	Terminator	action	1984

Table "Cast"

movie_id	actor_id	role
101	20	Ellen Ripley
101	23	Private Hudson
101	54	Corporal Hicks
102	21	Logan
abc	23	Punk Leader

Table "Actors"

id	name	dob
20	Sigourney Weaver	08-10-2049
21	Hugh Jackman	12-10-1968
null	Tom Hanks	09-07-1956
23	Bill Paxton	17-05-1955

Can we tell the DBMS what are valid tuples and attribute values?

→ Integrity Constraints

Overview

- Why Database Management Systems (DBMS)?
 - Challenges for data-intensive applications
 - From file-based data management to DBMS
 - Core concepts of DBMS (transactions, data abstraction)
- Relational Database Model
 - Motivation & history
 - Core concepts: relation, domain, schema, etc.
 - **Integrity constraints**
- Summary

Integrity Constraints

- **Integrity Constraint** — condition that restricts what constitutes valid data
 - DBMS checks that tables only ever contain valid data → data integrity
- 3 main structural integrity constraints of the Relation Model
("structural" = inherent to the data model, independent from the application)
 - Domain constraints (e.g., cannot store a string in a integer column)
 - Key constraints
 - Foreign key constraints
- General constraints
 - Depend on the specific application
 - Covered in later lectures (keyword: triggers)

Table "Cast"

movie_id	actor_id	role
101	20	Ellen Ripley
101	23	Private Hudson
102	21	Logan
abc	23	Punk Leader

Key Constraints

- **Superkey** — subset of attributes that uniquely identifies a tuple in a relation
 - e.g., {id, title}, {id, title, opened}
- **Key** — superkey that is also minimal, i.e., no proper subset of the key is a superkey
 - e.g., {id} (maybe: {title, opened})
- **Candidate keys** — set of all keys for a relation
- **Primary key** — selected candidate key for a relation
 - Important: values of primary key attributes cannot be *null* (entity integrity constraint)

Movies (id: **integer**, title: **text**, genre: **text**, opened: **date**)

Table "Movies"

id	title	genre	opened
101	Aliens	action	1986
102	Logan	drama	2017
103	Heat	crime	1995
104	Terminator	action	1984

Quick Quiz

Assume a forum database with the following relation filled with many thousands of users:

Accounts (email: **text**, password: **text**, name: **text**)

Which subsets of attributes are a

- **Superkey**
- **Key**

of relation "Accounts"?

A {email} *Superkey* *Key*

B {password}

C {name}

D {email, password} ✓

E {email, name} ✓

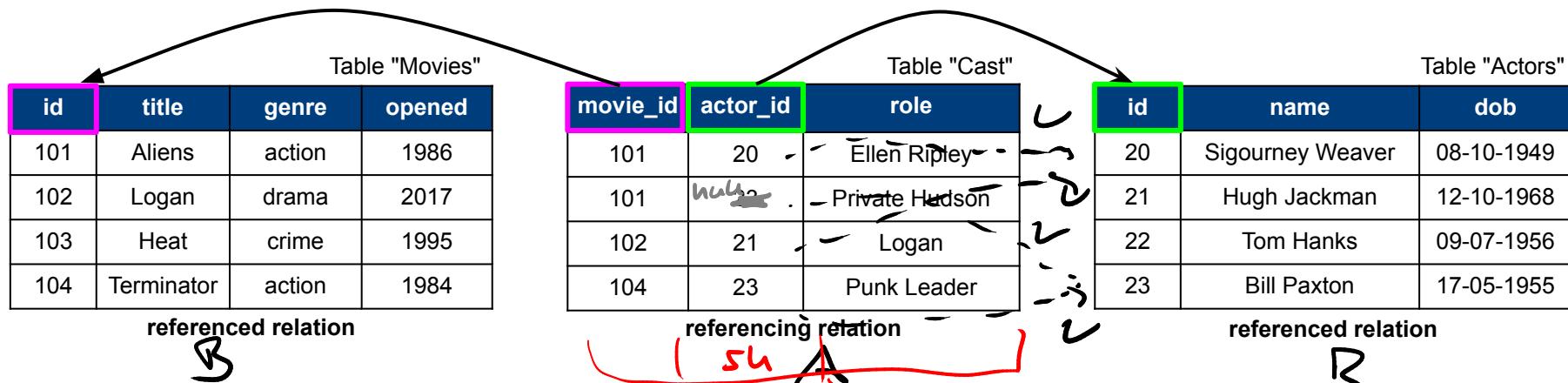
F {password, name}

G {email, password, name} ✓

Foreign Key Constraints

(also: referential integrity constraints)

- **Foreign key** — subset of attributes of relation A if it refers to the primary key in a relation B



- Requirement: each foreign key in referencing relation must
 - appear as primary key in referenced relation OR
 - be a *null* value

Foreign Key Constraints

- Referencing & referenced relation can be the same relation
 - Example: each employee has at most one manager

Table "Employees"

id	name	dob	salary	manager
1	Alice	10-08-1988	7,500	null
2	Bob	06-11-2001	4,800	3
3	Carol	25-02-1995	5,500	1
4	Dave	18-06-1999	6,000	null
5	Erin	09-05-2000	5,000	1

- A relation can be referencing and referenced relation for different relations

Table "Genre"

genre	description
action	exciting stuff
drama	suspenseful stuff
crime	mysterious stuff

Table "Movies"

id	title	genre	opened
101	Aliens	action	1986
102	Logan	drama	2017
103	Heat	crime	1995
104	Terminator	action	1984

Table "Cast"

movie_id	actor_id	role
101	20	Ellen Ripley
101	23	Private Hudson
102	21	Logan
104	23	Punk Leader

Quick Quiz

relations

- Assume the two tables $R(A, B)$ and $S(C, D)$ with
 - $\text{dom}(A) = \text{dom}(D) = \{w, x, y, z\}$
 - $\text{dom}(B) = \{1, 2, 3, 4\}$
 - $\text{dom}(C) = \{a, b, c, d\}$
 - Foreign key constraint $S.D \rightarrow R.A$

Which tuples in the tables on the right **violate** any **key** and/or **foreign key** constraints?

Table "R"

	A	B
R1:	x	4
R2:	z	4
R3:	null	2
R4:	y	null
R5:	x	1

Table "S"

	C	D
S1:	d	null
S2:	a	w
S3:	b	x
S4:	c	x
S5:	d	y

Integrity Constraints

- Limitations

- Structural integrity constraints do not cover application-independent constraints (e.g., limiting the domain to valid values)
- Not covered: application-dependent constraints derived from deeper semantics of the data

- Practical considerations

- Integrity constraints are optional, not mandatory (but they allow pushing checks from the application into the DBMS)
- Integrity constraints may affect performance* (checking constraints require additional processing steps)

Table "Actors"

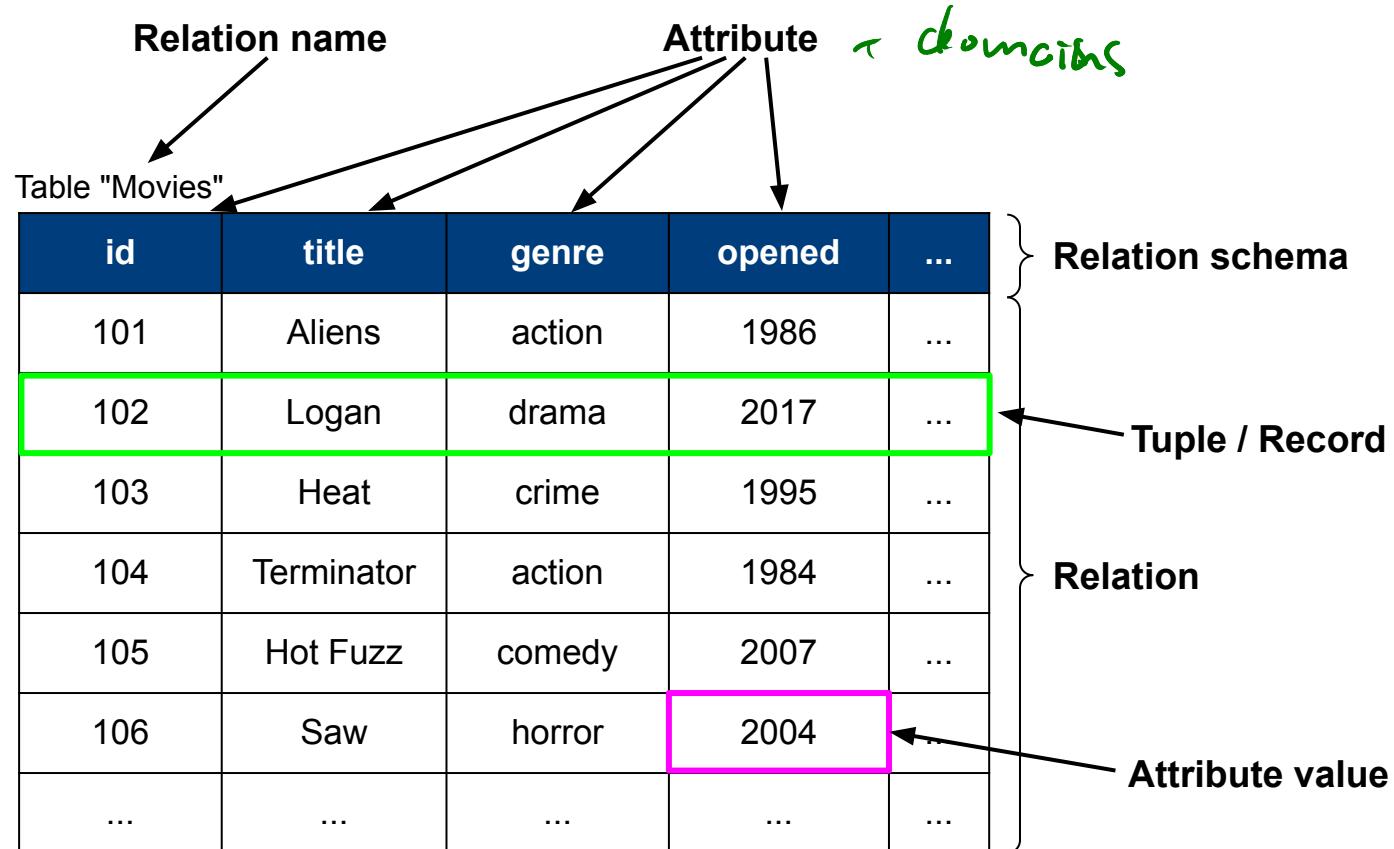
id	name	dob
20	Sigourney Weaver	08-10-2049
21	Hugh Jackman	12-10-1968
22	Tom Hanks	09-07-1956
23	Bill Paxton	17-05-1955

***Sidenote:** Key constraints typically involve the creation indexes which can significantly boost query performance!

Overview

- Why Database Management Systems (DBMS)?
 - Challenges for data-intensive applications
 - From file-based data management to DBMS
 - Core concepts of DBMS (transactions, data abstraction)
- Relational Database Model
 - Motivation & history
 - Core concepts: relation, domain, schema, etc.
 - Integrity constraints
- Summary

Relational Model — Cheat Sheet



Relational Model — Cheat Sheet

Term	Description (informal)
attribute	Column of a table
domain	Set of possible values for an attribute
attribute value	Element of a domain
relation schema	Set of attributes (with their data types + relation name)
relation	Set of tuples
tuple	Row of a table
database schema	Set of relation schemas
database	Set of relations / tables

Relational Model — Cheat Sheet

Term	Description (informal)
(candidate) key	Minimal set of attributes that uniquely identify a tuple in a relation
primary key	Selected key (in case of multiple candidate keys)
foreign key	Set of attributes that is a key in referenced relation
prime attribute	Attribute of a (candidate) key

- Terminology: DB. vs DBS vs. DBMS

$$\text{DBS} = \text{DBMS} + n^* \text{DB} \quad (n > 0)$$

Summary

- **Advantages of DBMS for large-scale data management** (compared to "files only")
 - Transactions with ACID properties to guarantee integrity of the data
 - Levels of abstraction for data independence
- **Relational Model**
 - Unified representation of all data as tables (relations)
 - (Structural) integrity constraints to specify restrictions on what constitutes correct/valid data
- **Outlook for next lecture:**
 - Creating and modifying databases with SQL
 - Describe database schemas and integrity constraints

Quick Quiz Solutions

Quick Quiz (Slide 26)

- Solution
 - Tuple 4 has a (non-NULL) value for B that is not in the domain of B
 - Tuple 9 has a (non-NULL) value for B that is not in the domain of B
 - Tuple 1 and 8 are duplicates which are not allowed in sets

Quick Quiz (Slide 32)

- Solution
 - Superkeys: any subset of attributes containing "email"
 - Key: {email}
- Additional comments
 - We assume that we allow duplicate names; some forums might require unique user names in which case {name} would also be a valid key
 - Don't forget that keys may contain more than 1 attribute; this is just a small example

Quick Quiz (Slide 35)

- Solution
 - Tuples R1 and R5 have duplicate values for the primary key A (violates key constraint)
 - Tuples S1 and S5 have duplicate values for the primary key C (violates key constraint)
 - Tuple R3 has a NULL value of its primary key (violates primary key)
 - Tuple S2 has a non-NULL values for attribute D that is not an existing primary key value in relation R (violates foreign key constraint)

CS2102: Database Systems

Lecture 2 — SQL (Part 1)

Course Logistics

- **Project registration**
 - Canvas Groups with self sign-up
 - Canvas → CS2102 → People → Project (tab) → Project 1-125
 - Group size: 4 (members do not have to be in the same tutorial)
 - Use Canvas Discussion to look for members or team (random assignment to groups after Friday, 17:00)
- **Tutorials**
 - Appeals regarding allocation to be done on CourseReg
 - Mandatory attendance (you can skip up to 2 tutorials without penalty)
 - We expect students to come prepared (Check out the questions before coming to the tutorial!)

Quick Recap: Relational DBMS (RDBMS)

- RDBMS = DBMS + Relational Model
 - Unified representation of all data as relations (tables)
 - Integrity constraints to specify restrictions on what constitutes correct/valid data
 - Transactions with ACID properties to guarantee integrity of the data
 - Levels of abstraction for data independence

A Relational Model of Data for Large Shared Data Banks

E. F. CODD

IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.

Table "Employees"

id	name	dob	salary
1	Alice	10-08-1988	7,500
2	Bob	06-11-2001	4,800
3	Carol	25-02-1995	5,500

Quick Clarifications

- Terminology: "key" vs. "candidate key"
 - Same concept; terms used interchangeably
("candidate" highlights that there might be more than one key)
 - Additionally: (candidate) keys cannot be *null*
(otherwise they could not serve as chosen primary key)

Overview

- **SQL — overview**
 - History and usages
 - SQL language groups
- **Creating a database with SQL**
 - Basic DDL & DML commands
 - Defining integrity constraints
 - Advanced: deferrable constraints
- **Modifying a database with SQL**
 - Basic DDL commands

SQL — Structured Query Language

- De-facto standard language to "talk" to a RDBMS: **SQL**
 - Developed Donald D. Chamberlin and Raymond F. Boyce (IBM Research, 1974)
 - Originally called SEQUEL (Structured English Query Language)
 - SQL is not a general-purpose language (such as Python, Java, C++, etc.) but a **domain-specific language**
 - SQL is a declarative language: focus on *what* to compute, not on *how* to compute
- **SQL Standard**
 - First standard: SQL-86; most recent standard: SQL-2019 (new standard every ~3-5 years)
 - New standards introduce new language concepts (e.g., support new features of RDBMS)
 - Many RDBMS add their own "flavor" to SQL

Using SQL

- Interactive SQL: directly writing SQL statements to an interface

- Command line interface
 - e.g., PostgreSQL's **psql** [1]

```
List of relations
Schema | Name   | Type  | Owner
public  | aircrafts | table | postgres
public  | airports  | table | postgres
public  | countries | table | postgres
public  | flightcodes | table | postgres
public  | flights   | table | postgres
(5 rows)

flightsdb=# SELECT code, name FROM aircrafts LIMIT 3;
code | name
-----
75D | Boeing 757-200
737 | Boeing 737
E75 | Embraer 175 (short wing)
(3 rows)

flightsdb=#
```

The screenshot shows the pgAdmin 4 interface. On the left, the 'Browser' panel displays a tree view of database objects under 'Servers (1) > vdw-com2 > Databases (4) > flightsdb'. Under flightsdb, nodes include 'code', 'name', 'Schemas (1)', and 'Subscriptions'. The 'Query Editor' tab is active, showing the following SQL query:

```
1 SELECT code, name
2 FROM aircrafts
3 LIMIT 3;
```

The 'Data Output' tab is selected, displaying the results of the query:

code	name
75D	Boeing 757-200
737	Boeing 737
E75	Embraer 175 (short wing)

[1] <https://www.postgresql.org/docs/current/static/app-psql.html>

[2] <https://www.pgadmin.org/>

Using SQL

- Non-interactive
 - SQL statements are included in an application written in a host language
 - Two basic approaches to include SQL in host languages: SLI & CLI
- Statement Level Interface (SLI)
 - Application is a mixture of host language statements and SQL statements
 - Examples: Embedded SQL, Dynamic SQL
- Call Level Interface (CLI)
 - Application is completely written in host language
 - SQL statements are strings passed as arguments to host language procedures or libraries
 - Examples: ODBC (Open DataBase Connectivity), JDBC (Java DataBase Connectivity)

Statement Level Interface (SLI) — Example

```
int main()
{
    EXEC SQL WHENEVER NOT FOUND DO BREAK;
    EXEC SQL BEGIN DECLARE SECTION;
    char v_code[32], v_name[32];
    EXEC SQL END DECLARE SECTION;

    // Connect to database
    EXEC SQL BEGIN DECLARE SECTION;
    const char *target = "flightsdb@localhost";
    const char *user = "postgres";
    const char *passwd = "_____";
    EXEC SQL END DECLARE SECTION;
    EXEC SQL CONNECT TO :target USER :user USING :passwd;

    // Declare cursor
    EXEC SQL DECLARE c CURSOR FOR
    SELECT code, name FROM aircrafts LIMIT 3;

    // Open cursor
    EXEC SQL OPEN c;

    // Loop through cursor and display results
    for(;;) {
        EXEC SQL FETCH NEXT FROM c INTO :v_code, :v_name;
        printf(">>> code: %s, name: %s\n", v_code, v_name);
    }

    // Cleanup (close cursor, commit, disconnect)
    EXEC SQL CLOSE c;
    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT;

    return 0;
}
```

```
#!/bin/bash

# Run ecpg preprocessor to convert C program with embedded SQL statements
# to normal C code; replaces the SQL invocations with special function calls.
ecpg flightsdb.pgc

# Compile generated C code; requires to include all header files the compiler
# needs to understand the special function calls (files come with PostgreSQL).
gcc -g -I/usr/include/postgresql -c flightsdb.c

# Build output to executable file; also needs access to the header files.
gcc -o flightsdb flightsdb.o -L/usr/include/postgresql -lecpq
```

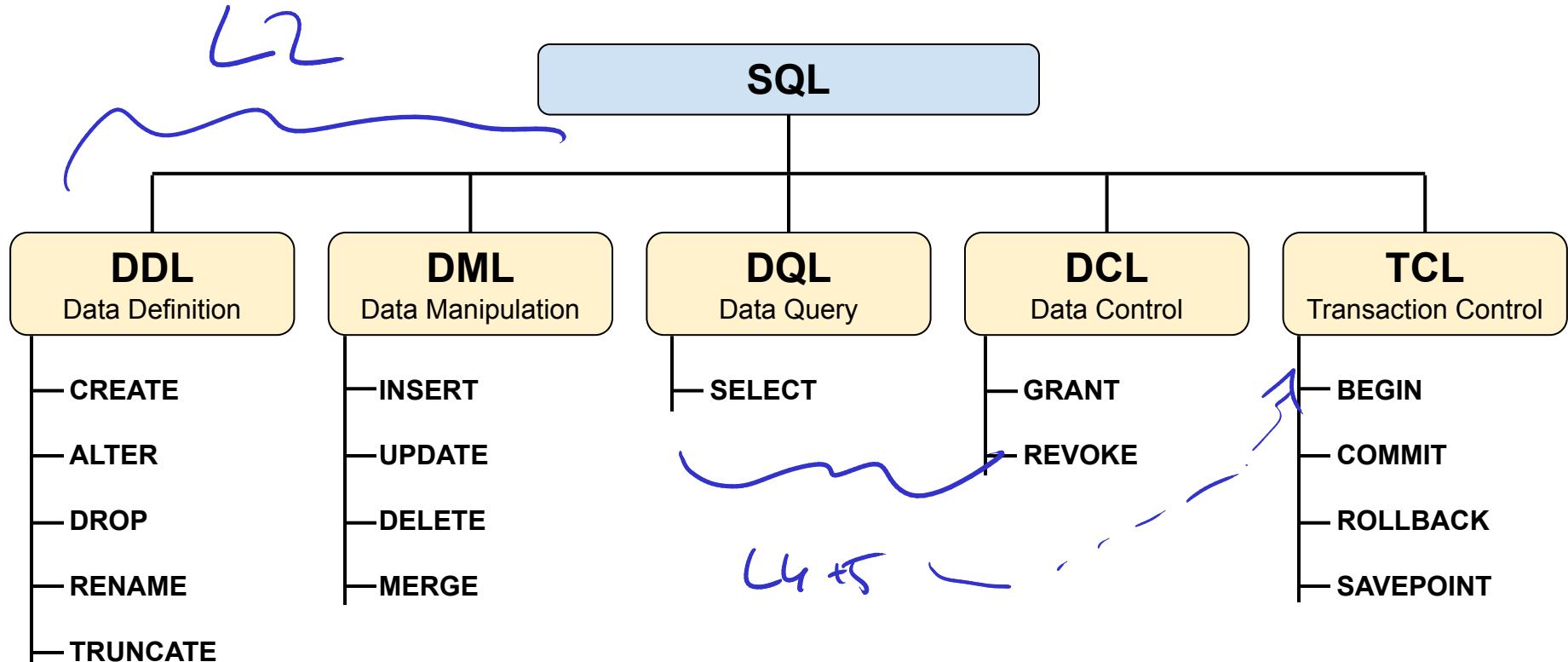
```
>>> code: 75D, name: Boeing 757-200
>>> code: 737, name: Boeing 737
>>> code: E75, name: Embraer 175 (short wing)
```

Call Level Interface (CLI) — Example

```
1 import psycopg2 # Host language library (here psycopg2 for Python)
2
3 # Connect to database
4 db = psycopg2.connect(host="localhost", database="flightsdb", user="postgres", password="████████")
5
6 # Create cursor
7 cursor = db.cursor()
8
9 # Open cursor by executing query (string parameter passed to execute() method)
10 cursor.execute("SELECT code, name FROM aircrafts LIMIT 3")
11
12 # Loop over all results until no next tuple is returned
13 while True:
14     row = cursor.fetchone()
15     if row is None:
16         break
17     print(row)
18
19 # Cleanup
20 cursor.close()
21 db.close()
```

```
('75D', 'Boeing 757-200')
('737', 'Boeing 737')
('E75', 'Embraer 175 (short wing)')
```

SQL — Types of Commands/Statements



Overview

- SQL — overview
 - History and usages
 - SQL language groups
- Creating a database with SQL
 - Basic DDL & DML commands
 - Defining integrity constraints
 - Advanced: deferrable constraints
- Modifying a database with SQL
 - Basic DDL commands

DDL — Creating Tables

- Basic syntax: definition of table name and attributes (with data types)

Employees (id: **integer**, name: **text**, age: **integer**, role: **text**)



```
CREATE TABLE Employees (
    id      INTEGER,
    name   VARCHAR(50),
    age    INTEGER,
    role   VARCHAR(50)
);
```

- Extended syntax: definition of additional data integrity constraints

Data Types (PostgreSQL)

- Basic data types

(supported by most RDBMS)

boolean	logical Boolean (true/false)
integer	signed four-byte integer
float8	double precision floating-point number (8 bytes)
numeric [(p,s)]	exact numeric of selectable precision
char(n)	fixed-length character string
varchar(n)	variable-length character string
text	variable-length character string
date	calendar date (year, month, day)
timestamp	date and time

} *string*

- Many extended data types

- Document types: XML, JSON
- Spatial types: point, line, polygon, circle, box, path
- Special types: money/currency, MAC/IP address

- Definition user-defined types (UDTs)

DML — Inserting Data (Basic Examples)

```
CREATE TABLE Employees (
    id   INTEGER,
    name VARCHAR(50),
    age  INTEGER,
    role VARCHAR(50)
);
```

- Example: Inserting 3 employees

- Specifying all attribute values

```
INSERT INTO Employees VALUES (101, 'Sarah', 25, 'dev');
```

(id, name, age, role) all attr.

- Specifying selected attribute values

```
INSERT INTO Employees (id, name) VALUES (102, 'Judy'), (103, 'Max');
```



Employees

id	name	age	role
101	Sarah	25	dev
102	Judy	null	null
103	Max	null	null

DML — Inserting Data (Basic Examples)

- Example: Inserting 3 employees

- Specifying all attribute values

```
INSERT INTO Employees VALUES (101, 'Sarah', 25, 'dev');
```

- Specifying selected attribute values

```
INSERT INTO Employees (id, name) VALUES (102, 'Judy'), (103, 'Max');
```



Employees

id	name	age	role
101	Sarah	25	dev
102	Judy	null	<u>sales</u>
103	Max	null	<u>sales</u>

```
CREATE TABLE Employees (
    id      INTEGER,
    name   VARCHAR(50),
    age    INTEGER,
    role   VARCHAR(50) DEFAULT 'sales'
);
```

DML — Deleting Data (Basic Examples)

-- Delete all tuples

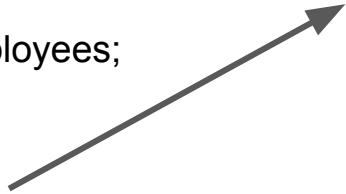
DELETE FROM Employees;

Employees

id	name	age	role
----	------	-----	------

Employees

id	name	age	role
101	Sarah	25	dev
102	Judy	null	sales
103	Max	null	sales

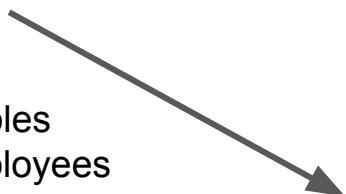


-- Delete selected tuples

DELETE FROM Employees
WHERE role = 'dev';

Employees

id	name	age	role
102	Judy	null	sales
103	Max	null	sales



DML — Updating Data (Basic Examples)

DELETE From Em
WHERE
age < 18
IS NULL

Employees

id	name	age	role
101	Sarah	25	dev
102	Judy	null	sales
103	Max	null	sales

-- Sarah's birthday
UPDATE Employees
SET age = age + 1
WHERE name = 'Sarah';



-- New privacy law
UPDATE Employees
SET age = 0;



Employees

id	name	age	role
101	Sarah	26	dev
102	Judy	null	sales
103	Max	null	sales



Employees

id	name	age	role
101	Sarah	0	dev
102	Judy	0	sales
103	Max	0	sales

Employees

id	name	age	role
101	SARAH	25	DEV
102	JUDY	null	SALES
103	MAX	null	SALES

Overview

- SQL — overview
 - History and usages
 - SQL language groups
- Creating a database with SQL
 - Basic DDL & DML commands
 - **Defining integrity constraints**
 - Advanced: deferrable constraints
- Modifying a database with SQL
 - Basic DDL commands

Prerequisite — Handling *null* Values

- Recall: rules of handling *null* values
 - The result of a comparison operation with *null* is *unknown*
 - The result of an arithmetic operation with *null* is *null*

→ Three-valued logic: **true**, **false**, **unknown**

Assume that value of x is <i>null</i>	
$x < 2020$	→ <i>unknown</i>
$x = \text{null}$	→ <i>unknown</i>
$x \neq \text{null}$	→ <i>unknown</i>
$x + 5$	→ <i>null</i>

- Questions
 - How to check if a value is equal to *null*?
 - How to treat *null* values as ordinary values for comparison?

} Important for writing SQL queries & checking integrity constraints!

IS (NOT) NULL Comparison Predicate

- Check if a value is equal to null (since "`=`" would return unknown)
 - If x is a *null* value → " $x \text{ IS } \text{NULL}$ " evaluates to **true**
 - If x is a non-*null* value → " $x \text{ IS } \text{NULL}$ " evaluates to **false**
- Equivalence
 - " $x \text{ IS NOT } \text{NULL}$ " is equivalent to "**NOT** ($x \text{ IS } \text{NULL}$)"

`<>`

} vice versa for " $x \text{ IS NOT } \text{NULL}$ "

x	y
1	1
1	2
<i>null</i>	1
<i>null</i>	<i>null</i>



x	y	x IS NULL	y IS NULL
1	1	<i>false</i>	<i>false</i>
1	2	<i>false</i>	<i>false</i>
<i>null</i>	1	<i>true</i>	<i>false</i>
<i>null</i>	<i>null</i>	<i>true</i>	<i>true</i>

IS (NOT) NOT DISTINCT Comparison Predicate

- "x IS DISTINCT FROM y"

- equivalent to "x <> y" if x and y are non-null values
- if x and y both null → evaluates to **false**
- if only one value is null → evaluates to **true**

vice versa for "x IS NOT DISTINCT FROM y"

- Equivalence

- "x IS NOT DISTINCT FROM y" is equivalent to "NOT (x IS DISTINCT FROM y)"

x	y
1	1
1	2
null	1
null	null



x	y	x < <u>></u> y	x IS DISTINCT FROM y
1	1	FALSE	FALSE
1	2	TRUE	TRUE
null	1	null*	TRUE
null	null	null*	FALSE

* PostgreSQL represents "unknown" using null

DDL — Data Integrity Constraints: Overview

- **Types of Constraints** ("named" or "unnamed")

- Not-null constraints
- Unique constraints
- Primary key constraints
- Foreign key constraints
- General constraints



A constraint is violated
if it evaluates to **false**

- **Constraint specifications** (difference "where" a constraint is specified)

- Column constraint: applies to single column, specified at column definition
- Table constraint: applies to one or more columns, specified after all column definitions
- Assertion: stand-alone command (**create assertion ...**)

Not-Null Constraints

- Example: the id or name of an employee cannot be *null*

unnamed constraint (name assigned by DBMS)

```
CREATE TABLE Employees (
    id    INTEGER NOT NULL,
    name  VARCHAR(50) NOT NULL,
    age   INTEGER,
    role  VARCHAR(50),
);
```

named constraint (easier bookkeeping)

```
CREATE TABLE Employees (
    id    VARCHAR(50) CONSTRAINT nn_id NOT NULL,
    name  VARCHAR(50) CONSTRAINT nn_name NOT NULL,
    age   INTEGER,
    role  VARCHAR(50),
);
```

- Not-null constraint violation:
 - There exists a tuple $t \in \text{Employees}$ where " $t.\text{id} \text{ IS NOT NULL}$ " evaluates to **false**
 - There exists a tuple $t \in \text{Employees}$ where " $t.\text{name} \text{ IS NOT NULL}$ " evaluates to **false**

Unique Constraints

- Example: the id of an employee must be unique

unnamed column constraint

```
CREATE TABLE Employees (
    id   INTEGER UNIQUE,
    name VARCHAR(50),
    age  INTEGER,
    role  VARCHAR(50)
);
```

named column constraint

```
CREATE TABLE Employees (
    id   INTEGER CONSTRAINT u_id UNIQUE,
    name VARCHAR(50),
    age  INTEGER,
    role  VARCHAR(50)
);
```

unnamed table constraint

```
CREATE TABLE Employees (
    id   INTEGER,
    name VARCHAR(50),
    age  INTEGER,
    role  VARCHAR(50),
    UNIQUE (id)
);
```

named table constraint

```
CREATE TABLE Employees (
    id   INTEGER,
    name VARCHAR(50),
    age  INTEGER,
    role  VARCHAR(50),
    CONSTRAINT u_id UNIQUE (id)
);
```

Unique Constraints

- Unique constraint for more than one attribute / column
 - Can only be specified using table constraints
 - Example: Each pair of employee name and project name must be unique

Teams (eid: **integer**, pname: **text**, hours: **integer**)

unnamed table constraint

```
CREATE TABLE Teams (
    eid      INTEGER,
    pname   VARCHAR(100),
    hours   INTEGER,
    UNIQUE (eid, pname)
);
```



named table constraint

```
CREATE TABLE Teams (
    eid      INTEGER,
    pname   VARCHAR(100),
    hours   INTEGER,
    CONSTRAINT u_allocation UNIQUE (eid, pname)
);
```

Unique Constraints

Quick Quiz: Is the unique constraint of table "Teams" violated in the example below?

```
CREATE TABLE Teams (
    eid      INTEGER,
    pname   VARCHAR(100),
    hours   INTEGER,
    UNIQUE (eid, pname)
);
```

Teams

eid	pname	hours
101	BigAI	10
105	BigAI	5
102	GlobalDB	20
101	null	null
101	null	null
103	CoreOS	40
109	CoreOS	null

- Unique constraint violation

- For any two tuples $t_i, t_k \in \text{Teams}$:
- $"(t_i.\text{eid} <> t_k.\text{eid}) \text{ or } (t_i.\text{pname} <> t_k.\text{pname})"$ evaluates to **false**

True False or unknown \leftrightarrow unknown

Primary Key Constraints

Quick Quiz: What is the difference between using "primary key" and "unique not null"?

- Quick recap: primary key
 - Selected key uniquely identifying tuples in a table
 - Prime attributes (i.e. attributes of primary key) cannot be null

Employees (id: integer, name: **text**, age: **integer**, role: **text**)

```
CREATE TABLE Employees (
    id    INTEGER PRIMARY KEY,
    name  VARCHAR(50),
    age   INTEGER,
    role  VARCHAR(50)
);
```

```
CREATE TABLE Employees (
    id    INTEGER UNIQUE NOT NULL,
    name  VARCHAR(50),
    age   INTEGER,
    role  VARCHAR(50)
);
```

same effect

Primary Key Constraints

- Primary key constraint for more than one attribute / column

Teams (eid: integer, pname: text, hours: integer)

unhanded

```
CREATE TABLE Teams (
    eid      INTEGER,
    pname   VARCHAR(100),
    hours   INTEGER,
    PRIMARY KEY (eid, pname)
);
```

handed

```
CREATE TABLE Teams (
    eid      INTEGER,
    pname   VARCHAR(100),
    hours   INTEGER,
    CONSTRAINT pk_allocation PRIMARY KEY (eid, pname)
);
```

Sidenote

- Specification of constraints — basic rules

- All constraints can be specified "named" or "unnamed"
(unnamed constraints still get named by the DBMS in a meaningful way; names can be looked up)
- All column constraints can be specified as table constraints
(exception: "not null" only possible as column constraint)
- Table constraints referring to a single column can be specified as column constraint
- Column and table constraints can be combined (even w.r.t. to the same column)

```
CREATE TABLE Employees (
    id      INTEGER NOT NULL,
    name   VARCHAR(50),
    age    INTEGER,
    role   VARCHAR(50),
    UNIQUE (id)
);
```

Foreign Key Constraints

- Quick recap: foreign key constraint

- Subset of attributes of relation A if it refers to the primary key in a relation B

Employees (id: integer, name: text, age: integer, role: text, hired: date)

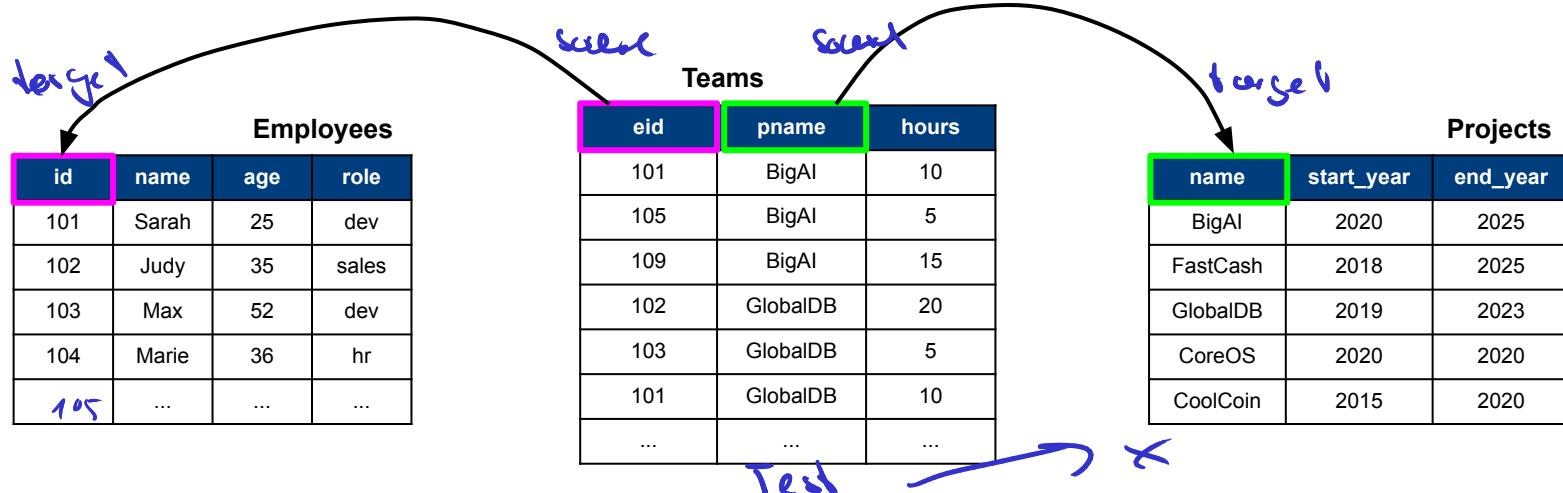
Teams (eid: integer, pname: text, hours: integer)

Projects (name: text, start_year: integer, end_year: integer)

Foreign key constraints

Teams.eid → Employees.id

Teams.pname → Projects.name



Foreign Key Constraints

```
CREATE TABLE Employees (
    id      INTEGER PRIMARY KEY,
    name   VARCHAR(50),
    age    INTEGER,
    role   VARCHAR(50)
);
```

```
CREATE TABLE Projects (
    name      VARCHAR(50) PRIMARY KEY,
    start_year INTEGER,
    end_year  INTEGER
);
```

```
CREATE TABLE Teams (
    eid        INTEGER,
    pname     VARCHAR(100),
    hours     INTEGER,
    PRIMARY KEY (ename, pname),
    FOREIGN KEY (eid) REFERENCES Employees (id),
    FOREIGN KEY (pname) REFERENCES Projects (name)
);
```

source *target*

Foreign Key Constraints — Violations

- Quick recap: each foreign key in referencing relation must
 - appear as primary key in referenced relation OR
 - be a *null* value

The diagram illustrates a foreign key constraint between the 'Teams' and 'Projects' tables. A curved arrow points from the 'pname' column in the 'Teams' table to the 'name' column in the 'Projects' table, indicating that the 'pname' values in 'Teams' must refer to valid 'name' values in 'Projects' or be null.

Teams		
eid	pname	hours
101	BigAI	10
105	BigAI	5
109	BigAI	15
102	GlobalDB	20
103	GlobalDB	5
101	GlobalDB	10
...

Projects		
name	start_year	end_year
BigAI	2020	2025
FastCash	2018	2025
GlobalDB	2019	2023
CoreOS	2020	2020
CoolCoin	2015	2020

Questions:

- What happens if the first tuple in "Project" should be deleted?
- What if the project "BigAI" should be renamed to "SmartAI"?

Note: Trying to insert or update a tuple in "Teams" with a new project name that is not in "Project" will always violate the foreign constraint.

Foreign Key Constraints — Violations

- Extend syntax to specify behavior when data in referenced table changes
 - Specify action in case of violation of a foreign key constraint
 - ON DELETE/UPDATE <action>** to distinguish action w.r.t. to a delete or update in referenced table
 - Both specifications are optional

```
CREATE TABLE Teams (
    eid      INTEGER,
    pname   VARCHAR(100),
    hours   INTEGER,
    PRIMARY KEY (ename, pname),
    FOREIGN KEY (eid) REFERENCES Employees (id) ON DELETE <action> ON UPDATE <action>,
    FOREIGN KEY (pname) REFERENCES Projects (name) ON DELETE <action> ON UPDATE <action>
);
```

Foreign Key Constraints — Violations

- Possible actions for **on delete** and **on update**

NO ACTION rejects delete/update if it violates constraint (default value)

RESTRICT similar to "no action" except that check of constraint cannot be deferred
(deferrable constraints are discussed in a bit)

CASCADE propagates delete/update to referencing tuples

SET DEFAULT updates foreign keys of referencing tuples to some default value
(important: default value must be a primary key in the referenced table!)

SET NULL updates foreign keys of referencing tuples to *null*
(important: corresponding column must be allowed to contain *null* values!)

Foreign Key Constraints

Quick Quiz: The SQL command below is correct but what will cause problems. Why?

```
CREATE TABLE Teams (
    eid      INTEGER,
    pname   VARCHAR(100),
    hours   INTEGER,
    PRIMARY KEY (eid, pname),
    FOREIGN KEY (eid) REFERENCES Employees (id) ON DELETE NO ACTION ON UPDATE CASCADE,
    FOREIGN KEY (pname) REFERENCES Projects (name) ON DELETE SET NULL ON UPDATE CASCADE
);
```

optional since it is the default action



- Effects on handling violations of foreign key constraints
 - Updates of "Employees.id" and "Projects.name" are propagated to affected tuples in "Teams"
 - Deleting a project will set "Teams.pname" to *null* for employees working on that project
 - Deleting an employee will raise an error if that employee is still assigned to a team

Foreign Key Constraints — Example

```
CREATE TABLE Teams (
    eid      INTEGER,
    pname   VARCHAR(100),
    hours   INTEGER,
    PRIMARY KEY (eid, pname),
    FOREIGN KEY (eid) REFERENCES Employees (id) ON UPDATE CASCADE,
    FOREIGN KEY (pname) REFERENCES Projects (name) ON UPDATE CASCADE
);
```

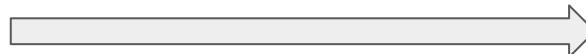
Projects

name	start_year	end_year
BigAI	2020	2025
FastCash	2018	2025
...

Teams

eid	pname	hours
101	BigAI	10
105	BigAI	5
109	BigAI	15
102	GlobalDB	20
...

UPDATE Projects
SET name = 'SmartAI'
WHERE name = 'BigAI';



Projects

name	start_year	end_year
SmartAI	2020	2025
FastCash	2018	2025
...

Teams

eid	pname	hours
101	SmartAI	10
105	SmartAI	5
109	SmartAI	15
102	GlobalDB	20
...

Foreign Key Constraints — Example

```
CREATE TABLE Teams (
    eid      INTEGER,
    pname    VARCHAR(100) DEFAULT 'FastCash', -- default value must be primary key in "Projects"!
    hours    INTEGER,
    PRIMARY KEY (eid, pname),
    FOREIGN KEY (eid) REFERENCES Employees (id) ON UPDATE CASCADE,
    FOREIGN KEY (pname) REFERENCES Projects (name) ON UPDATE CASCADE ON DELETE SET DEFAULT
);
```

Projects

name	start_year	end_year
BigAI	2020	2025
FastCash	2018	2025
...

Teams

eid	pname	hours
101	BigAI	10
105	BigAI	5
109	BigAI	15
102	GlobalDB	20
...

**DELETE FROM Projects
WHERE name = 'BigAI';**



Projects

name	start_year	end_year
FastCash	2018	2025
...

Teams

eid	pname	hours
101	FastCash	10
105	FastCash	5
109	FastCash	15
102	GlobalDB	20
...

Foreign Key Constraints

- Practical considerations
 - Specified constraints might not behave as expected (e.g., **SET NULL** issue with prime attributes)
 - Particularly **ON DELETE CASCADE** can have very bad consequences
 - **CASCADE** may significantly affect overall performance
- Careful design and specification of foreign key constraints is crucial!

Check Constraints

- **CHECK** constraint
 - Most basic general constraint (i.e., not a structural integrity constraint)
 - Allows to specify that column values must satisfy a Boolean expression
 - Scope: one table, single row
- Example: The hours an employee is allocated to a project must be > 0

```
CREATE TABLE Teams (
    eid      INTEGER,
    pname   VARCHAR(100),
    hours   INTEGER CHECK (hours > 0),
    -- hours  INTEGER CONSTRAINT positive_hours CHECK (hours > 0),
    PRIMARY KEY (eid, pname),
    FOREIGN KEY (eid) REFERENCES Employees (id),
    FOREIGN KEY (pname) REFERENCES Projects (name)
);
```

Check Constraints

- **CHECK** constraints can refer to multiple columns
 - Example: The start year of a project cannot be larger value than the end year

```
CREATE TABLE Projects (
    name      VARCHAR(50) PRIMARY KEY,
    start_year INTEGER,
    end_year   INTEGER,
    -- CHECK (start_year <= end_year),
    CONSTRAINT valid_lifetime CHECK (start_year <= end_year)
);
```

Check Constraints

- **CHECK** constraints can be arbitrarily complex Boolean expressions
 - Example: minimum hour requirements for different projects

```
CREATE TABLE Teams (
    eid      INTEGER,
    pname   VARCHAR(100),
    hours   INTEGER,
    PRIMARY KEY (eid, pname),
    FOREIGN KEY (eid) REFERENCES Employees (id),
    FOREIGN KEY (pname) REFERENCES Projects (name),
    CHECK (
        (pname = 'CoreOS' AND hours >= 30)
        OR
        (pname <> 'CoreOS' AND hours > 0)
    )
);
```

Assertions

- **CREATE ASSERTION** statement (since SQL-92)
 - Formulation of (almost) arbitrary constraints
 - Scope: multiple tables, multiple rows
 - Example: "Each project must have at least one team member being 30 or older"
 - Assertion in practice: various potential side effects and limitations, e.g.:
 - Assertions cannot modify the data
 - No proper error handling
 - Not linked to a specific table
(e.g., dropping a table does not affect assertion)
- Most RDBMS do not support assertions but **triggers** (more powerful alternative)

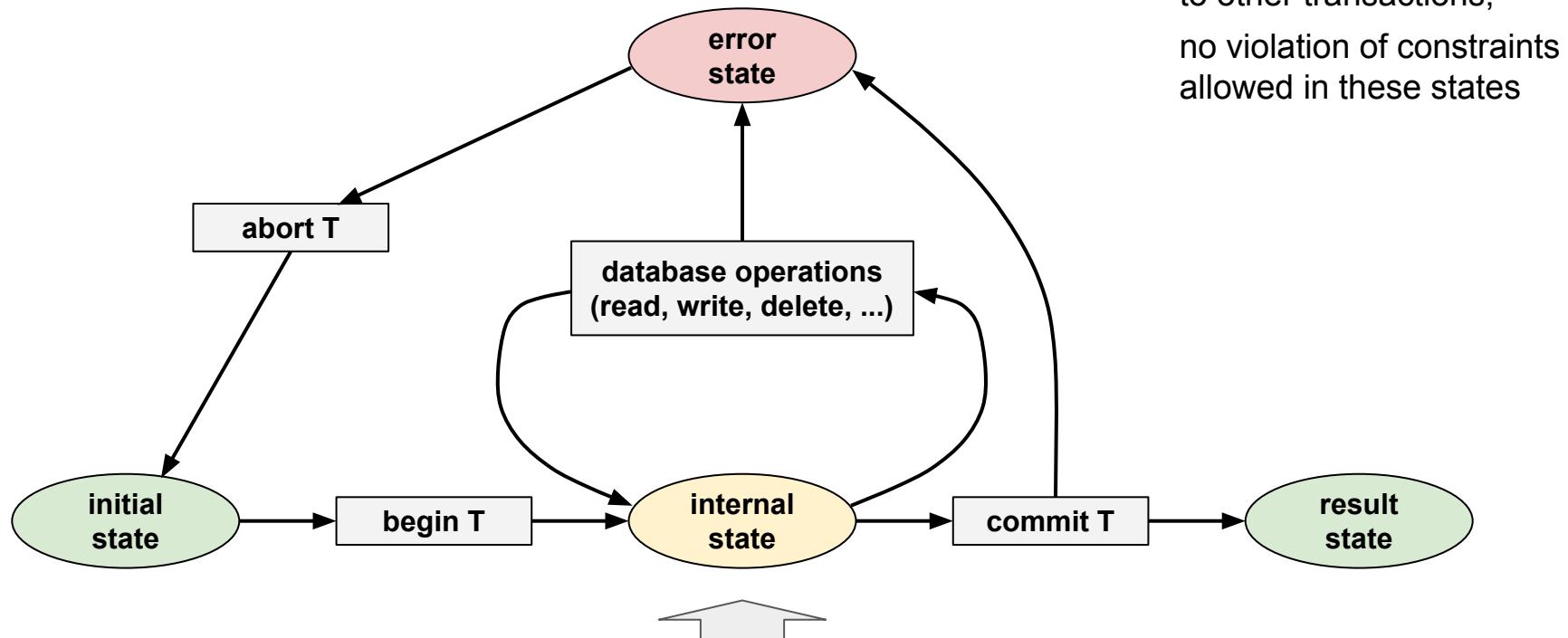
Overview

- SQL — overview
 - History and usages
 - SQL language groups
- Creating a database with SQL
 - Basic DDL & DML commands
 - Defining integrity constraints
 - **Advanced: deferrable constraints**
- Modifying a database with SQL
 - Basic DDL commands

Deferrable Constraints — Motivation

- Default behavior for constraints
 - Constraints are checked immediately at the end of SQL statement execution (even within a transaction containing multiple SQL statements)
 - A violation will cause the statement to be rolled back
- Relaxed constraint checks: **Deferrable Constraints**
 - Check can be deferred for some constraints to the end of a transaction
 - Available for: UNIQUE, PRIMARY KEY, FOREIGN KEY

Deferrable Constraints — Motivation



Deferrable constraints may (temporarily) be violated within the scope of a transaction

Deferrable Constraints — Example

- Motivating example without deferrable constraints

Employees		
id	name	manager
101	Sarah	null
102	Judy	101
103	Max	102

```
CREATE TABLE Employees (
    id      INTEGER PRIMARY KEY,
    name    VARCHAR(50),
    manager INTEGER,
    CONSTRAINT manager_fkey FOREIGN KEY (manager) REFERENCES Employees (id)
        NOT DEFERRABLE -- default value (optional), check if constraint is immediate and cannot be changed
);
INSERT INTO Employees VALUES (101, 'Sarah', null), (102, 'Judy', 101), (103, 'Max', 102);

BEGIN;
DELETE FROM Employees WHERE id = 102;           -- Judy got fired → constraint violated → ABORT
UPDATE Employees SET manager = 101 WHERE id = 103; -- Max gets a new manager
COMMIT;
```

Deferrable Constraints — Example

Employees

id	name	manager
101	Sarah	null
102	Judy	101
103	Max	102

```
CREATE TABLE Employees (
    id      INTEGER PRIMARY KEY,
    name    VARCHAR(50),
    manager INTEGER,
    CONSTRAINT manager_fkey FOREIGN KEY (manager) REFERENCES Employees (id)
        DEFERRABLE INITIALLY DEFERRED -- check of constraint deferred by default
);
INSERT INTO Employees VALUES (101, 'Sarah', null), (102, 'Judy', 101), (103, 'Max', 102);

BEGIN;
DELETE FROM Employees WHERE id = 102;           -- Judy got fired → constraint violated but not checked
UPDATE Employees SET manager = 101 WHERE id = 103; -- Max gets a new manager → constraint re-established
COMMIT;
```

Deferrable Constraints — Example

Employees

id	name	manager
101	Sarah	null
102	Judy	101
103	Max	102

```
CREATE TABLE Employees (
    id      INTEGER PRIMARY KEY,
    name    VARCHAR(50),
    manager INTEGER,
    CONSTRAINT manager_fkey FOREIGN KEY (manager) REFERENCES Employees (id)
        DEFERRABLE INITIALLY IMMEDIATE -- check of constraint immediate by default, but can be changed
);
```

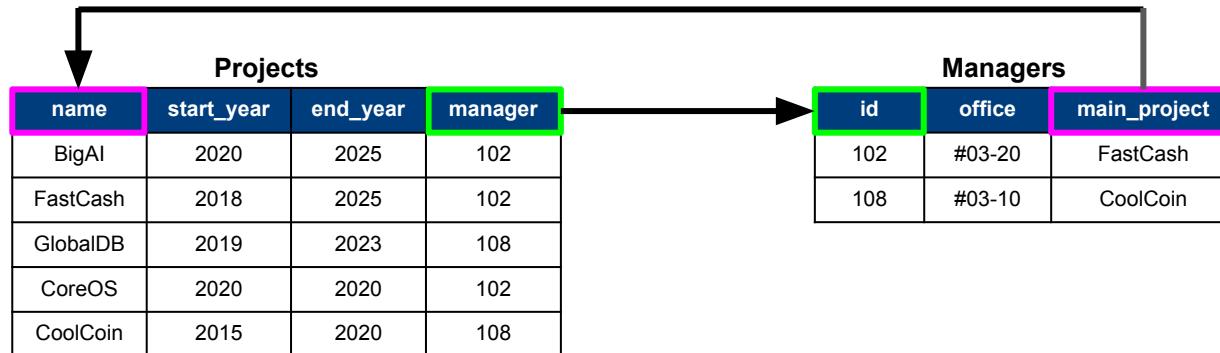
```
INSERT INTO Employees VALUES (101, 'Sarah', null), (102, 'Judy', 101), (103, 'Max', 102);
```

naming convenient

```
BEGIN;
SET CONSTRAINT manager_fkey DEFERRED;   -- Set check of constraint from "immediate" to "deferred"
DELETE FROM Employees WHERE id = 102;    -- Judy got fired → constraint violated but not checked
UPDATE Employees SET manager = 101 WHERE id = 103; -- Max gets a new manager → constraint re-established
COMMIT;
```

Deferrable Constraints — Benefits

- No need to care about order of SQL statements within a transaction
- Allows for cyclic foreign key constraints



- Performance boost when constraint checks are bottleneck
 - Example: batch insert of large number of tuples

Deferrable Constraints — (Potential) Downsides

- Troubleshooting can be more difficult
- Data definitions no longer unambiguous
- Performance penalty when performing queries

Overview

- **SQL — overview**
 - History and usages
 - SQL language groups
- **Creating a database with SQL**
 - Basic DDL & DML commands
 - Defining integrity constraints
 - Advanced: deferrable constraints
- **Modifying a database with SQL**
 - Basic DDL commands

DDL — Modifying a Schema

- **ALTER TABLE** statements to modify an existing data definition
 - CREATE TABLE statements do not have to be final data definition
 - Common: adding/dropping column, adding dropping constraints, changing data types
- Examples: Change specification of a single column

```
ALTER TABLE Projects ALTER COLUMN name TYPE VARCHAR(200); -- change data type to VARCHAR(200)
```

```
ALTER TABLE Projects ALTER COLUMN start_year SET DEFAULT 2021; -- set default value of column "start_year"
```

```
ALTER TABLE Projects ALTER COLUMN start_year DROP DEFAULT; -- drop default value of column "start_year"
```

DDL — Modifying a Schema

- Examples: Adding and dropping columns

```
ALTER TABLE Projects ADD COLUMN budget NUMERIC DEFAULT 0.0; -- add new column with a default value
```

```
ALTER TABLE Projects DROP COLUMN budget; -- drop column from table
```

- Examples: Adding and dropping constraints

```
ALTER TABLE Teams ADD CONSTRAINT eid_fkey FOREIGN KEY (eid) REFERENCES Employees (id);  
-- add foreign key constraint
```

```
ALTER TABLE Teams DROP CONSTRAINT eid_fkey;  
-- drop foreign key constraint (name of constraint might be retrieved from metadata)
```

DDL — Drop Tables

- **DROP TABLE** to delete tables from database

- Without dependent objects (incl. foreign key constraints, views, etc.)

```
DROP TABLE Projects;
```

```
DROP TABLE IF EXISTS Projects; -- check first if table exists; avoids throwing an error
```

- With dependent objects (assume foreign key constraint Teams.pname→Projects.name)

```
DROP TABLE Projects; -- will throw an error because of foreign key constraint
```

```
DROP TABLE Projects CASCADE; -- will delete table "Projects" and foreign key constraint  
-- (will not delete table "Teams"!)
```

Summary

- SQL — *the standard language for RDBMS*
 - Different language groups: DDL, DML, DQL, DCL, TCL
- Focus in this lecture: DDL and DML
 - DDL: **CREATE TABLE**, **ALTER TABLE**, **DROP TABLE**
 - DML: **INSERT**, **UPDATE**, **DELETE** , (**TRUNCATE**)
- Key challenge: specification of integrity constraints
 - **NOT NULL**, **UNIQUE**, **PRIMARY KEY**, **FOREIGN KEY**, **CHECK**
 - Specification actions in case of foreign key constraint violations (**ON UPDATE/DELETE**)
 - Relaxed checks of violations with deferrable constraints

Quick Quiz Solutions

Quick Quiz (Slide 27)

- Solution

- The unique constraint is NOT violated
- Example: (101, BigAI) vs (101, null)
 - "101 <> 101" evaluates to **false**
 - "Big <> null" evaluates to **unknown**
 - "**false or unknown**" evaluates to **unknown, not false**

Quick Quiz: Is the unique constraint of table "Teams" violated in the example below?

Teams

eid	pname	hours
101	BigAI	10
105	BigAI	5
102	GlobalDB	20
101	null	null
101	null	null
103	CoreOS	40
109	CoreOS	null

Quick Quiz (Slide 28)

- **Solution**
 - Only one "primary key" constraint can be defined
 - Multiple "unique not null" constraints can be defined

- **Additional comments**
 - Attributes with either "primary key" and "unique (not null)" constraint can be referenced by foreign keys

Quick Quiz: What is the difference between using "primary key" and "unique not null"?

Quick Quiz (Slide 36)

- Solution

- "FOREIGN KEY (pname) REFERENCES Projects (name) ON DELETE SET NULL"
will try to set Teams.pname to NULL if a corresponding project would get deleted
- Problem: "pname" is a prime attribute (i.e., part of the primary key)
and prime attributes are not allowed to be NULL
- Violation of primary key constraint → error!

CS2102: Database Systems

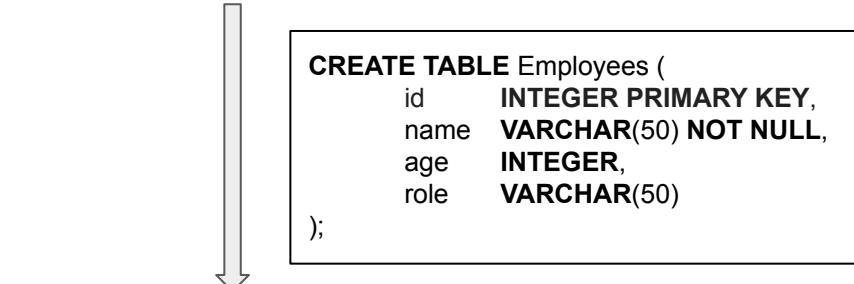
Lecture 3 — Entity Relationship Model (ER Model)

Quick Recap: SQL for Creating Databases

- **Data Definition Language (DDL)**

- Create, modify and drop tables to implement a given DB schema
- Specify integrity constraints (e.g., NOT NULL, PRIMARY KEY, FOREIGN KEY, CHECK)

Employees (`id: integer`, `name: text`, `age: integer`, `role: text`)



Employees

id	name	age	role
----	------	-----	------

- **Data Manipulation Language (DML)**

- Insert, update and delete data from tables

INSERT INTO Employees VALUES
(101, 'Sarah', 25, 'dev')
(102, 'Judy', 35, 'sales');

Employees

id	name	age	role
101	Sarah	25	dev
102	Judy	35	sales

We Sneakily Skipped a Step

- Open questions:
 - Where does the database schema come from?
 - What tables with which attributes do we need?
 - What data integrity constraints are required?
 - Table names, attribute names, data types, ...?

→ Database Design Process

Quick Quiz: Which table is "better"?

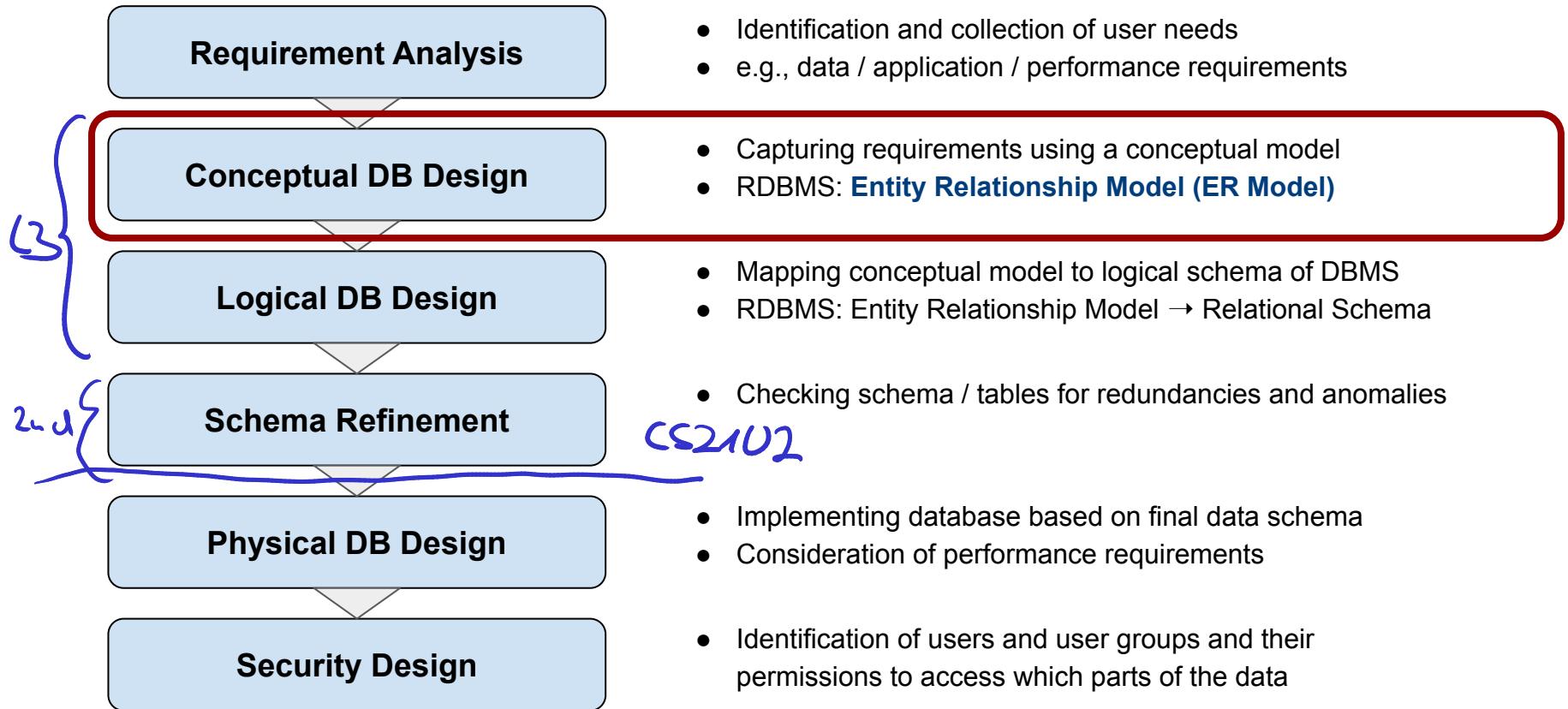
```
CREATE TABLE Employees (
    id      INTEGER PRIMARY KEY,
    name   VARCHAR(50) NOT NULL,
    age    INTEGER,
    role   VARCHAR(50)
);
```

or

```
CREATE TABLE Employees (
    id      INTEGER PRIMARY KEY,
    name   VARCHAR(50) NOT NULL,
    dob    DATE,
    role   VARCHAR(100)
);
```



Database Design Process — 6 Common Steps



Overview

- **Entity Relationship Model**
 - Overview + ER diagrams
 - Entity sets and attributes
 - Relationship sets
 - Cardinality & participation constraints
 - Dependency constraints: weak entity sets
 - Aggregation
- **Relational Mapping**
 - From ER diagram to database tables
- **Summary**

Requirement Analysis: Online Airline Reservation System (OARS)

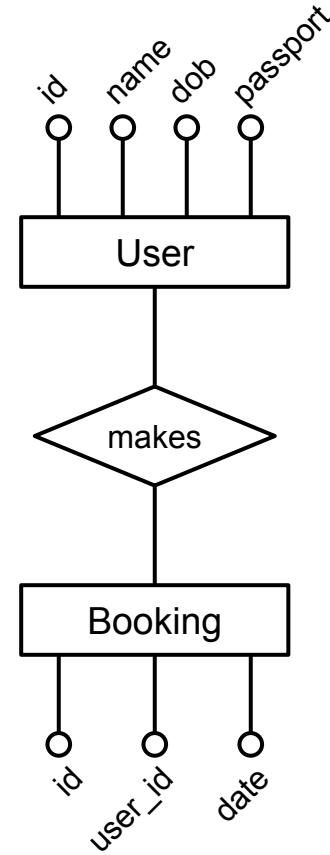
Users need to be able to make bookings from an origin to a destination airport which may comprise multiple connecting flights. Each flight has a flight number, the origin and destination airport, the distance in kilometers, the departure and arrival time, and the days of the week the flight is in operation.

A flight instance is the actual scheduled flight on a given day together with the assigned aircraft type. For example, flight SQ231 flies daily from Singapore to Sydney, typically with a Boeing 777-300ER (code: B77W).

For a valid booking, we need the user's name, sex, address, phone number(s), and the passport number. Users are only able to pay via credit card. When making a booking, the user can select the class, the seat number, as well as meal preferences (if available).

Entity Relationship Model

- ER Model
 - Most common model for conceptual database design
 - Developed by Peter Chen (1976)
 - Visualized using **ER diagrams**
(Important: many revised version – no one single set of notations!)
- Core concepts
 - All data is described in terms of **entities** and their **relationships**
 - Information about entities & relationships are described using **attributes**
 - Certain data constraints can be described using additional annotations



Overview

- **Entity Relationship Model**
 - Overview + ER diagrams
 - **Entity sets and attributes**
 - Relationship sets
 - Cardinality & participation constraints
 - Dependency constraints: weak entity sets
 - Aggregation
- **Relational Mapping**
 - From ER diagram to database tables
- **Summary**

Entities and Entity Sets

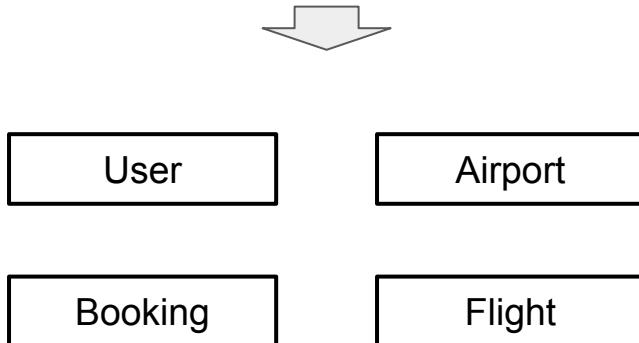
- **Entity**

- Real-world things or objects that are distinguishable from other objects
(e.g., an individual user, airport, flight, or booking)

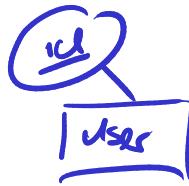
Users need to be able to make bookings from an origin to a destination airport which may comprise multiple connecting flights. Each flight has a flight number, [...]

- **Entity Set**

- Collection of entities of the same type
- Represented by rectangles in ER diagrams
- Names are typically nouns



Attributes



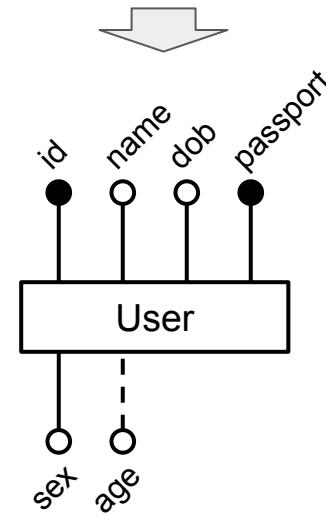
- **Attribute:**

- specific information describing an entity
- represented by a small circle in ER diagrams

- 2 main subtypes of attributes

- **Key attribute(s):** uniquely identifies each entity
 - Indicated by a filled circle in ER diagram
 - Different attributes may uniquely identify an entity
 - Multiple attributes may form a composite key
- **Derived attribute:** derived from other attributes
 - Indicated by a dashed line in ER diagram
 - Example: derive "age" from "dob"

For a valid booking, we need the **user's name, sex, address, phone number(s), and the passport number.**
Users are only able to pay via credit card. [...]



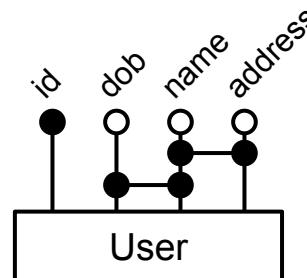
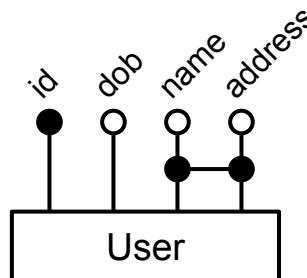
What about address and phone numbers?

Key Attributes

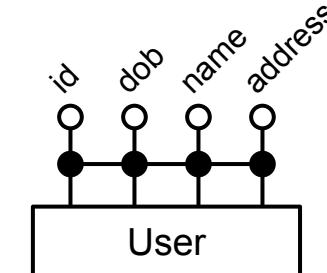
- **Composite key attributes:**

- 2 or more attributes together uniquely identify each entity
- An entity may have multiple composite key attributes
- Representation in ER diagram: additional connecting line

- **Examples** (for illustration purposes; not necessarily realistic!)

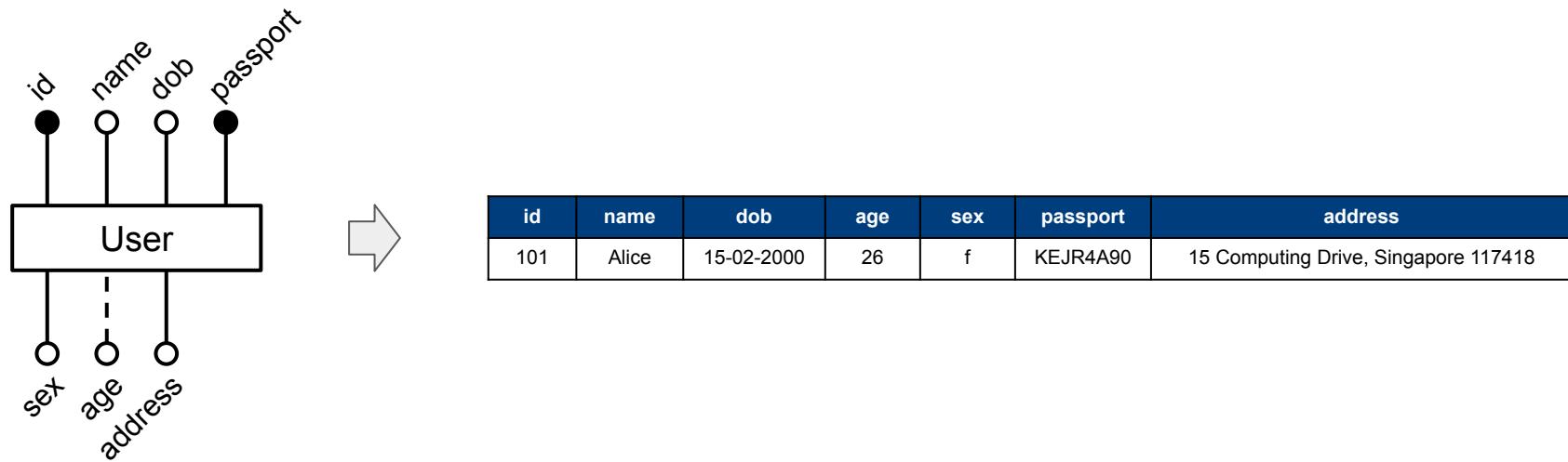


- At least all attributes uniquely identify an entity
- We typically prefer a minimum set of attributes



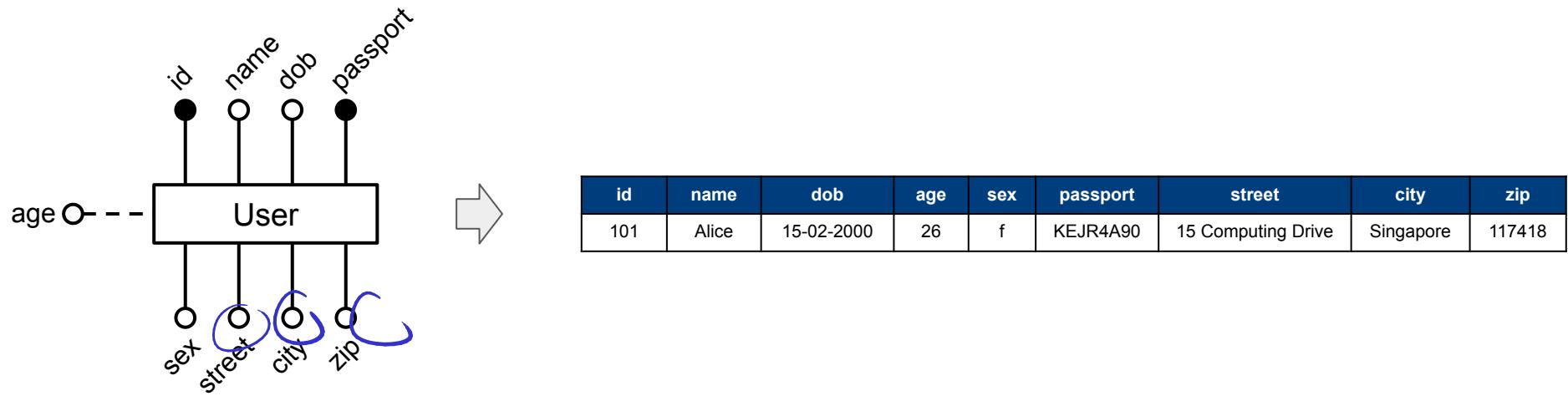
"Composite" Attributes

- Common: requirement analysis often vague / ambiguous / unclear
 - Not always obvious how certain attributes should be modeled
 - Example "address": single string attribute vs. multiple attributes



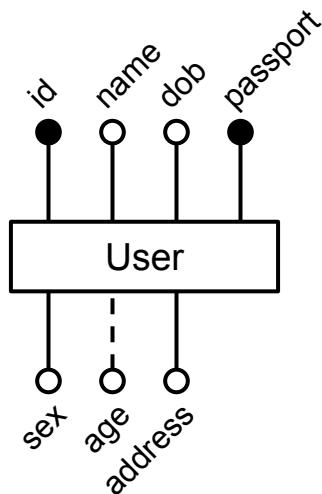
"Composite" Attributes

- Common: requirement analysis often vague / ambiguous / unclear
 - Not always obvious how certain attributes should be modeled
 - Example "address": single string attribute vs. multiple attributes

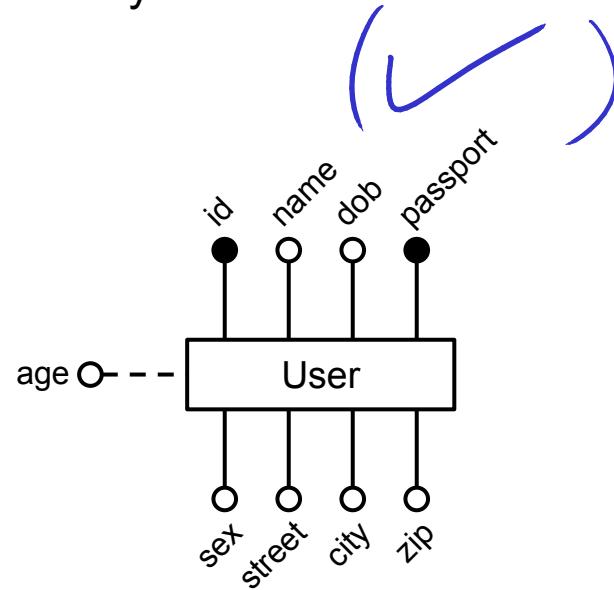


Quick Quiz

Which solution is typically the **preferred** one?
But always? And why?

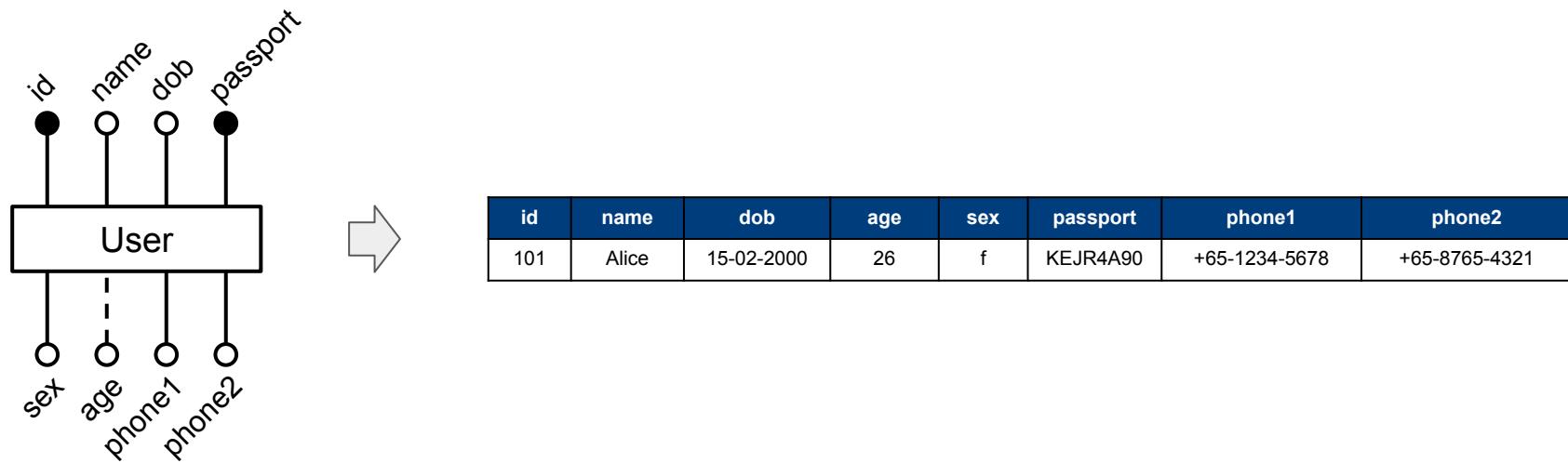


vs



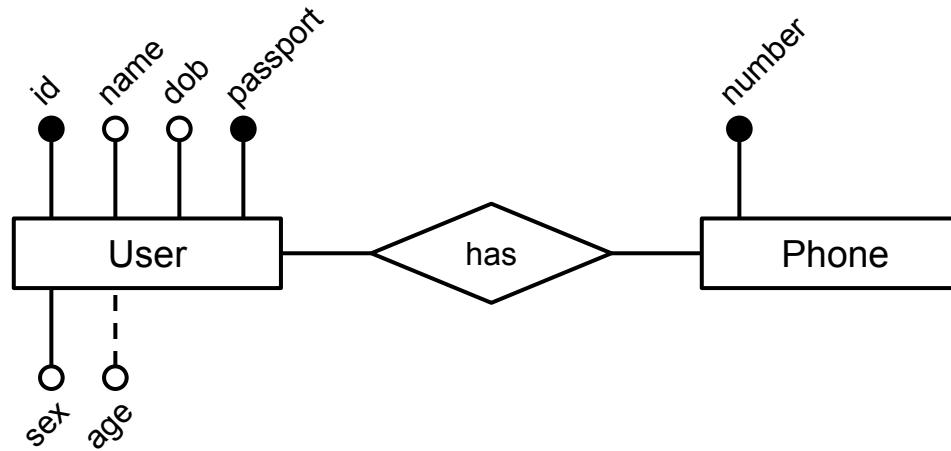
Multivalued Attributes

- Common: an attribute may refer to a set/list of values
 - Examples: phone numbers, hobbies, tags/keywords
 - However: all attributes must be single-valued
 - Example "phone numbers": fixed number of single-valued attributes vs. dedicated entity set



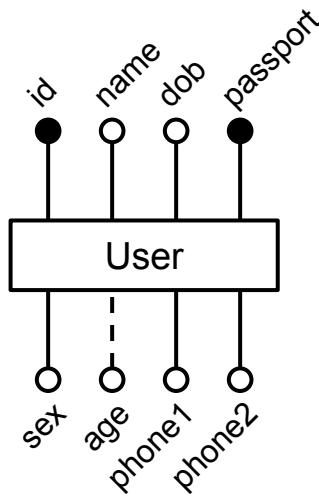
Multivalued Attributes

- Common: an attribute may refer to a set/list of values
 - Examples: phone numbers, hobbies, tags/keywords
 - However: all attributes must be single-valued
 - Example "phone numbers": fixed number of single-valued attributes vs. dedicated entity set

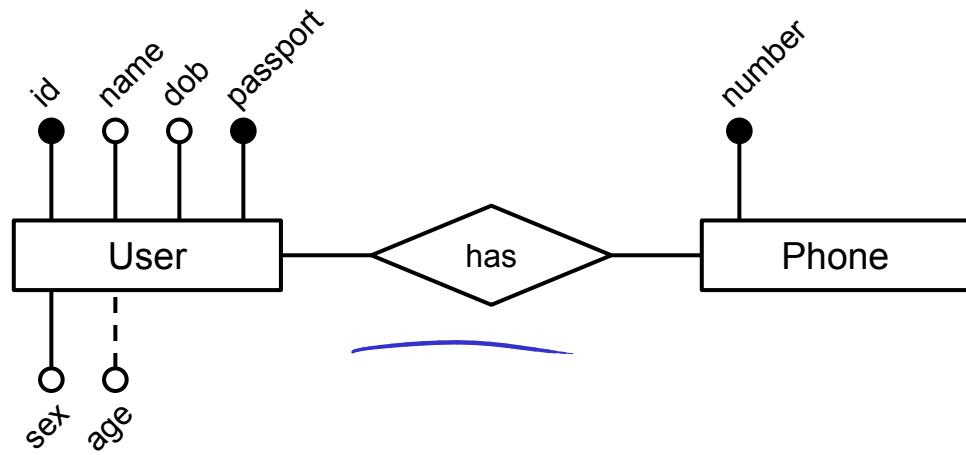


Quick Quiz

Which solution is typically the **preferred** one?
But always? And why?



vs



Side Note

- PostgreSQL (and most modern RDBMS)
 - Not limited to basic single-valued data types
 - Support for complex / composite data types
 - Support for user-defined composite types

Quick Quiz: What are potential downsides of this more complex data types?

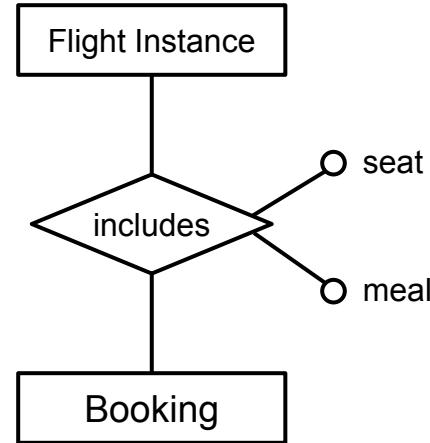
8.13. XML Type	8.13.1. Creating XML Values
	8.13.2. Encoding Handling
	8.13.3. Accessing XML Values
8.14. JSON Types	8.14.1. JSON Input and Output Syntax
	8.14.2. Designing JSON Documents
	8.14.3. jsonb Containment and Existence
	8.14.4. jsonb Indexing
	8.14.5. jsonb Subscripting
	8.14.6. Transforms
	8.14.7. jsonpath Type
8.15. Arrays	8.15.1. Declaration of Array Types
	8.15.2. Array Value Input
	8.15.3. Accessing Arrays
	8.15.4. Modifying Arrays
	8.15.5. Searching in Arrays
	8.15.6. Array Input and Output Syntax
8.16. Composite Types	8.16.1. Declaration of Composite Types
	8.16.2. Constructing Composite Values
	8.16.3. Accessing Composite Types
	8.16.4. Modifying Composite Types
	8.16.5. Using Composite Types in Queries
	8.16.6. Composite Type Input and Output Syntax

Overview

- **Entity Relationship Model**
 - Overview + ER diagrams
 - Entity sets and attributes
 - **Relationship sets**
 - Cardinality & participation constraints
 - Dependency constraints: weak entity sets
 - Aggregation
- **Relational Mapping**
 - From ER diagram to database tables
- **Summary**

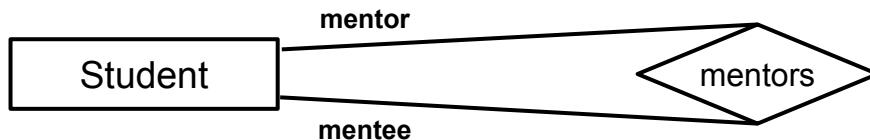
Relationships and Relationship Sets

- **Relationship**
 - Association among two or more entities
- **Relationship Set**
 - Collection of relationships of the same type
 - Represented by diamonds in ER diagrams
 - Can have their own attributes that further describe the relationship
 - Names are typically verbs
- **Additional annotations to further specify relationships**
 - Roles, degree, cardinalities, participation, dependencies



Relationship Roles

- **Role**
 - Descriptor of an entity set's participation in a relationship
 - Most of the time implicitly given by the name of the entity sets
 - Explicit role labels only common in case of ambiguities
(typically in case the same entity sets participate in the same relationship more than once)
- Example: Students can mentor other students

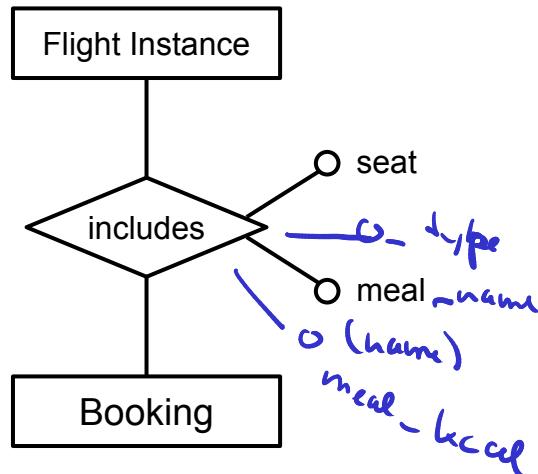


Degree of Relationship Sets

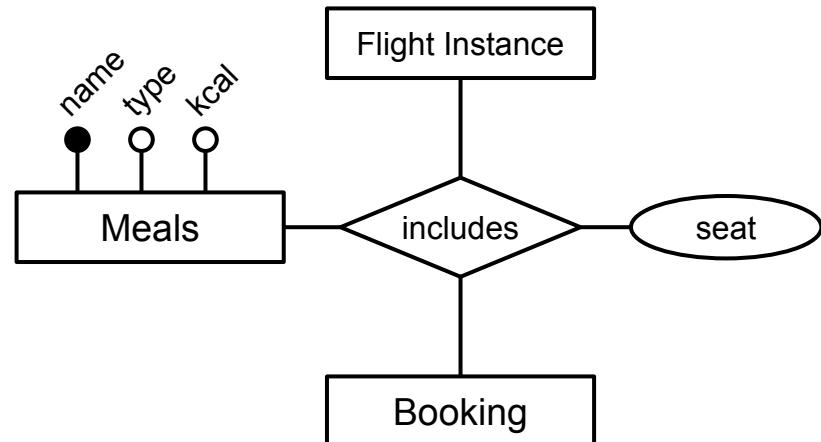
- **Degree**

- In principle, no limitation on how many entity roles participate in a relationship
- An n -ary relationship set involves n entity roles → $n =$ degree of relationship set

$n = 2 \rightarrow$ binary relationship set

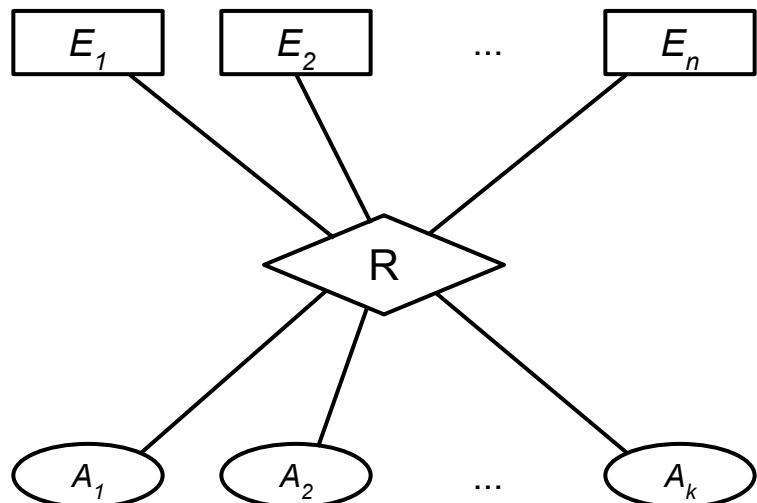


$n = 3 \rightarrow$ ternary relationship set



Degree of Relationship Sets

- General n-ary relationship set R
 - n participating entity sets E_1, E_2, \dots, E_n
 - k relationship attributes A_1, A_2, \dots, A_k



"In typical modeling, binary relationships are the most common and relationships with $n > 3$ are very rare" - Peter Chen (2009)

Overview

- **Entity Relationship Model**
 - Overview + ER diagrams
 - Entity sets and attributes
 - Relationship sets
 - **Cardinality & participation constraints**
 - Dependency constraints: weak entity sets
 - Aggregation
- **Relational Mapping**
 - From ER diagram to database tables
- **Summary**

Cardinality & Participation Constraints

- **Cardinalities of Relationship Sets**

- Describe how often an entity can participate in a relationship at most
- 3 basic cardinality constraints
 - **Many-to-many** (e.g., a flight can be performed by different aircrafts; an aircraft can perform different flights)
 - **Many-to-one** (e.g., a user can make many bookings, but each booking is done by one user)
 - **One-to-one** (e.g., a user is associated with one set of credit card details, and vice versa)

upper bound

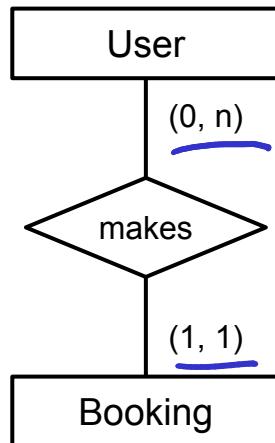
- **Participation constraints**

- Describe how often an entity has to participate in a relationship at least
- Is the participation of an entity in a relationship even mandatory?

lower bound

Cardinality & Participation Constraints

- Representation in ER diagram
 - (min,max) label at connections between entity and relationship sets



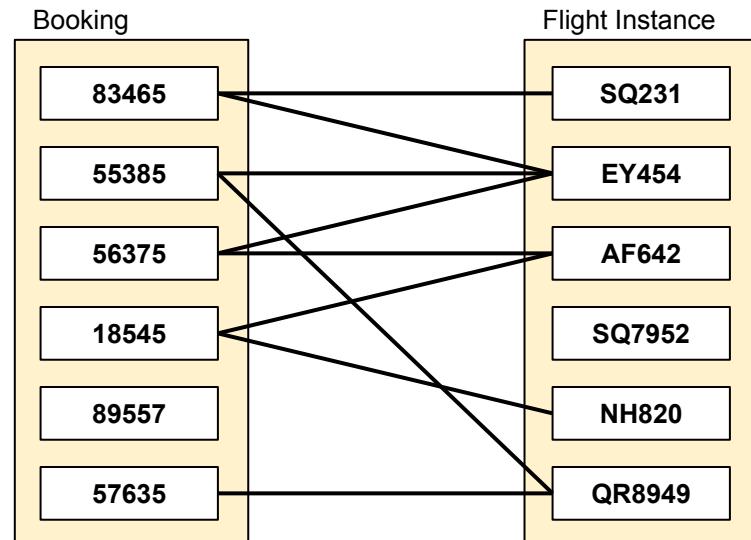
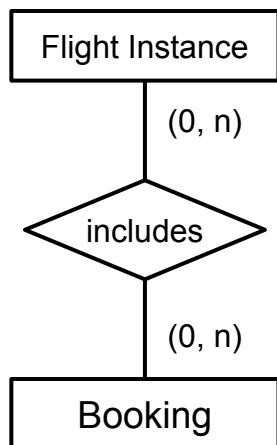
Interpretation

- Each user can make multiple bookings
(but not every user must have made a booking)
- Each booking was done by exactly one user
(implies that each booking is associated with a user)

Cardinality: Many-to-Many (no mandatory participation)

- Many-to-many relationship between bookings and flight instances

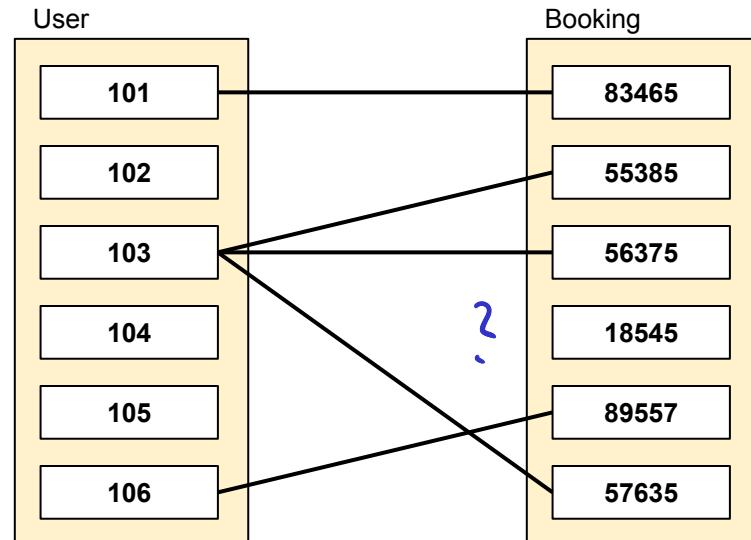
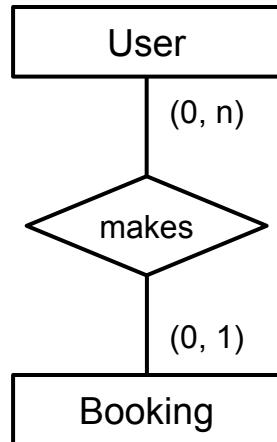
- Each booking can include 0 or more flight instances
(note that a booking with 0 flights might not be meaningful; we will improve on that)
- Each flight instance can be part of 0 or more bookings



Cardinality: Many-to-One (no mandatory participation)

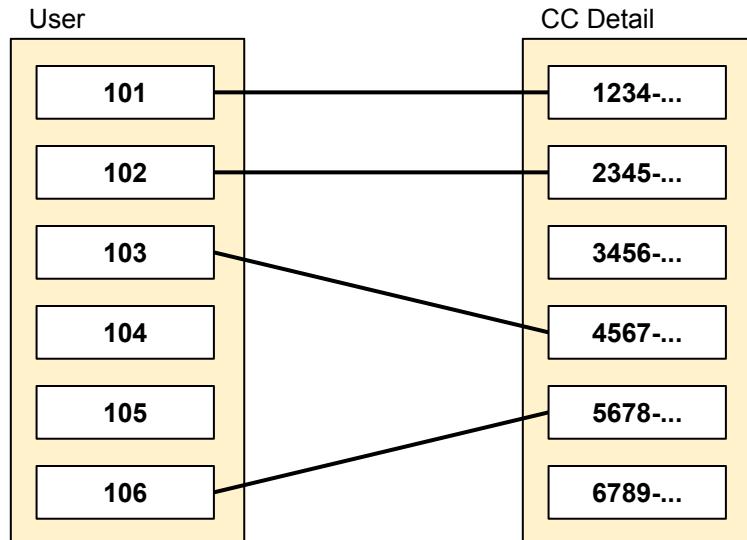
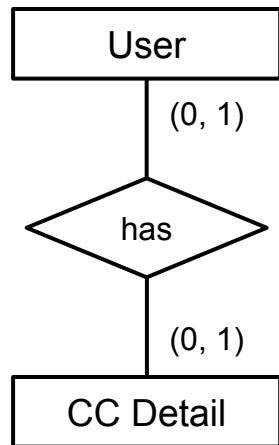
- Many-to-one relationship between users and bookings

- Each user can make 0 or more bookings
- Each booking is done by one 1 user at most
(again, not perfect yet, and we will improve on that)



Cardinality: One-to-One (no mandatory participation)

- One-to-one relationship between users and credit card details
 - Each user can provide only 1 set of credit card details at most
 - Each set of credit card details is associated with 1 user at most



Participation Constraints

- Limitation of (basic) cardinality constraints from previous examples

- A booking can include 0 flights
- A booking can be done by 0 users
- A set of credit card details does not need to be associated with a user

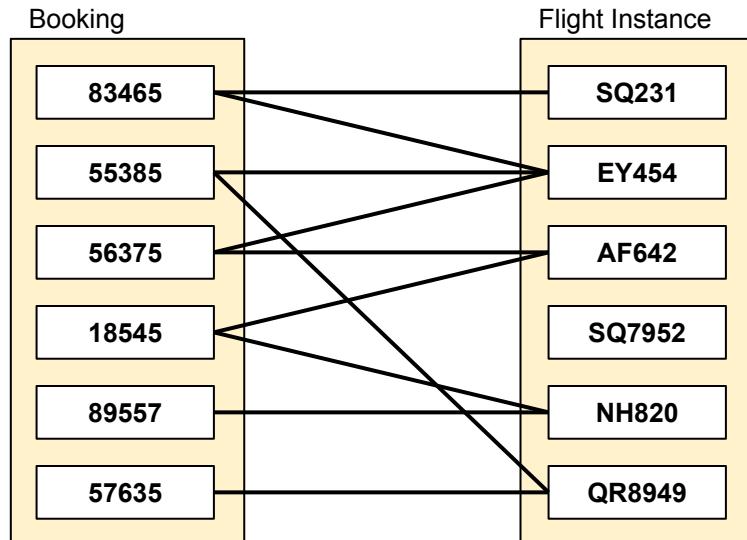
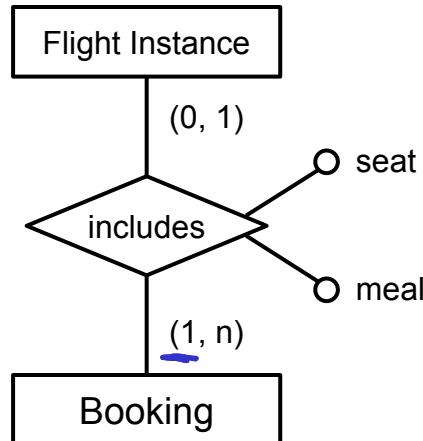


an entity does not have to participate in a relation

→ Let's include **participation constraints**

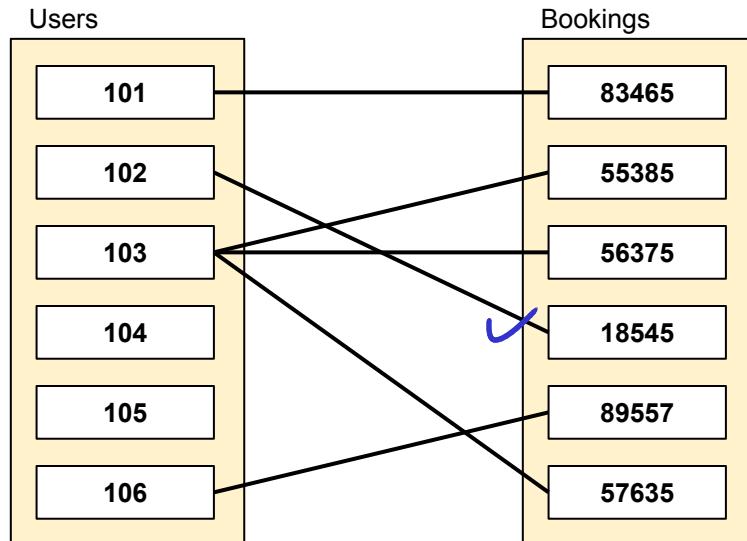
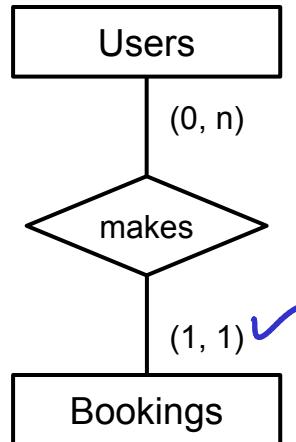
Cardinality & Participation Constraints

- Many-to-many relationship between bookings and flight instances
 - Each booking includes 1 or more flight instances
 - Each flight instance can be part of 0 or more bookings



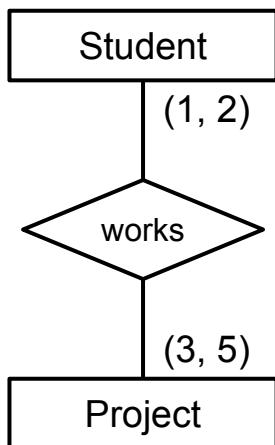
Cardinality & Participation Constraints

- Many-to-one relationship between users and bookings
 - Each user can make 0 or more bookings
 - Each booking is done by exactly 1 user



Cardinality & Participation Constraints

- Flexibility of (min,max) notation
 - Minimum not limited to 0 or 1; maximum not limited to n
 - Arbitrary specific values to capture real-world constraint



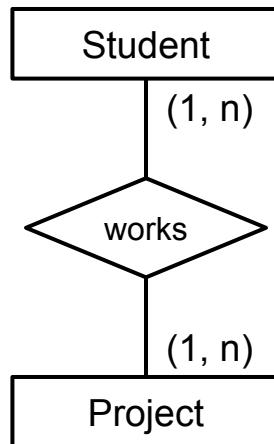
Interpretation

- Each student must work on at least 1 project
- Each student may not work on more than 2 projects
- Each project consists of at least 3 students
- Each project may not consist of more than 5 students

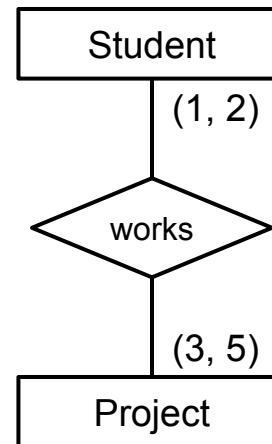
Quick Quiz

Why do values other than 0/1/n add significant complexity?

(just think about it for a minute here; we will cover it later)



vs



→ triggers

Overview

- **Entity Relationship Model**

- Overview + ER diagrams
- Entity sets and attributes
- Relationship sets
- Cardinality & participation constraints
- **Dependency constraints: weak entity sets**
- Aggregation

- **Relational Mapping**

- From ER diagram to database tables

- **Summary**

Dependency Constraints

- **Weak entity sets**

- Entity set that does not have its own key
- A weak entity can only be uniquely identified by considering the primary key of the **owner entity**
- A weak entity's existence depends on the existence of its owner entity
- Weak entity set and identifying relation set are represented via double-lined rectangles / diamonds

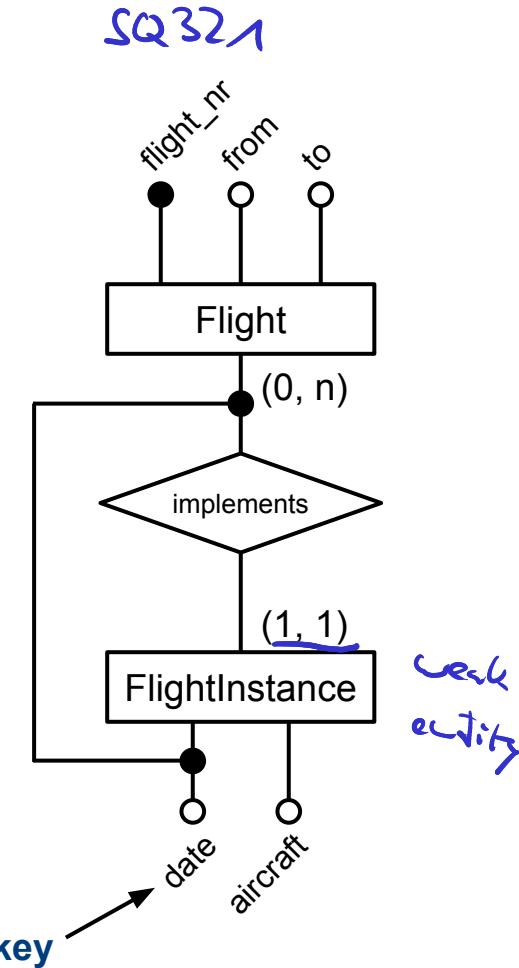
- Requirements

- Many-to-one relationship (identifying relationship) from weak entity set to owner entity set
(one-to-one possible but less common)
- Weak entity set must have (1, 1) attached to identifying relationship

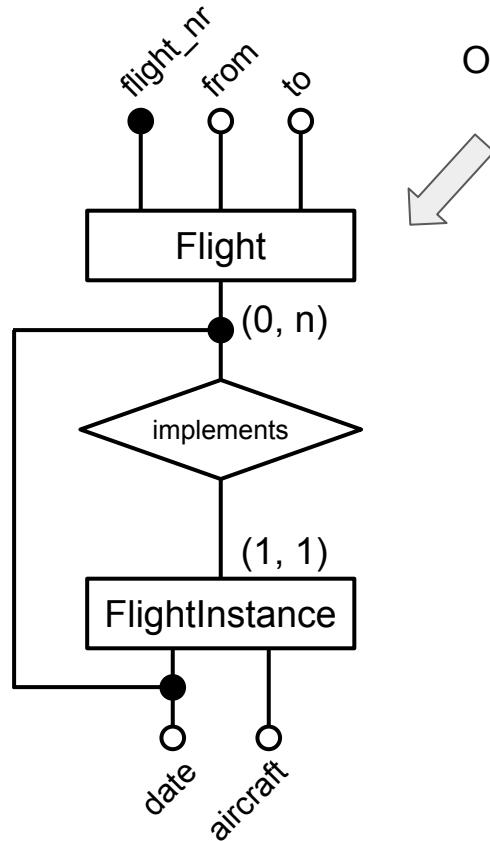
Dependency Constraints

- Example

- A flight instance is the actual scheduled flight (with a unique flight number) on a given day
 - Each flights instance is identified by the "flight_nr and the "date"
 - "date" is a **partial key**
- A flight instance cannot "exist" without the flight

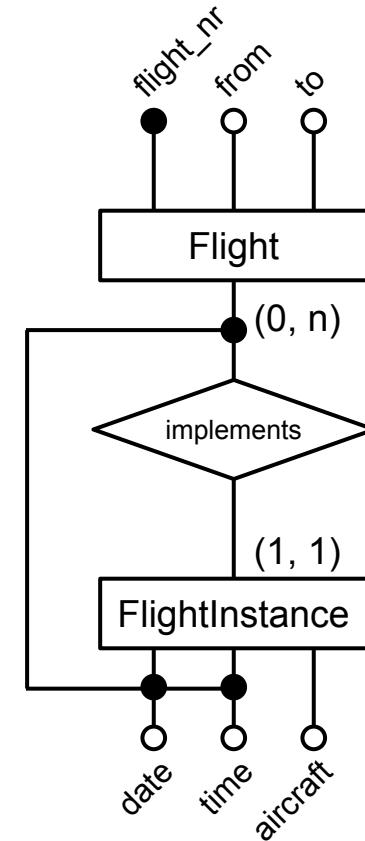


Dependency Constraints



Only 1 flight (instance)
per day maximum

More than 1 flight
(instance) per day

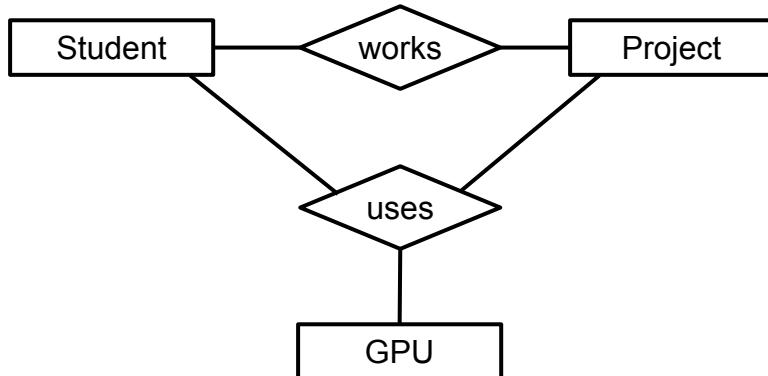


Overview

- **Entity Relationship Model**
 - Overview + ER diagrams
 - Entity sets and attributes
 - Relationship sets
 - Cardinality & participation constraints
 - Dependency constraints: weak entity sets
 - **Aggregation**
- **Relational Mapping**
 - From ER diagram to database tables
- **Summary**

Extended Concepts — Aggregation

- Concepts of ER diagrams so far
 - Only relationships between entity sets
 - No relationships between entity sets and relationship sets
- Motivating example



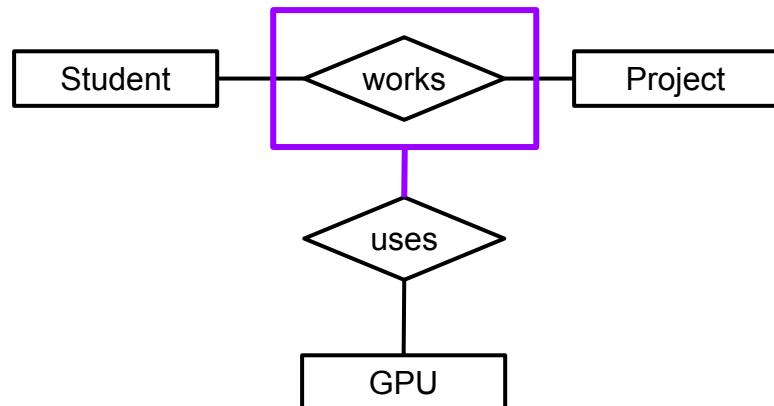
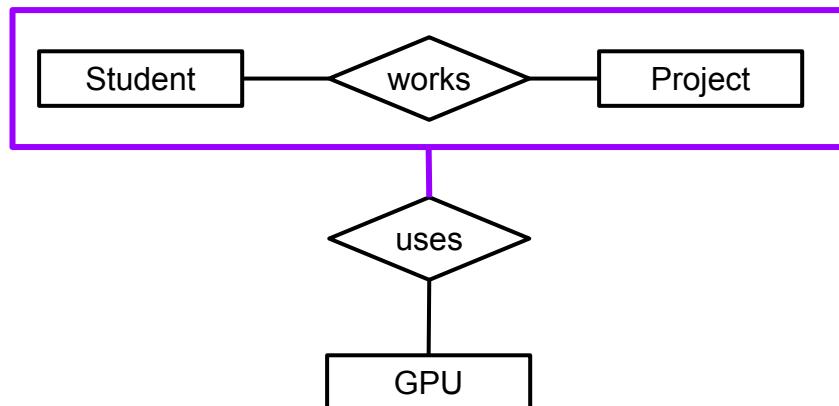
Limitations:

- Relationship between "works" and "uses" not explicitly captured
- "works" and "uses" are kind of redundant relationships

→ **Aggregation**

Extended Concepts — Aggregation

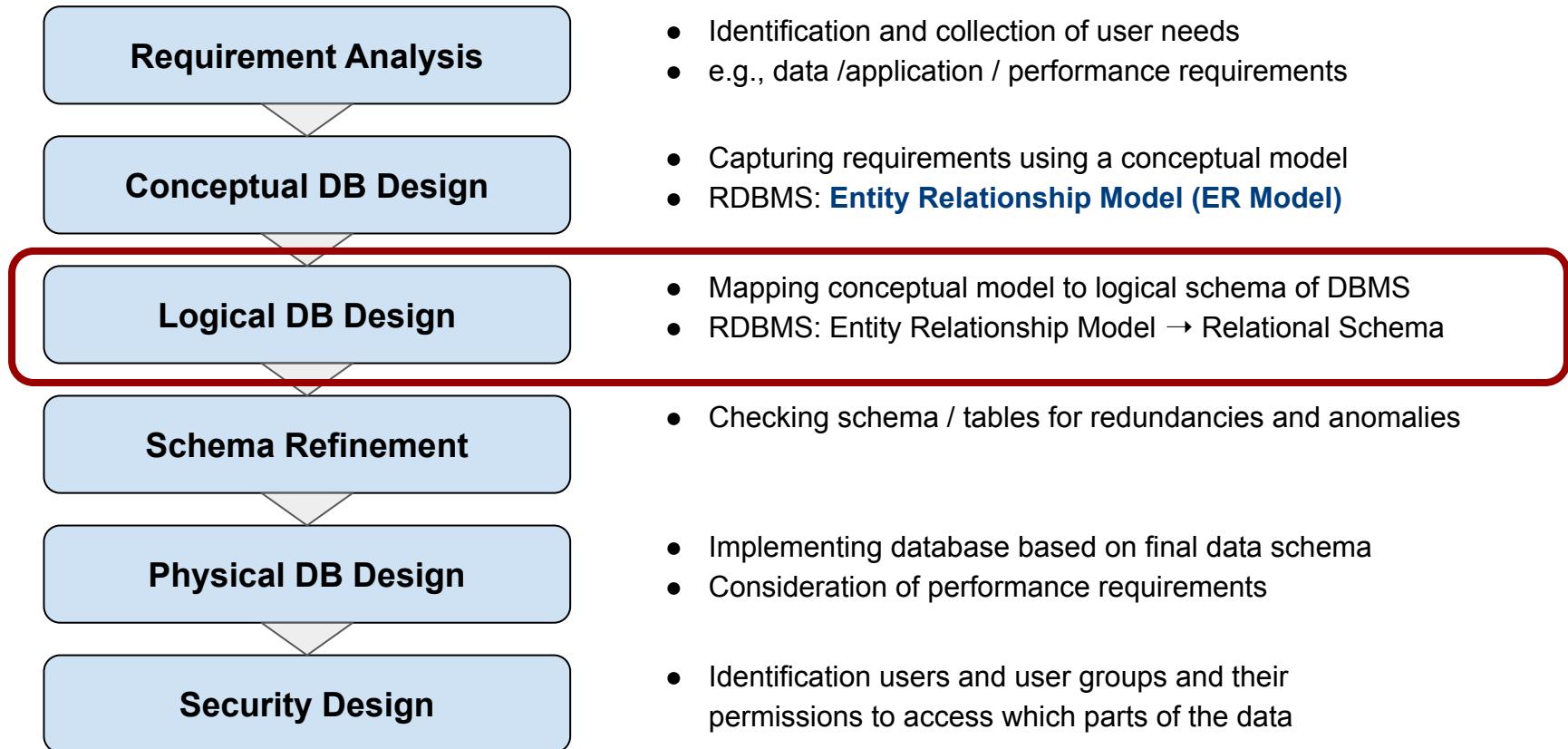
- Aggregation — basic idea
 - Abstraction that treats relationships as higher-level entities
 - Example: treat Students-works-Projects as an entity set
- Notation in ER diagram (2 equivalent alternatives)



Overview

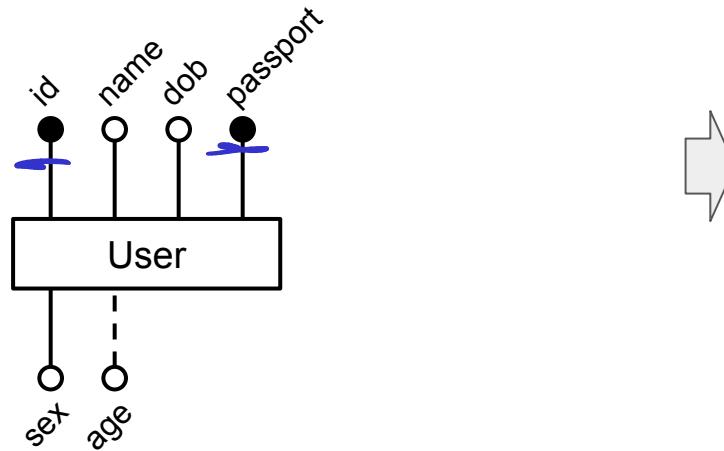
- Entity Relationship Model
 - Overview + ER diagrams
 - Entity sets and attributes
 - Relationship sets
 - Cardinality & participation constraints
 - Dependency constraints: weak entity sets
 - Aggregation
- Relational Mapping
 - From ER diagram to database tables
- Summary

Database Design Process — 6 Common Steps



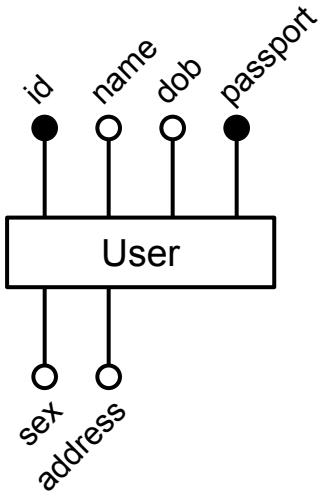
Entity Sets

- Straightforward mapping from entity sets to tables (except for composite & multivalued attributes)
 - Name of entity set → name of table
 - Attributes of entity set → attributes of table
 - Key attributes of entity set → primary key of table

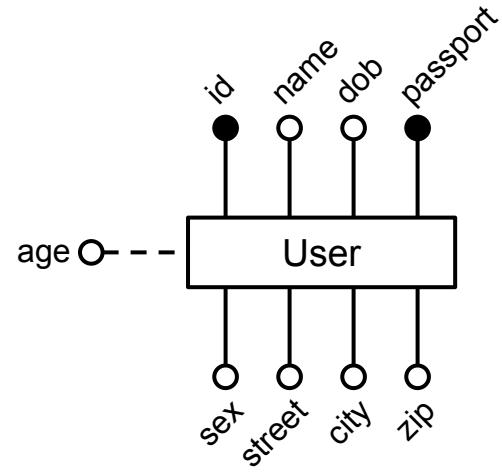


```
CREATE TABLE Users (
    id      INTEGER,
    name    VARCHAR(100),
    dob     DATE,
    sex     CHAR(1),
    age     INTEGER,
    passport VARCHAR(20),
    PRIMARY KEY (id),
    UNIQUE (passport)
);
```

Note: PostgreSQL supports [Generated Column](#) but there are some caveats when used in practice that are beyond our scope.



VS

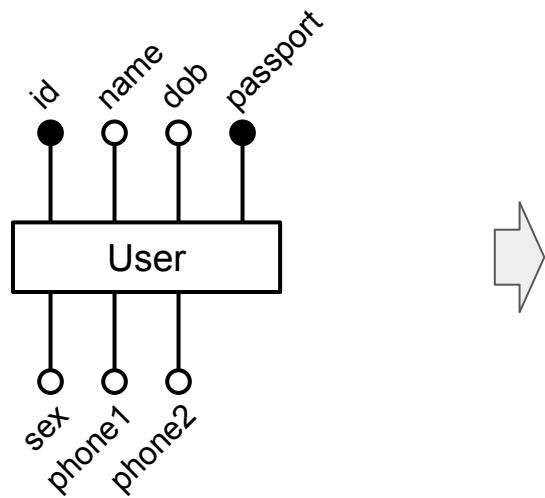


```
CREATE TABLE Users (
    id      INTEGER,
    name   VARCHAR(100),
    dob    DATE,
    sex    CHAR(1)
    passport VARCHAR(20),
    address VARCHAR(200),
    PRIMARY KEY (id),
    UNIQUE (passport)
);
```

```
CREATE TABLE Users (
    id      INTEGER,
    name   VARCHAR(100),
    dob    DATE,
    sex    CHAR(1)
    passport VARCHAR(20),
    street  VARCHAR(100),
    city   VARCHAR(100),
    zip    VARCHAR(10),
    PRIMARY KEY (id),
    UNIQUE (passport)
);
```

Multivalued Attributes

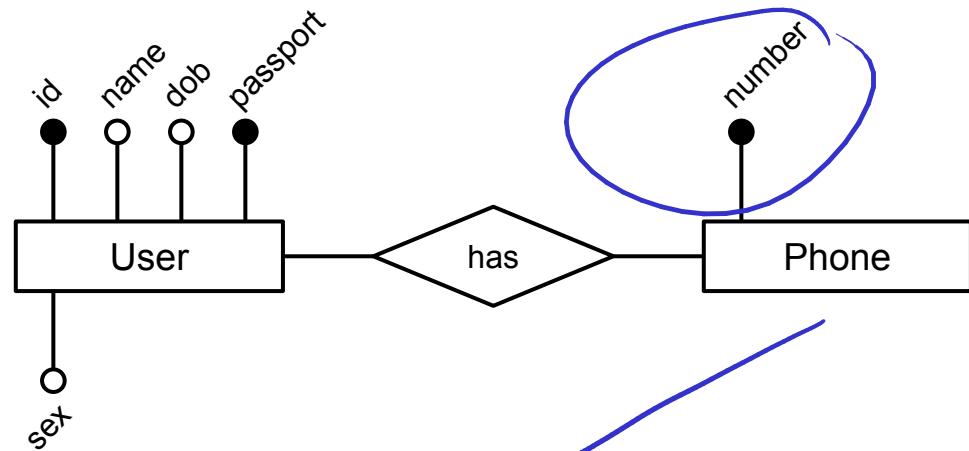
- Fixed number of single-valued attributes



```
CREATE TABLE Users (
    id      INTEGER,
    name   VARCHAR(100),
    dob    DATE,
    sex    CHAR(1),
    passport VARCHAR(20),
    phone1 VARCHAR(20),
    phone2 VARCHAR(200),
    PRIMARY KEY (id),
    UNIQUE (passport)
);
```

Multivalued Attributes

- Separate entity set for phone numbers

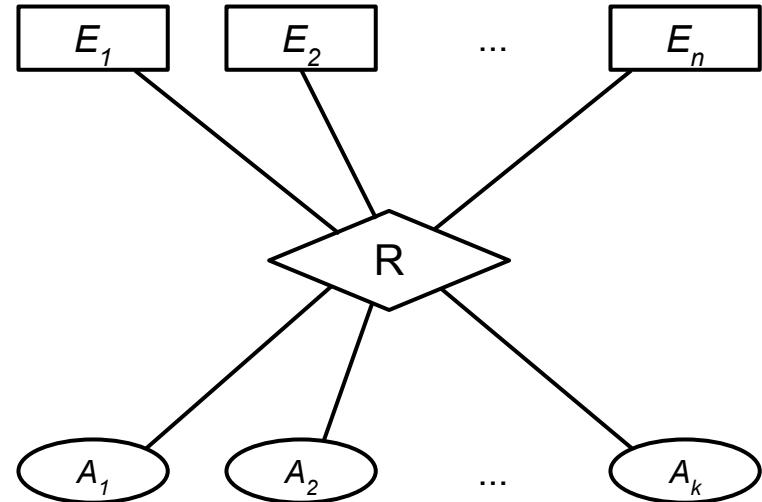


```
CREATE TABLE Users (
    id      INTEGER,
    name   VARCHAR(100),
    dob    DATE,
    sex    CHAR(1),
    passport VARCHAR(20),
    PRIMARY KEY (id),
    UNIQUE (passport)
);
```

```
CREATE TABLE Phones (
    number  VARCHAR(20),
    user_id INTEGER,
    PRIMARY KEY (number),
    FOREIGN KEY (user_id) REFERENCES Users (id)
);
```

Relationship Sets

- General n-ary relationship set R
 - n participating entity sets E_1, E_2, \dots, E_n
 - k relationship attributes A_1, A_2, \dots, A_k
 - Let $\text{Key}(E_i)$ be the attributes of the selected key of entity set E_i



→ Attributes of relationship set R

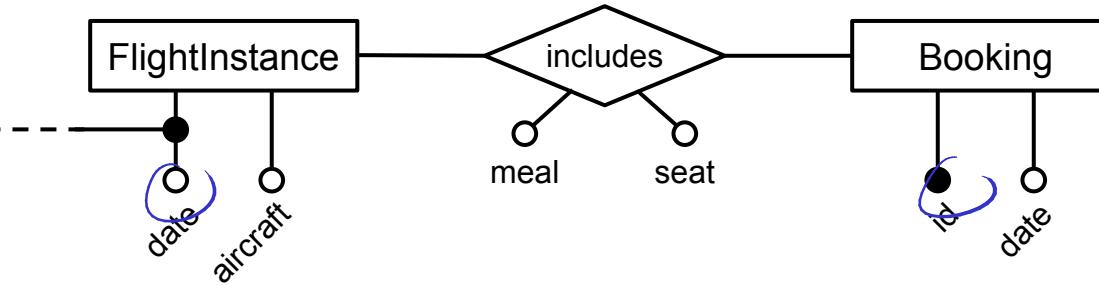
- $\text{Key}(E_1), \text{Key}(E_2), \dots, \text{Key}(E_n)$ — key attributes of all participating entity sets E_i
- A_1, A_2, \dots, A_k — all relationship attributes of R

Cardinality: Many-to-Many (no mandatory participation)

Quick Quiz: Where does "flight_nr" come from?

CREATE TABLE FLIGHt (

fur
date
:
:



CREATE TABLE Includes (

flight_nr VARCHAR(10),
flight_date DATE,
booking_id INTEGER,
seat VARCHAR(10),
meal VARCHAR(50)

PRIMARY KEY (flight_nr, flight_date, booking_id),

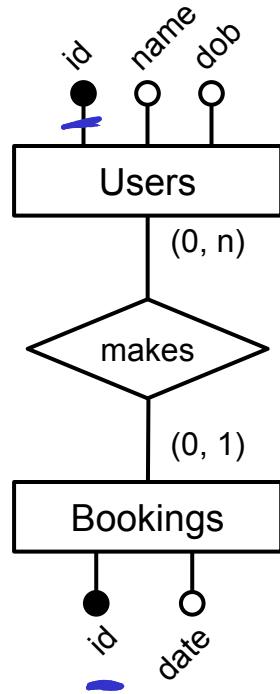
FOREIGN KEY (flight_nr, flight_date) **REFERENCES** FlightInstances (flight_nr, date),

FOREIGN KEY (booking_id) **REFERENCES** Bookings (id),

);

Cardinality: Many-to-One (no mandatory participation)

- Approach 1: Represent "makes" with a separate table
 - Similar to Many-to-Many but with different primary key!

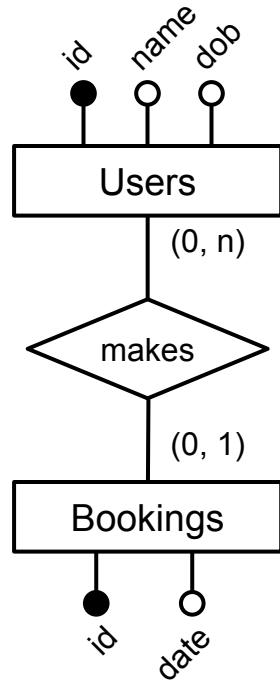


```
CREATE TABLE Makes (
    user_id    INTEGER,
    booking_id INTEGER,
    PRIMARY KEY (booking_id),
    FOREIGN KEY (user_id) REFERENCES Users (id),
    FOREIGN KEY (booking_id) REFERENCES Bookings (id)
);
```

Cardinality: Many-to-One (no mandatory participation)

Quick Quiz: What is typically the preferred approach? 1 or 2?

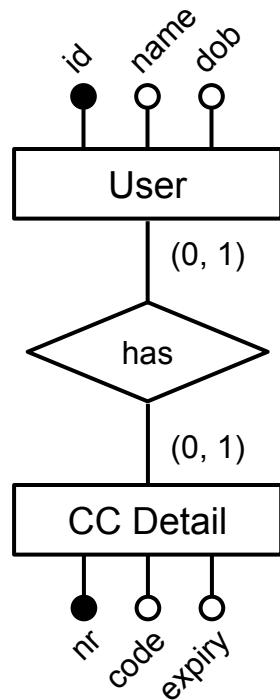
- **Approach 2: Combine "makes" and "Bookings" into one table**
 - Possible because given a booking, we can uniquely identify the user who made it



```
CREATE TABLE Bookings (
    id      INTEGER,
    date   DATE,
    user_id INTEGER,
    PRIMARY KEY (id),
    FOREIGN KEY (user_id) REFERENCES Users (id)
);
```

Cardinality: One-to-One (no mandatory participation)

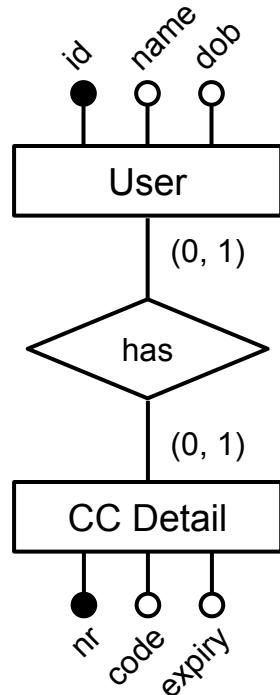
- Approach 1: Represent "has" with a separate table
 - Similar to Many-to-One but primary key can be chosen



```
CREATE TABLE Has (
    user_id      INTEGER,
    cc_nr        CHAR(16) UNIQUE,
    PRIMARY KEY (user_id),
    FOREIGN KEY (user_id) REFERENCES Users (id),
    FOREIGN KEY (cc_nr) REFERENCES CCDetails (id)
);
```

Cardinality: One-to-One (no mandatory participation)

- Approach 2: Combine "has" and "Users" or "has" and "CC Details"



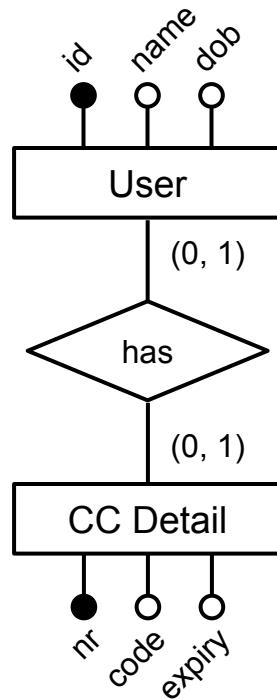
```
CREATE TABLE Users (
    id      INTEGER,
    name   VARCHAR(100),
    dob    DATE,
    cc_nr  CHAR(16) UNIQUE,
    PRIMARY KEY (id),
    FOREIGN KEY (cc_nr) REFERENCES CCDetails (nr)
);
```

```
CREATE TABLE CCdetails (
    nr      CHAR(16),
    code   CHAR(3),
    expiry DATE,
    user_id INTEGER UNIQUE,
    PRIMARY KEY (nr),
    FOREIGN KEY (user_id) REFERENCES Users (id)
);
```

Cardinality Constraints: One-to-One

Quick Quiz: What could be a downside of this approach?
(Hint: security)

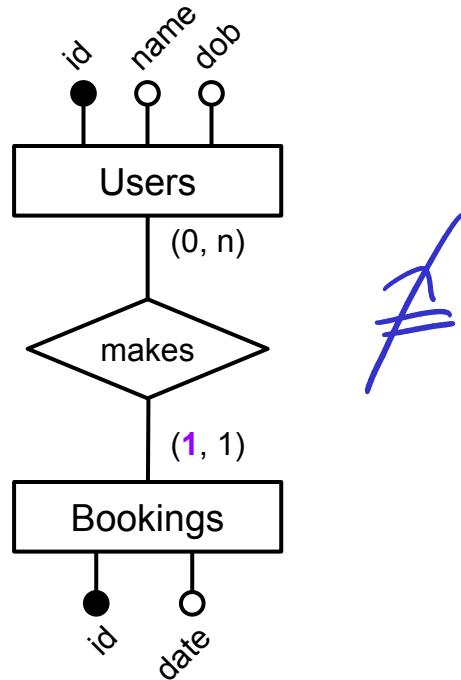
- Approach 3: Combine "has", "Users", and "CC Details"



```
CREATE TABLE Users (
    id      INTEGER,
    name   VARCHAR(100),
    dob    DATE,
    cc_nr  CHAR(16) UNIQUE,
    cc_code CHAR(3),
    cc_expiry DATE,
    PRIMARY KEY (id)
);
```

Cardinality & Participation Constraints

- Approach 1 (separate table): fails to capture mandatory participation!

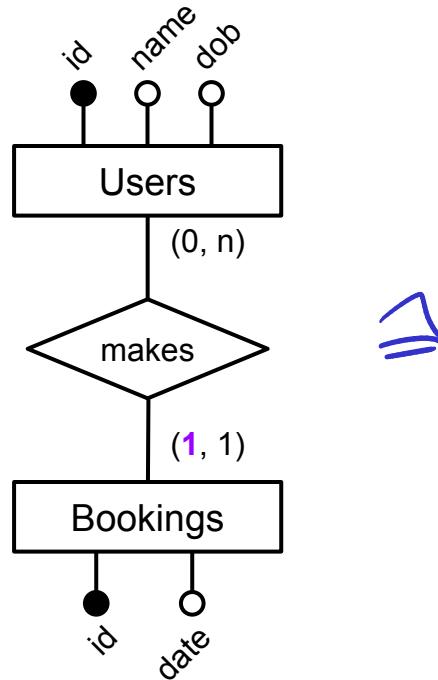


```
CREATE TABLE Makes (
    user_id      INTEGER NOT NULL,
    booking_id   INTEGER,
    PRIMARY KEY (booking_id),
    FOREIGN KEY (user_id) REFERENCES Users (id),
    FOREIGN KEY (booking_id) REFERENCES Bookings (id)
);
```

- Schema does not enforce mandatory participation of "Bookings" w.r.t. "Makes"
- e.g.: "Makes" can be empty while both "Users" and "Bookings" are non-empty

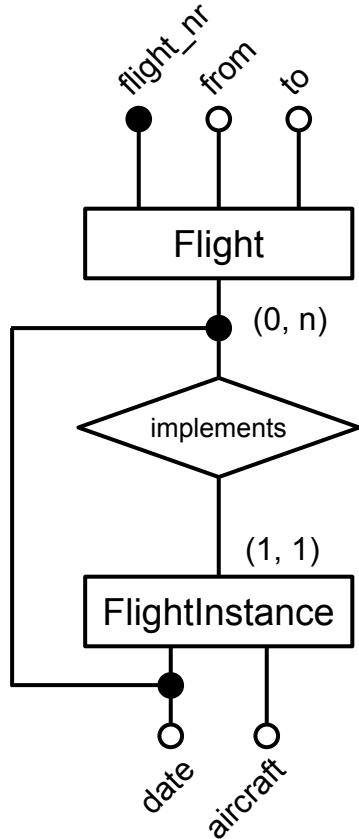
Cardinality & Participation Constraints

- **Approach 2:** Combine "makes" and "Bookings" into one table
 - Enforces total participation via NOT NULL constraint



```
CREATE TABLE Bookings (
    id      INTEGER,
    date   DATE,
    user_id INTEGER NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (user_id) REFERENCES Users (id)
);
```

Weak Entity Sets



```
CREATE TABLE Flights (
    flight_nr      VARCHAR(10),
    from           VARCHAR(10),
    to             VARCHAR(10),
    PRIMARY KEY (flight_nr)
);
```

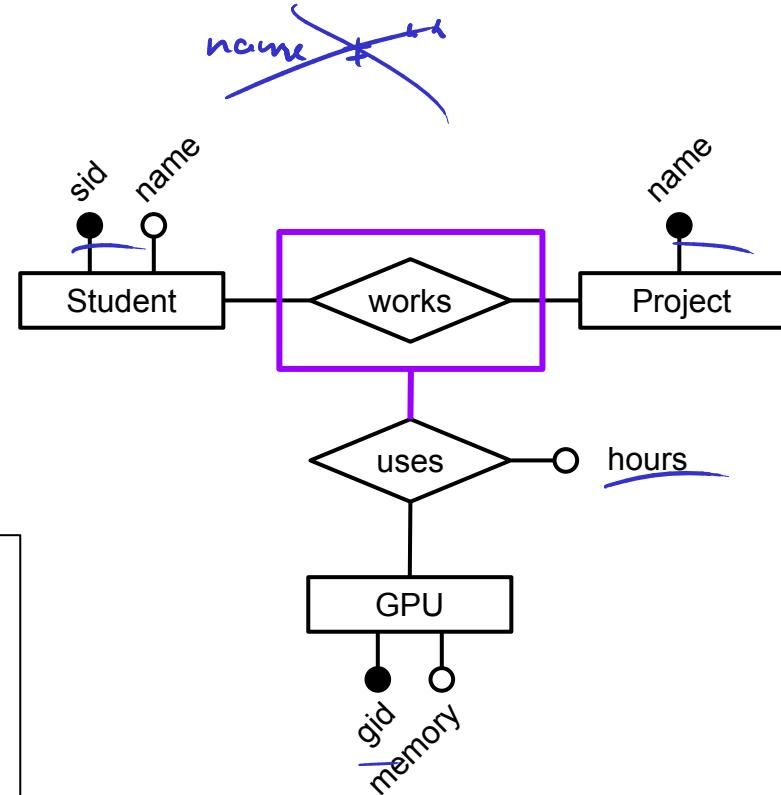
```
CREATE TABLE FlightInstances (
    flight_nr      VARCHAR(10),
    date           DATE,
    aircraft       VARCHAR(10),
    PRIMARY KEY (flight_nr, date),
    FOREIGN KEY (flight_nr) REFERENCES Flights (flight_nr)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

Aggregation — Relational Mapping

Schema definition of "uses"

- Primary key of aggregation relationship → (sid, pname)
- Primary key of associated entity set "GPUs" → gid
- Descriptive attributes of "uses" → hours

```
CREATE TABLE Uses (
    gid      INTEGER,
    sid      CHAR(20),
    pname    VARCHAR(50),
    hours   NUMERIC,
    PRIMARY KEY (gid, sid, pname),
    FOREIGN KEY (gid) REFERENCES GPUs (gid),
    FOREIGN KEY (sid, pname) REFERENCES works (sid, pname)
);
```



ER Design & Relational Mapping — Basic Guidelines

- Guidelines for ER design
 - An ER diagram should capture as many of the constraints as possible
 - An ER diagram must not impose any constraints that are not required
- Guidelines for relational mapping
 - (i.e., from ER diagram to relational database schema)
 - The relational schema should enforce as many if the constraints as possible using column and/or table constraints
 - The relational schema should not impose and constraints that are not required

Overview

- Entity Relationship Model
 - Overview + ER diagrams
 - Entity sets and attributes
 - Relationship sets
 - Cardinality & participation constraints
 - Dependency constraints: weak entity sets
 - Aggregation
- Relational Mapping
 - From ER diagram to database tables
- Summary

Summary

- Entity-Relationship (ER) model
 - Basic concepts: entity sets, relationship sets, attributes
 - Cardinality constraints and participation constraints
 - Extended concepts: (ISA hierarchies) aggregation ✓
 - Relational Mapping
 - Mapping ER diagram to database schema
 - Not all constraints of ER diagram may be captured
 - Outlook for next lecture
 - SQL for querying a database (recommendation: study RA)
- } Visualized using **ER diagrams**

Quick Quiz Solutions

Quick Quiz (Slide 3)

- Solution

- Storing the "dob" instead of "age" is arguably the preferred approach
- The value of "age" changes each year (not really a big deal)
- "dob" provides more detailed information compared the "age"

Quick Quiz (Slide 14)

- Solution

- Modeling "address" and "phone" as a single-values string might be OK-ish if we never use these attributes to select rows
- If we only need to get the address or all phone numbers for a given user then this solution might be good enough
- However, queries using "address" or "phone" to filter rows will become unnecessarily complicated or even impossible
- A query such as "Return all users with addresses with the ZIP code 123456" is possible since SQL supports string pattern matching and even regular expression. The performance would degrade, though.
- More intricate queries might still be formulated but the complexity of the SQL query would quickly blow up

Quick Quiz (Slide 17)

- Solution

- Using a fixed number of single-valued attributes avoids "splitting" the data across multiple tables (and avoids joining them as part of queries – costly operation)
- Two disadvantages when a fixed number of single-valued attributes
 - Unable to store then 2 phone numbers
 - Requires storing NULL values if user does not have exactly 2 phone numbers
- A separate table only stores the information needed and is more flexible, but will relies on join operations to bring the information together

Quick Quiz (Slide 18)

- Solution
 - Complex data types are generally more difficult to query
 - For example: How to check the value for an optional field in a JSON document?
Maybe be possible but often requires more complex and non-standard syntax

Quick Quiz (Slide 34)

- Solution
 - 0/1/n constraints can typically be captured using basic integrity constraints (as shown later)
 - Any more specific upper and lower bounds require more integrity checks, particularly since these constraints involve more than 1 table
 - General solution: **triggers**
 - Also not uncommon: don't use DBMS to enforce constraints

Quick Quiz (Slide 49)

- Solution
 - "Flight Instances" is weak entity set with "Flights" being the owner entity set
 - Thus, "Flight Instances" is identified by the key of "Flights" (i.e., "fnr") and its own partial key "date"

Quick Quiz (Slide 51)

- Solution
 - Approach 2 is generally the preferred approach as it leads to a smaller number of tables
 - Less tables also means that queries might need less join operations
(which are typically the more expensive operations)
 - Good rule of thumb but not a "law"

Quick Quiz (Slide 54)

- Solution
 - Access privileges can (mostly) only be set on the table level
 - Separating the basic user data and the credit card details allows assign different access privilege to different users
 - Also, separate table avoid NULL values if some users do not have a credit card

CS2102: Database Systems

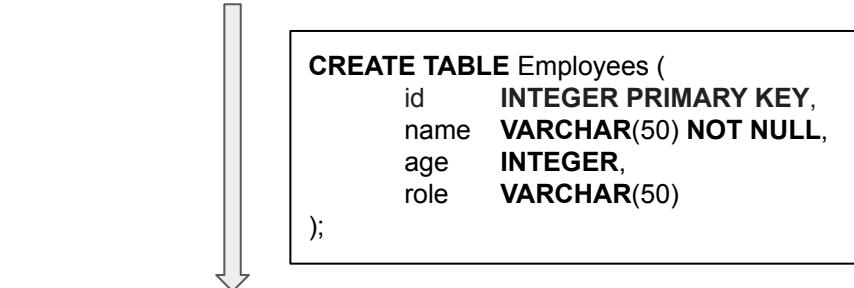
Lecture 3 — Entity Relationship Model (ER Model)

Quick Recap: SQL for Creating Databases

- **Data Definition Language (DDL)**

- Create, modify and drop tables to implement a given DB schema
- Specify integrity constraints (e.g., NOT NULL, PRIMARY KEY, FOREIGN KEY, CHECK)

Employees (`id: integer`, `name: text`, `age: integer`, `role: text`)



Employees

id	name	age	role
----	------	-----	------

- **Data Manipulation Language (DML)**

- Insert, update and delete data from tables

INSERT INTO Employees VALUES
(101, 'Sarah', 25, 'dev')
(102, 'Judy', 35, 'sales');

Employees

id	name	age	role
101	Sarah	25	dev
102	Judy	35	sales

We Sneakily Skipped a Step

- Open questions:
 - Where does the database schema come from?
 - What tables with which attributes do we need?
 - What data integrity constraints are required?
 - Table names, attribute names, data types, ...?

→ Database Design Process

Quick Quiz: Which table is "better"?

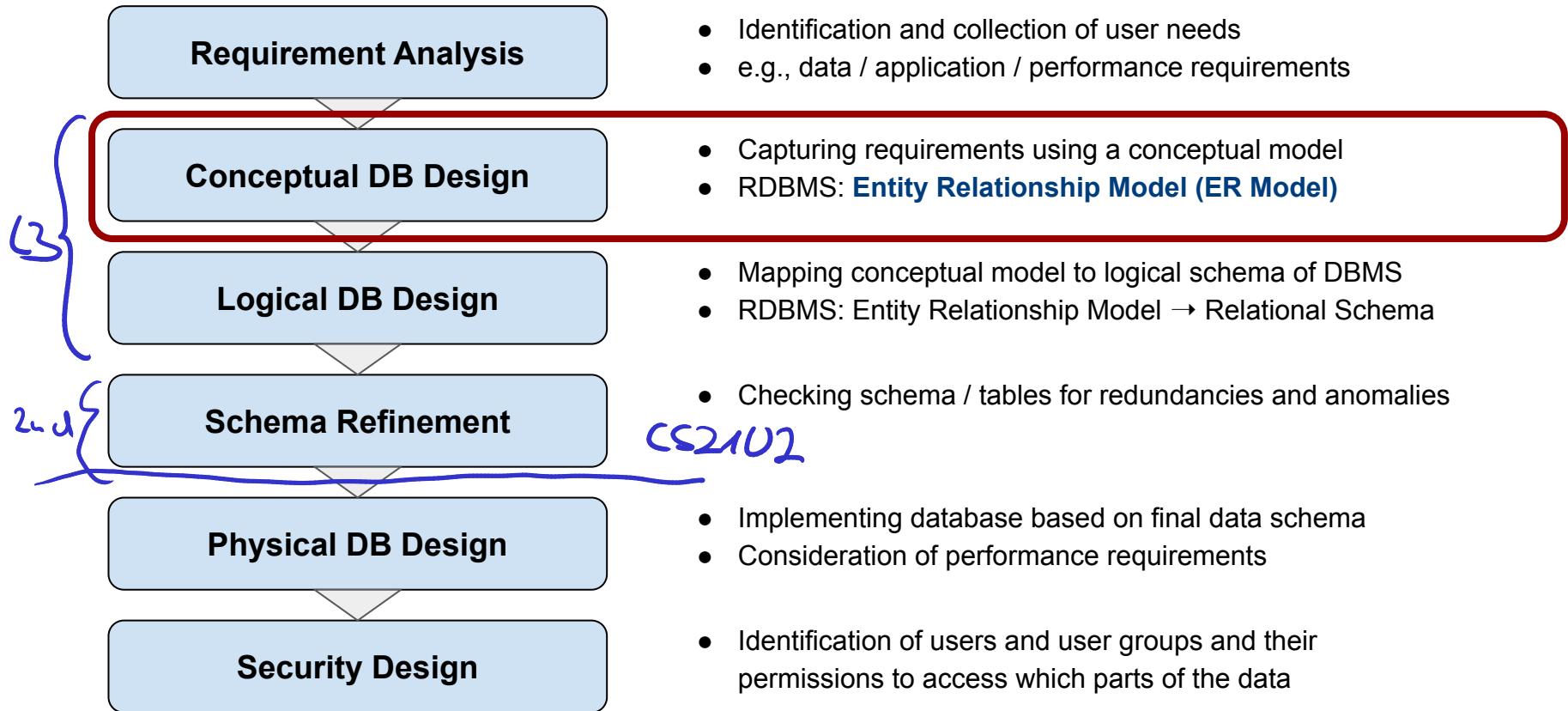
```
CREATE TABLE Employees (
    id      INTEGER PRIMARY KEY,
    name   VARCHAR(50) NOT NULL,
    age    INTEGER,
    role   VARCHAR(50)
);
```

or

```
CREATE TABLE Employees (
    id      INTEGER PRIMARY KEY,
    name   VARCHAR(50) NOT NULL,
    dob    DATE,
    role   VARCHAR(100)
);
```



Database Design Process — 6 Common Steps



Overview

- **Entity Relationship Model**
 - Overview + ER diagrams
 - Entity sets and attributes
 - Relationship sets
 - Cardinality & participation constraints
 - Dependency constraints: weak entity sets
 - Aggregation
- **Relational Mapping**
 - From ER diagram to database tables
- **Summary**

Requirement Analysis: Online Airline Reservation System (OARS)

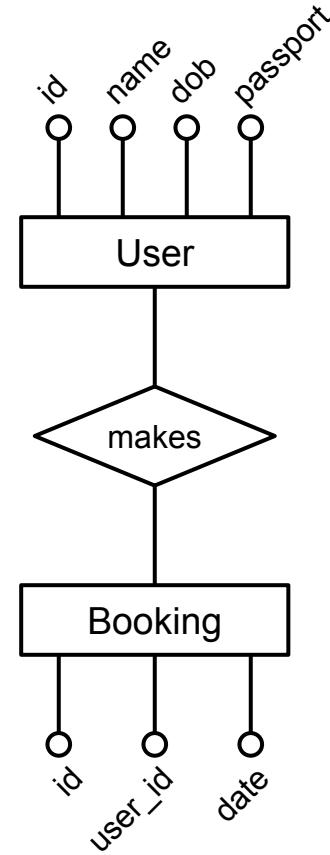
Users need to be able to make bookings from an origin to a destination airport which may comprise multiple connecting flights. Each flight has a flight number, the origin and destination airport, the distance in kilometers, the departure and arrival time, and the days of the week the flight is in operation.

A flight instance is the actual scheduled flight on a given day together with the assigned aircraft type. For example, flight SQ231 flies daily from Singapore to Sydney, typically with a Boeing 777-300ER (code: B77W).

For a valid booking, we need the user's name, sex, address, phone number(s), and the passport number. Users are only able to pay via credit card. When making a booking, the user can select the class, the seat number, as well as meal preferences (if available).

Entity Relationship Model

- ER Model
 - Most common model for conceptual database design
 - Developed by Peter Chen (1976)
 - Visualized using **ER diagrams**
(Important: many revised version – no one single set of notations!)
- Core concepts
 - All data is described in terms of **entities** and their **relationships**
 - Information about entities & relationships are described using **attributes**
 - Certain data constraints can be described using additional annotations



Overview

- **Entity Relationship Model**
 - Overview + ER diagrams
 - **Entity sets and attributes**
 - Relationship sets
 - Cardinality & participation constraints
 - Dependency constraints: weak entity sets
 - Aggregation
- **Relational Mapping**
 - From ER diagram to database tables
- **Summary**

Entities and Entity Sets

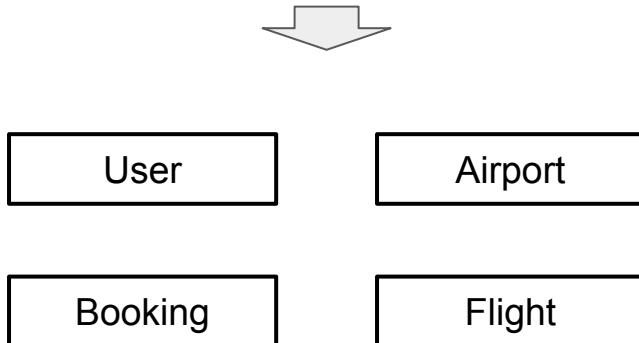
- **Entity**

- Real-world things or objects that are distinguishable from other objects
(e.g., an individual user, airport, flight, or booking)

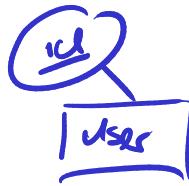
Users need to be able to make bookings from an origin to a destination airport which may comprise multiple connecting flights. Each flight has a flight number, [...]

- **Entity Set**

- Collection of entities of the same type
- Represented by rectangles in ER diagrams
- Names are typically nouns



Attributes



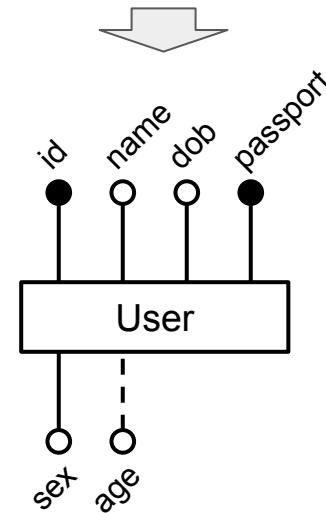
- **Attribute:**

- specific information describing an entity
- represented by a small circle in ER diagrams

- 2 main subtypes of attributes

- **Key attribute(s):** uniquely identifies each entity
 - Indicated by a filled circle in ER diagram
 - Different attributes may uniquely identify an entity
 - Multiple attributes may form a composite key
- **Derived attribute:** derived from other attributes
 - Indicated by a dashed line in ER diagram
 - Example: derive "age" from "dob"

For a valid booking, we need the **user's name, sex, address, phone number(s), and the passport number.**
Users are only able to pay via credit card. [...]



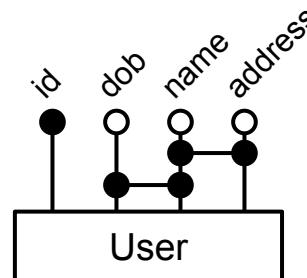
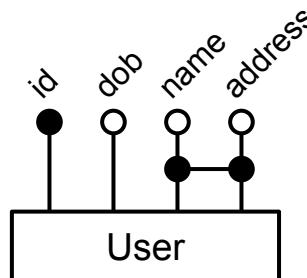
What about address and phone numbers?

Key Attributes

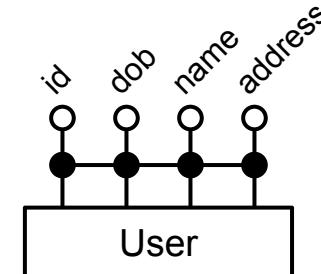
- **Composite key attributes:**

- 2 or more attributes together uniquely identify each entity
- An entity may have multiple composite key attributes
- Representation in ER diagram: additional connecting line

- **Examples** (for illustration purposes; not necessarily realistic!)

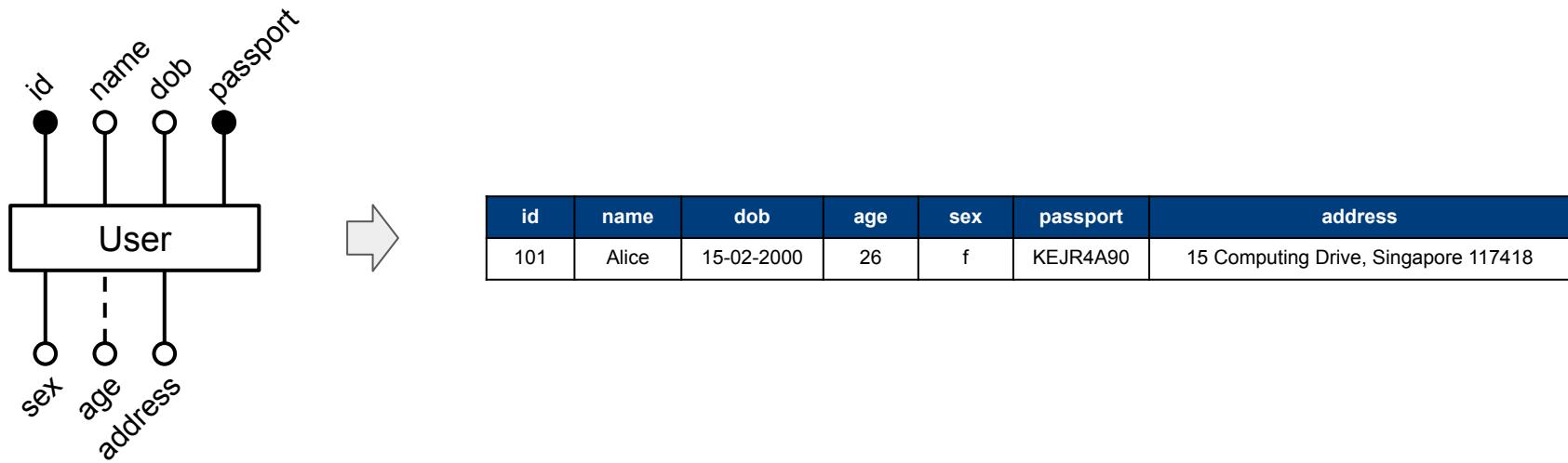


- At least all attributes uniquely identify an entity
- We typically prefer a minimum set of attributes



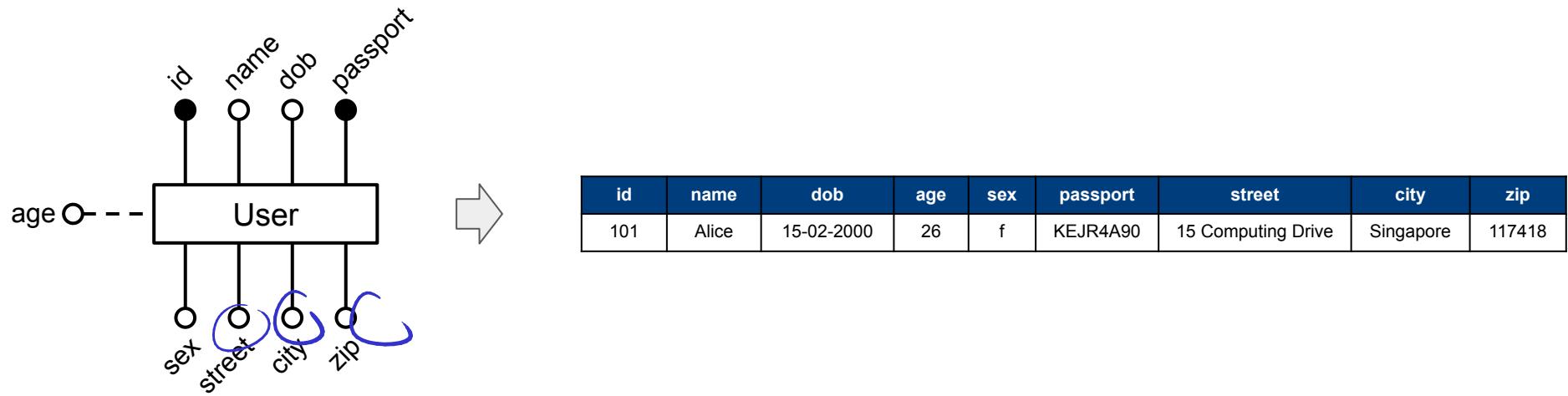
"Composite" Attributes

- Common: requirement analysis often vague / ambiguous / unclear
 - Not always obvious how certain attributes should be modeled
 - Example "address": single string attribute vs. multiple attributes



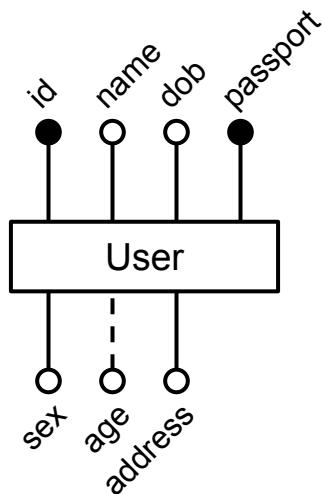
"Composite" Attributes

- Common: requirement analysis often vague / ambiguous / unclear
 - Not always obvious how certain attributes should be modeled
 - Example "address": single string attribute vs. multiple attributes

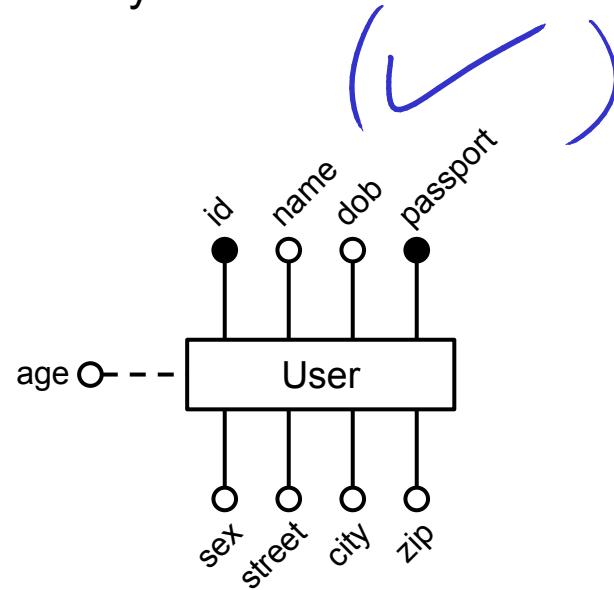


Quick Quiz

Which solution is typically the **preferred** one?
But always? And why?

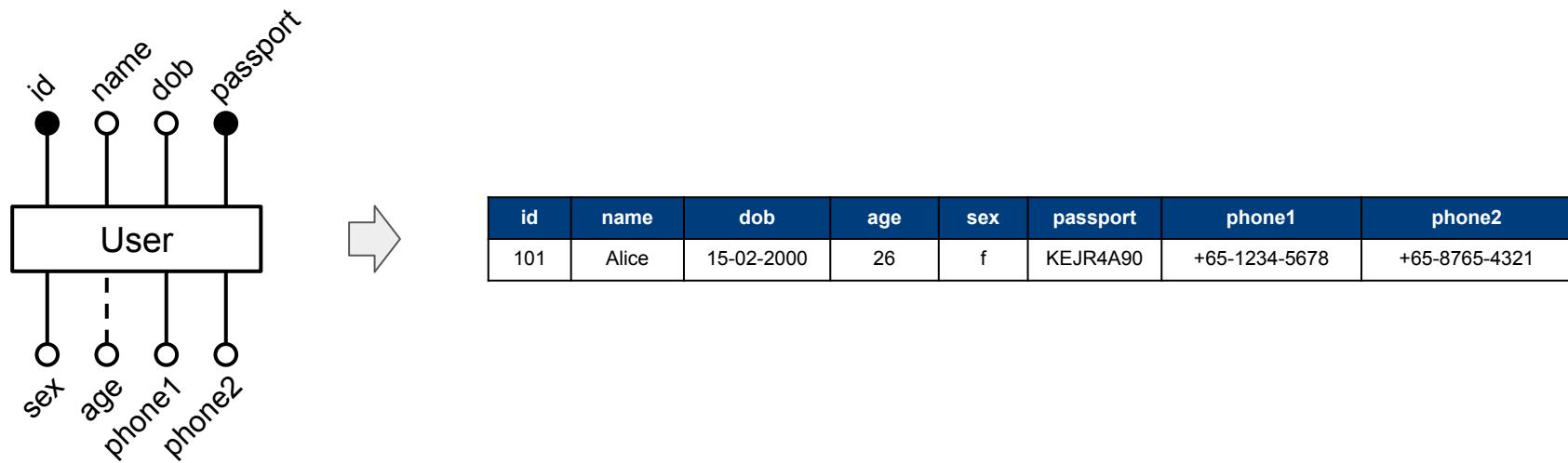


vs



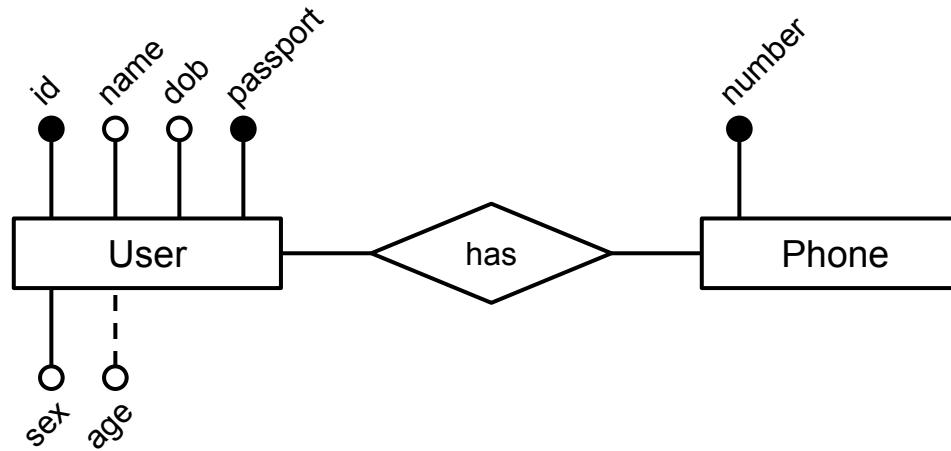
Multivalued Attributes

- Common: an attribute may refer to a set/list of values
 - Examples: phone numbers, hobbies, tags/keywords
 - However: all attributes must be single-valued
 - Example "phone numbers": fixed number of single-valued attributes vs. dedicated entity set



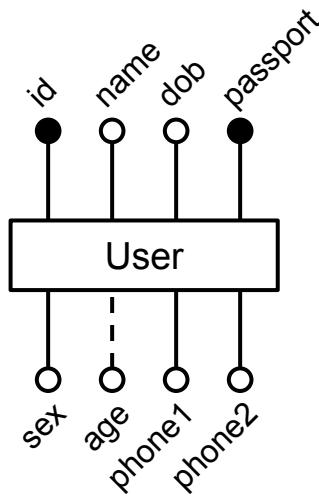
Multivalued Attributes

- Common: an attribute may refer to a set/list of values
 - Examples: phone numbers, hobbies, tags/keywords
 - However: all attributes must be single-valued
 - Example "phone numbers": fixed number of single-valued attributes vs. dedicated entity set

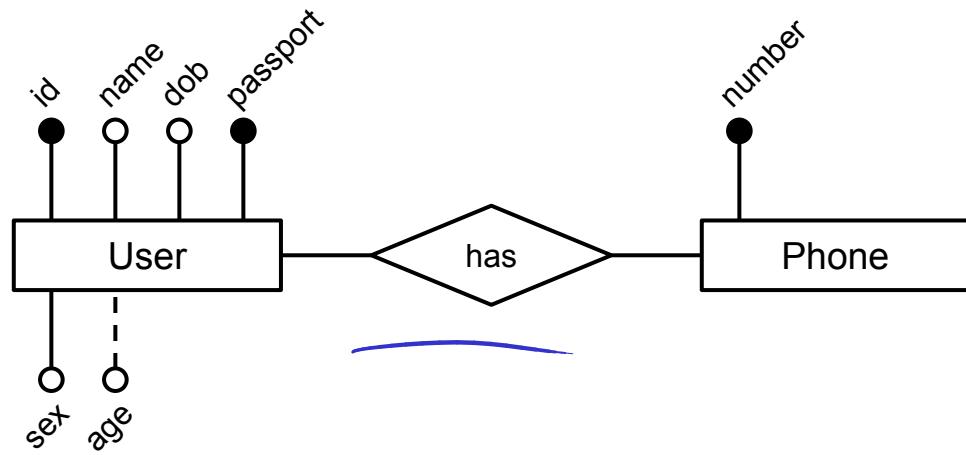


Quick Quiz

Which solution is typically the **preferred** one?
But always? And why?



vs



Side Note

- PostgreSQL (and most modern RDBMS)
 - Not limited to basic single-valued data types
 - Support for complex / composite data types
 - Support for user-defined composite types

Quick Quiz: What are potential downsides of this more complex data types?

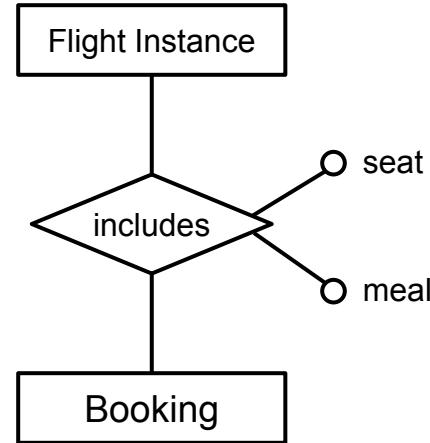
8.13. XML Type	8.13.1. Creating XML Values
	8.13.2. Encoding Handling
	8.13.3. Accessing XML Values
8.14. JSON Types	8.14.1. JSON Input and Output Syntax
	8.14.2. Designing JSON Documents
	8.14.3. jsonb Containment and Existence
	8.14.4. jsonb Indexing
	8.14.5. jsonb Subscripting
	8.14.6. Transforms
	8.14.7. jsonpath Type
8.15. Arrays	8.15.1. Declaration of Array Types
	8.15.2. Array Value Input
	8.15.3. Accessing Arrays
	8.15.4. Modifying Arrays
	8.15.5. Searching in Arrays
	8.15.6. Array Input and Output Syntax
8.16. Composite Types	8.16.1. Declaration of Composite Types
	8.16.2. Constructing Composite Values
	8.16.3. Accessing Composite Types
	8.16.4. Modifying Composite Types
	8.16.5. Using Composite Types in Queries
	8.16.6. Composite Type Input and Output Syntax

Overview

- **Entity Relationship Model**
 - Overview + ER diagrams
 - Entity sets and attributes
 - **Relationship sets**
 - Cardinality & participation constraints
 - Dependency constraints: weak entity sets
 - Aggregation
- **Relational Mapping**
 - From ER diagram to database tables
- **Summary**

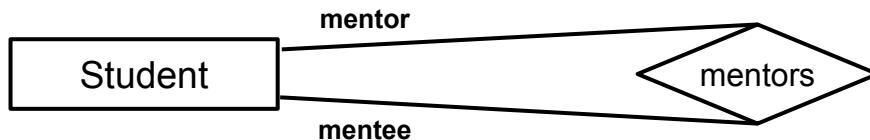
Relationships and Relationship Sets

- **Relationship**
 - Association among two or more entities
- **Relationship Set**
 - Collection of relationships of the same type
 - Represented by diamonds in ER diagrams
 - Can have their own attributes that further describe the relationship
 - Names are typically verbs
- **Additional annotations to further specify relationships**
 - Roles, degree, cardinalities, participation, dependencies



Relationship Roles

- **Role**
 - Descriptor of an entity set's participation in a relationship
 - Most of the time implicitly given by the name of the entity sets
 - Explicit role labels only common in case of ambiguities
(typically in case the same entity sets participate in the same relationship more than once)
- Example: Students can mentor other students

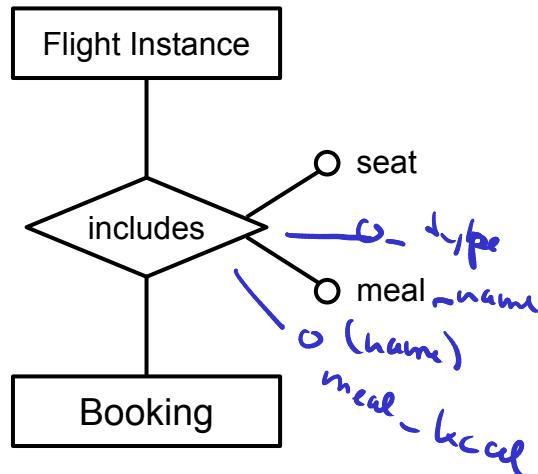


Degree of Relationship Sets

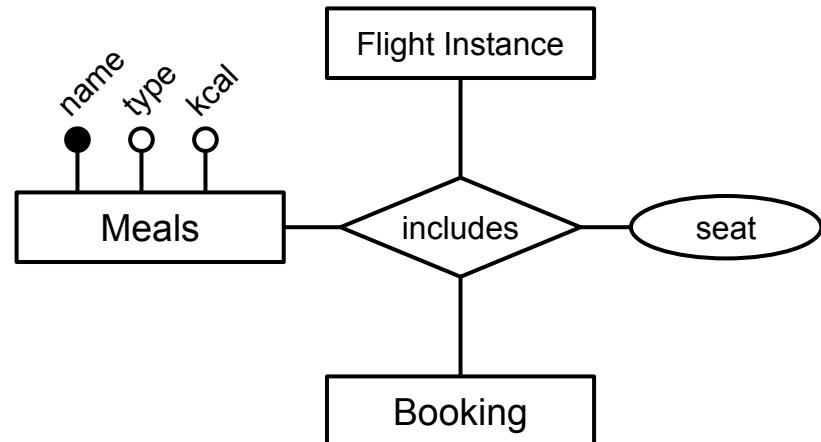
- **Degree**

- In principle, no limitation on how many entity roles participate in a relationship
- An n -ary relationship set involves n entity roles → $n =$ degree of relationship set

$n = 2 \rightarrow$ binary relationship set

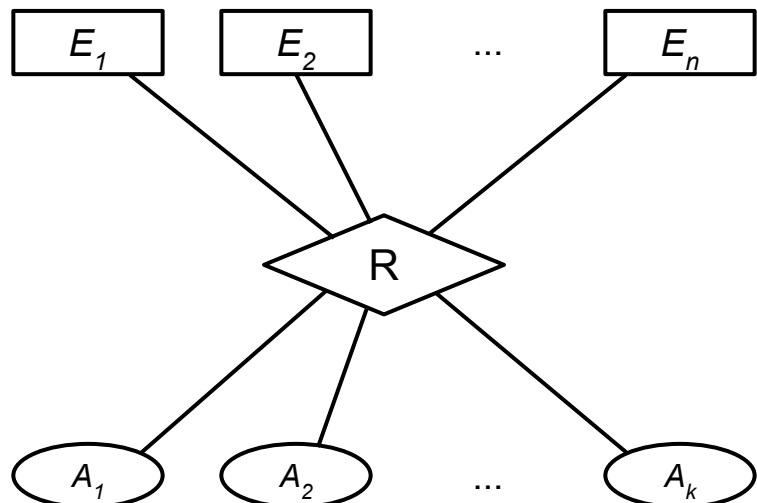


$n = 3 \rightarrow$ ternary relationship set



Degree of Relationship Sets

- General n-ary relationship set R
 - n participating entity sets E_1, E_2, \dots, E_n
 - k relationship attributes A_1, A_2, \dots, A_k



"In typical modeling, binary relationships are the most common and relationships with $n > 3$ are very rare" - Peter Chen (2009)

Overview

- **Entity Relationship Model**
 - Overview + ER diagrams
 - Entity sets and attributes
 - Relationship sets
 - **Cardinality & participation constraints**
 - Dependency constraints: weak entity sets
 - Aggregation
- **Relational Mapping**
 - From ER diagram to database tables
- **Summary**

Cardinality & Participation Constraints

- **Cardinalities of Relationship Sets**

- Describe how often an entity can participate in a relationship at most
- 3 basic cardinality constraints
 - **Many-to-many** (e.g., a flight can be performed by different aircrafts; an aircraft can perform different flights)
 - **Many-to-one** (e.g., a user can make many bookings, but each booking is done by one user)
 - **One-to-one** (e.g., a user is associated with one set of credit card details, and vice versa)

upper bound

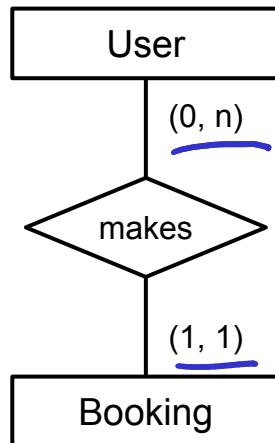
- **Participation constraints**

- Describe how often an entity has to participate in a relationship at least
- Is the participation of an entity in a relationship even mandatory?

lower bound

Cardinality & Participation Constraints

- Representation in ER diagram
 - (min,max) label at connections between entity and relationship sets



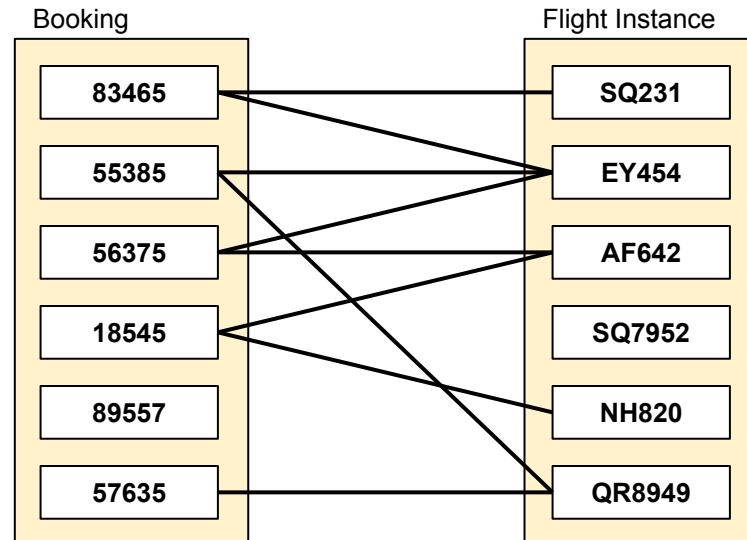
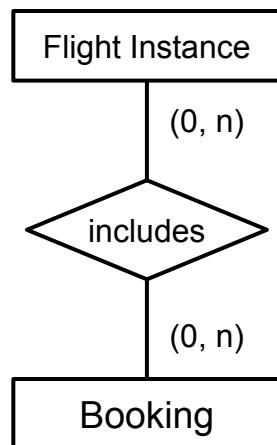
Interpretation

- Each user can make multiple bookings
(but not every user must have made a booking)
- Each booking was done by exactly one user
(implies that each booking is associated with a user)

Cardinality: Many-to-Many (no mandatory participation)

- Many-to-many relationship between bookings and flight instances

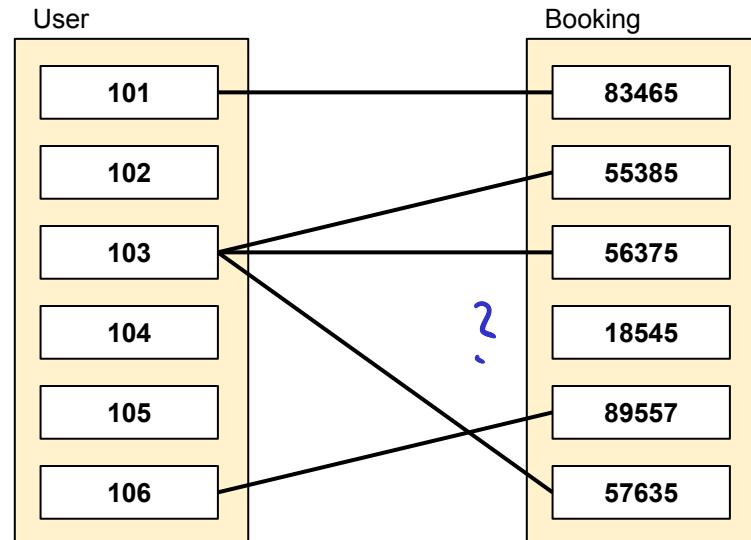
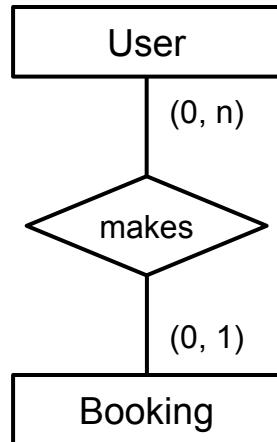
- Each booking can include 0 or more flight instances
(note that a booking with 0 flights might not be meaningful; we will improve on that)
- Each flight instance can be part of 0 or more bookings



Cardinality: Many-to-One (no mandatory participation)

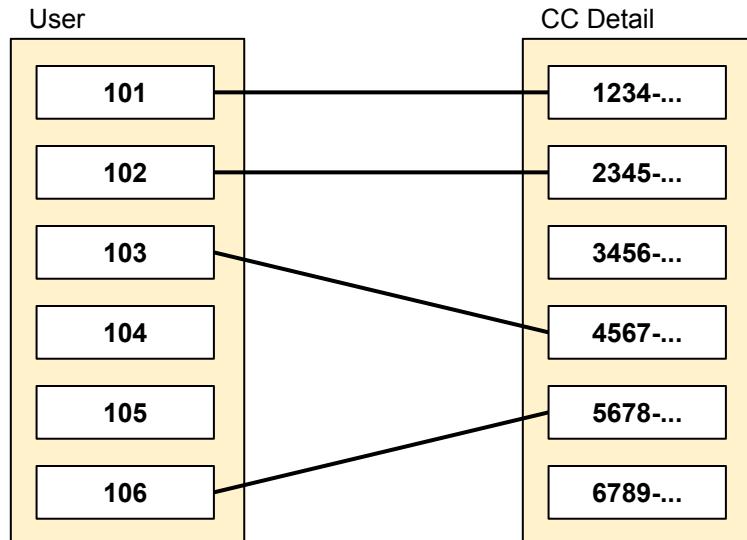
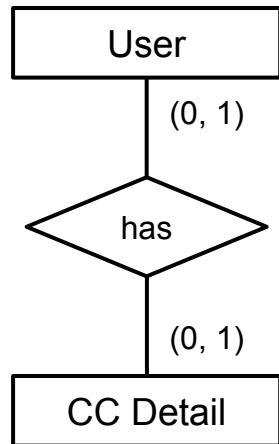
- Many-to-one relationship between users and bookings

- Each user can make 0 or more bookings
- Each booking is done by one 1 user at most
(again, not perfect yet, and we will improve on that)



Cardinality: One-to-One (no mandatory participation)

- One-to-one relationship between users and credit card details
 - Each user can provide only 1 set of credit card details at most
 - Each set of credit card details is associated with 1 user at most



Participation Constraints

- Limitation of (basic) cardinality constraints from previous examples

- A booking can include 0 flights
- A booking can be done by 0 users
- A set of credit card details does not need to be associated with a user

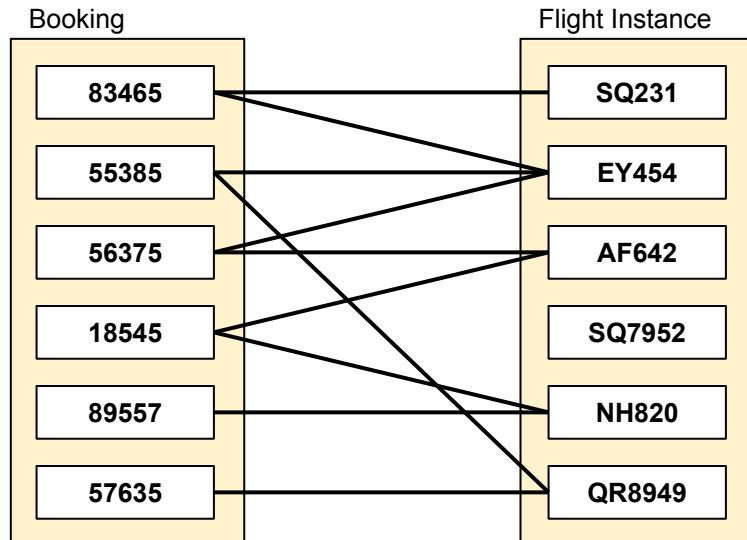
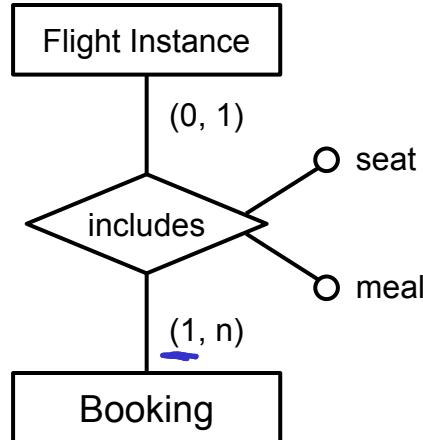


an entity does not have to participate in a relation

→ Let's include **participation constraints**

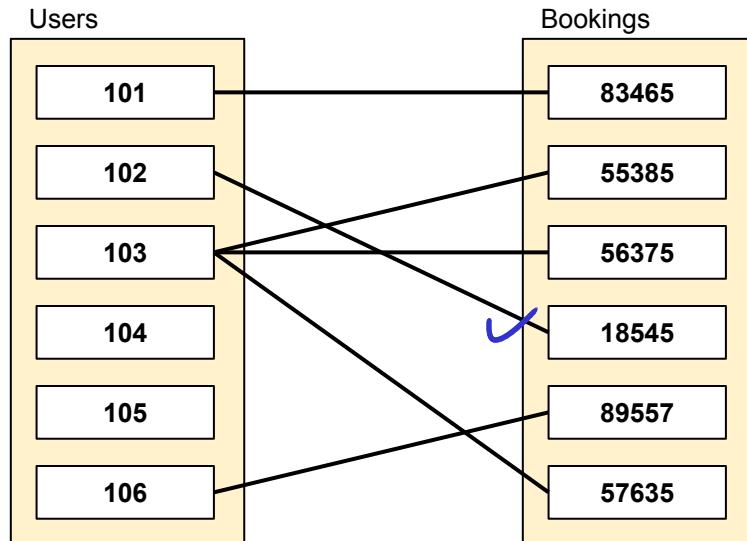
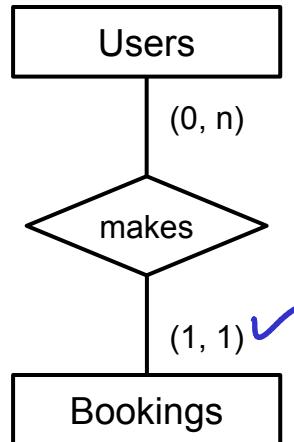
Cardinality & Participation Constraints

- Many-to-many relationship between bookings and flight instances
 - Each booking includes 1 or more flight instances
 - Each flight instance can be part of 0 or more bookings



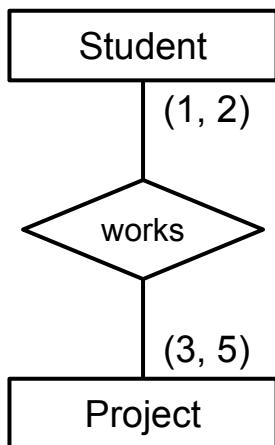
Cardinality & Participation Constraints

- Many-to-one relationship between users and bookings
 - Each user can make 0 or more bookings
 - Each booking is done by exactly 1 user



Cardinality & Participation Constraints

- Flexibility of (min,max) notation
 - Minimum not limited to 0 or 1; maximum not limited to n
 - Arbitrary specific values to capture real-world constraint



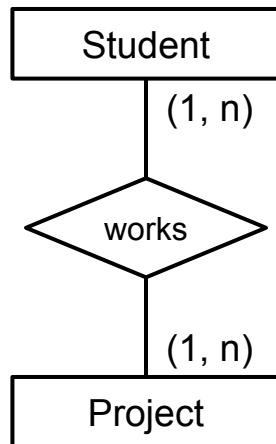
Interpretation

- Each student must work on at least 1 project
- Each student may not work on more than 2 projects
- Each project consists of at least 3 students
- Each project may not consist of more than 5 students

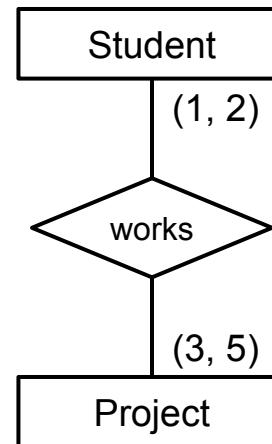
Quick Quiz

Why do values other than 0/1/n add significant complexity?

(just think about it for a minute here; we will cover it later)



vs



→ triggers

Overview

- **Entity Relationship Model**

- Overview + ER diagrams
- Entity sets and attributes
- Relationship sets
- Cardinality & participation constraints
- **Dependency constraints: weak entity sets**
- Aggregation

- **Relational Mapping**

- From ER diagram to database tables

- **Summary**

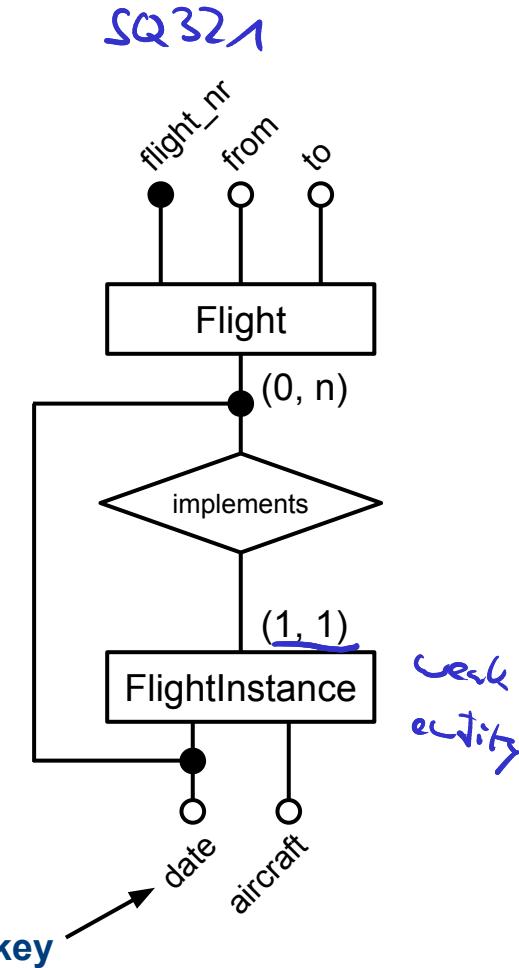
Dependency Constraints

- **Weak entity sets**
 - Entity set that does not have its own key
 - A weak entity can only be uniquely identified by considering the primary key of the **owner entity**
 - A weak entity's existence depends on the existence of its owner entity
 - Weak entity set and identifying relation set are represented via double-lined rectangles / diamonds
- **Requirements**
 - Many-to-one relationship (identifying relationship) from weak entity set to owner entity set
(one-to-one possible but less common)
 - Weak entity set must have (1, 1) attached to identifying relationship

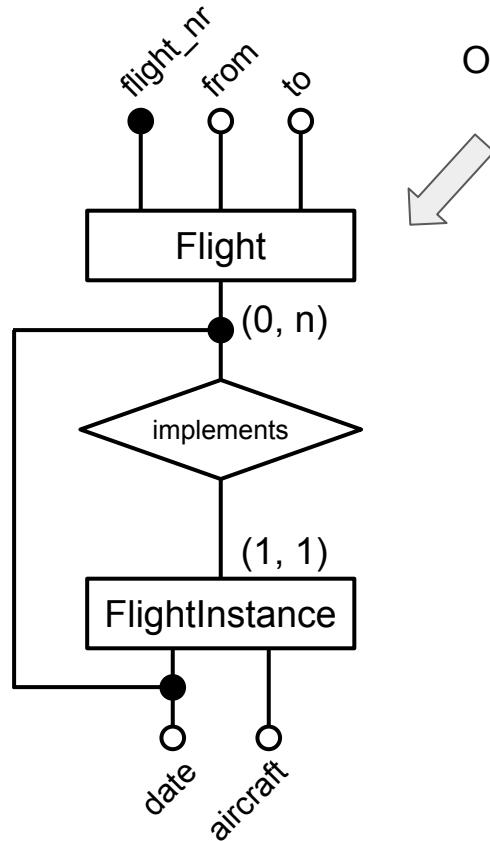
Dependency Constraints

- Example

- A flight instance is the actual scheduled flight (with a unique flight number) on a given day
 - Each flights instance is identified by the "flight_nr and the "date"
 - "date" is a **partial key**
- A flight instance cannot "exist" without the flight

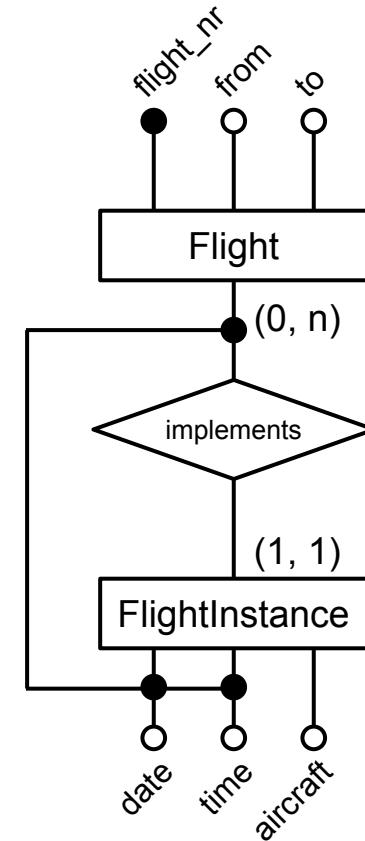


Dependency Constraints



Only 1 flight (instance)
per day maximum

More than 1 flight
(instance) per day

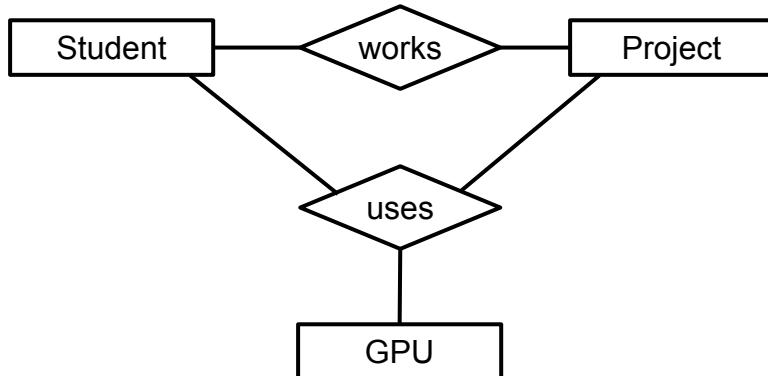


Overview

- **Entity Relationship Model**
 - Overview + ER diagrams
 - Entity sets and attributes
 - Relationship sets
 - Cardinality & participation constraints
 - Dependency constraints: weak entity sets
 - **Aggregation**
- **Relational Mapping**
 - From ER diagram to database tables
- **Summary**

Extended Concepts — Aggregation

- Concepts of ER diagrams so far
 - Only relationships between entity sets
 - No relationships between entity sets and relationship sets
- Motivating example



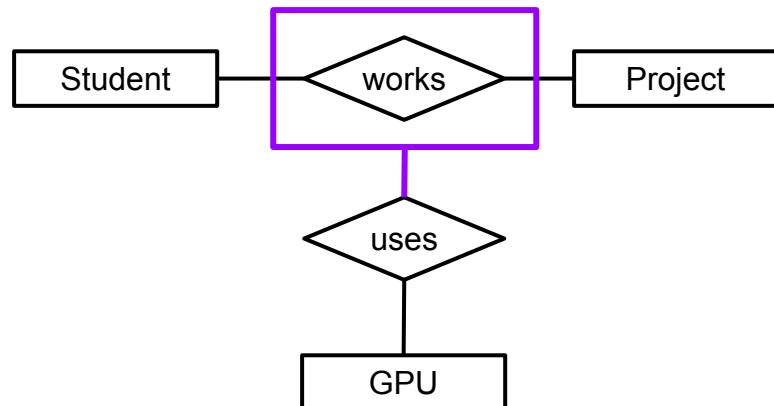
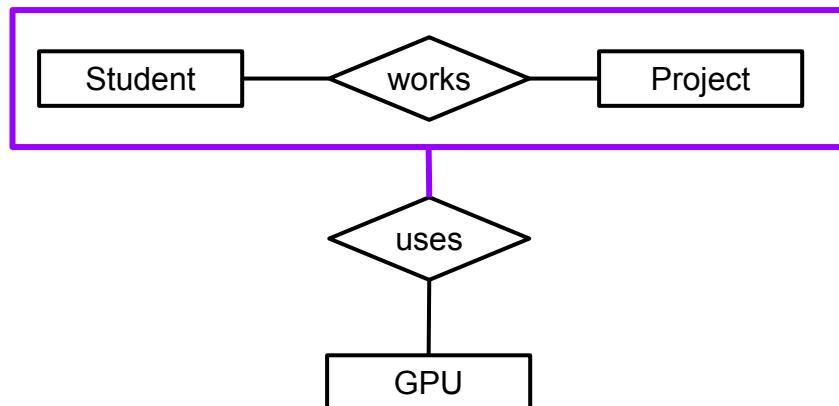
Limitations:

- Relationship between "works" and "uses" not explicitly captured
- "works" and "uses" are kind of redundant relationships

→ **Aggregation**

Extended Concepts — Aggregation

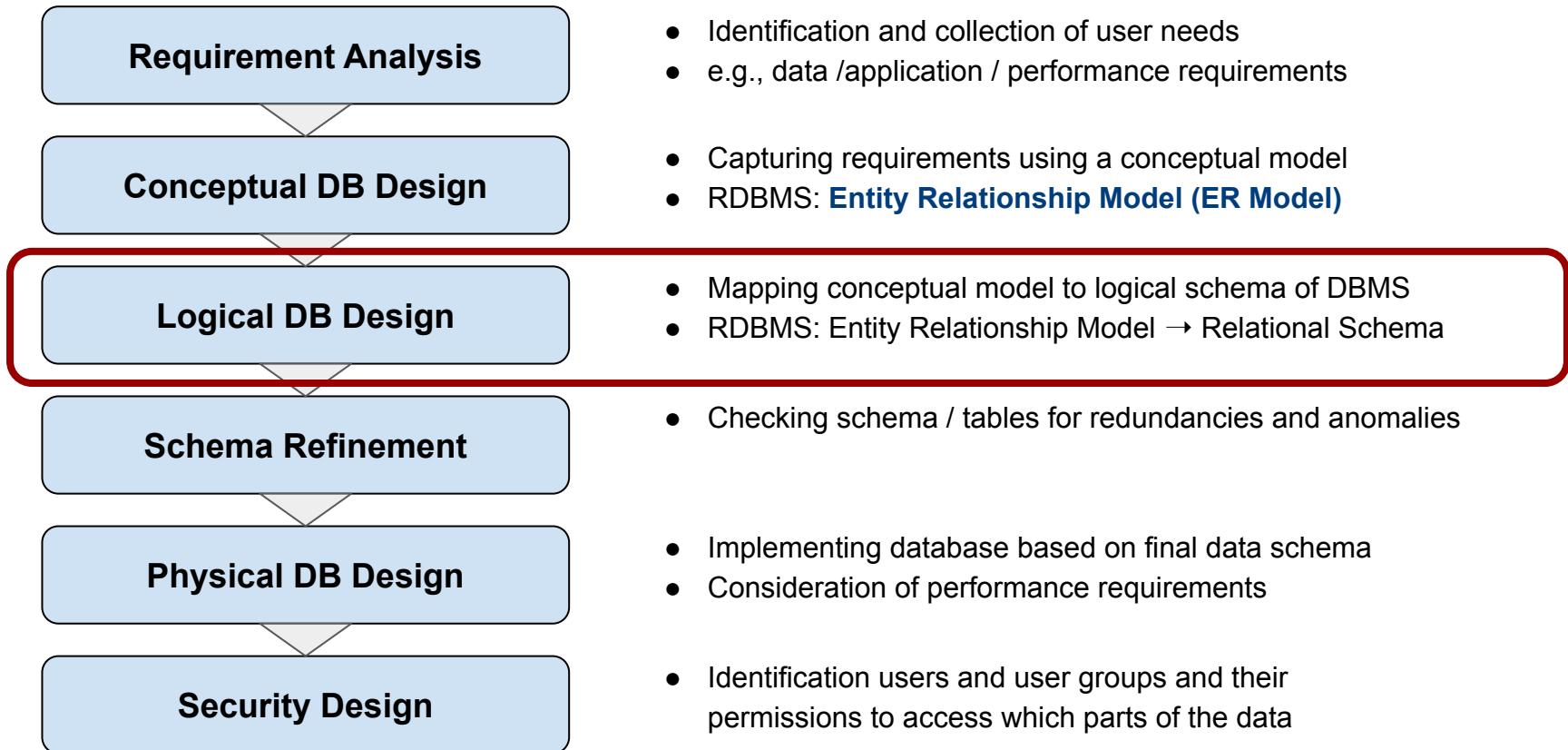
- Aggregation — basic idea
 - Abstraction that treats relationships as higher-level entities
 - Example: treat Students-works-Projects as an entity set
- Notation in ER diagram (2 equivalent alternatives)



Overview

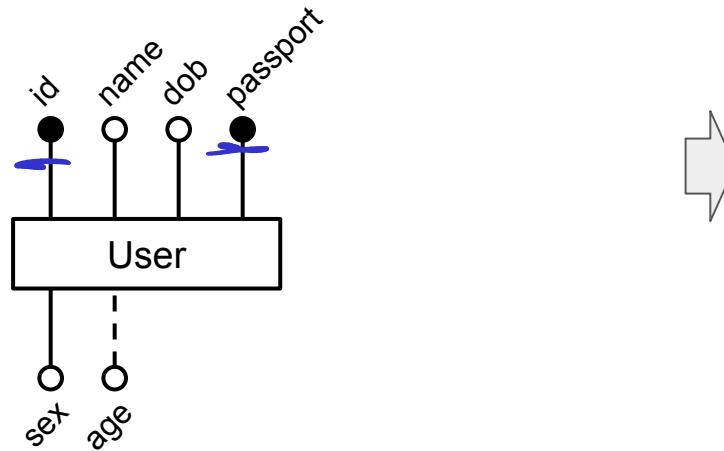
- Entity Relationship Model
 - Overview + ER diagrams
 - Entity sets and attributes
 - Relationship sets
 - Cardinality & participation constraints
 - Dependency constraints: weak entity sets
 - Aggregation
- Relational Mapping
 - From ER diagram to database tables
- Summary

Database Design Process — 6 Common Steps



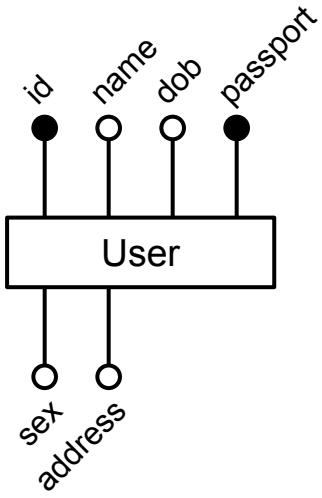
Entity Sets

- Straightforward mapping from entity sets to tables (except for composite & multivalued attributes)
 - Name of entity set → name of table
 - Attributes of entity set → attributes of table
 - Key attributes of entity set → primary key of table

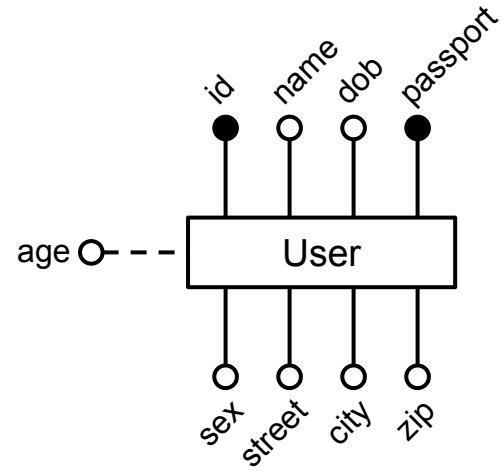


```
CREATE TABLE Users (
    id      INTEGER,
    name    VARCHAR(100),
    dob     DATE,
    sex     CHAR(1),
    age     INTEGER,
    passport VARCHAR(20),
    PRIMARY KEY (id),
    UNIQUE (passport)
);
```

Note: PostgreSQL supports [Generated Column](#) but there are some caveats when used in practice that are beyond our scope.



VS

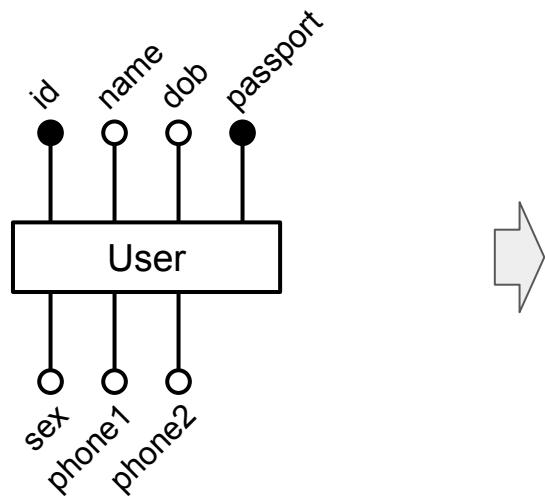


```
CREATE TABLE Users (
    id      INTEGER,
    name   VARCHAR(100),
    dob    DATE,
    sex    CHAR(1)
    passport VARCHAR(20),
    address VARCHAR(200),
    PRIMARY KEY (id),
    UNIQUE (passport)
);
```

```
CREATE TABLE Users (
    id      INTEGER,
    name   VARCHAR(100),
    dob    DATE,
    sex    CHAR(1)
    passport VARCHAR(20),
    street  VARCHAR(100),
    city   VARCHAR(100),
    zip    VARCHAR(10),
    PRIMARY KEY (id),
    UNIQUE (passport)
);
```

Multivalued Attributes

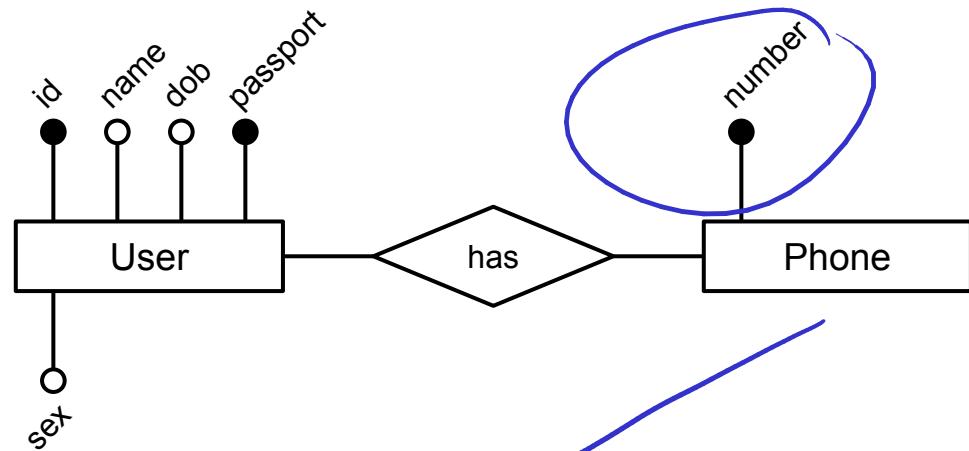
- Fixed number of single-valued attributes



```
CREATE TABLE Users (
    id      INTEGER,
    name   VARCHAR(100),
    dob    DATE,
    sex    CHAR(1),
    passport VARCHAR(20),
    phone1 VARCHAR(20),
    phone2 VARCHAR(200),
    PRIMARY KEY (id),
    UNIQUE (passport)
);
```

Multivalued Attributes

- Separate entity set for phone numbers

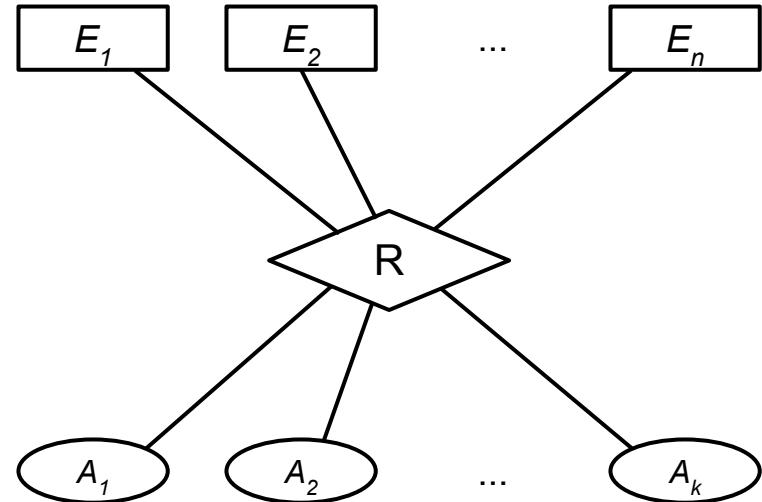


```
CREATE TABLE Users (
    id      INTEGER,
    name   VARCHAR(100),
    dob    DATE,
    sex    CHAR(1),
    passport VARCHAR(20),
    PRIMARY KEY (id),
    UNIQUE (passport)
);
```

```
CREATE TABLE Phones (
    number  VARCHAR(20),
    user_id INTEGER,
    PRIMARY KEY (number),
    FOREIGN KEY (user_id) REFERENCES Users (id)
);
```

Relationship Sets

- General n-ary relationship set R
 - n participating entity sets E_1, E_2, \dots, E_n
 - k relationship attributes A_1, A_2, \dots, A_k
 - Let $\text{Key}(E_i)$ be the attributes of the selected key of entity set E_i



→ Attributes of relationship set R

- $\text{Key}(E_1), \text{Key}(E_2), \dots, \text{Key}(E_n)$ — key attributes of all participating entity sets E_i
- A_1, A_2, \dots, A_k — all relationship attributes of R

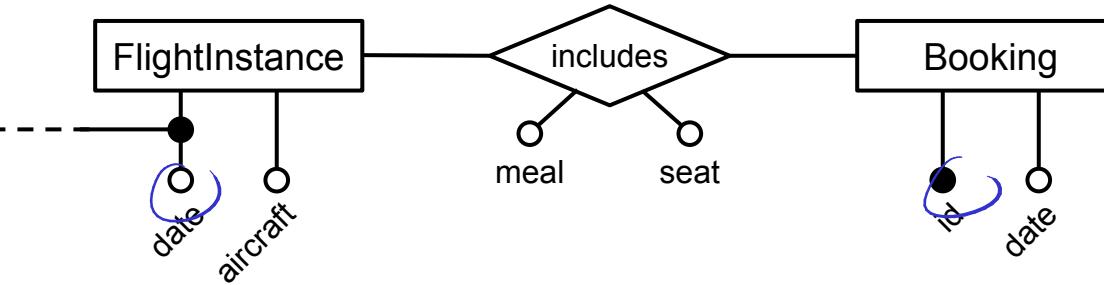
Cardinality: Many-to-Many (no mandatory participation)

Quick Quiz: Where does "flight_nr" come from?

CREATE TABLE FLIGHt (

fur
date
:
;

);



CREATE TABLE Includes (

flight_nr VARCHAR(10),
flight_date DATE,
booking_id INTEGER,
seat VARCHAR(10),
meal VARCHAR(50)

PRIMARY KEY (flight_nr, flight_date, booking_id),

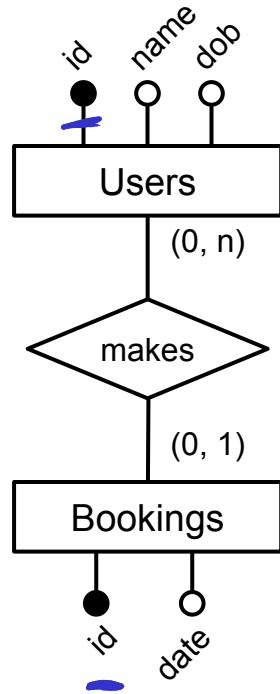
FOREIGN KEY (flight_nr, flight_date) **REFERENCES** FlightInstances (flight_nr, date),

FOREIGN KEY (booking_id) **REFERENCES** Bookings (id),

);

Cardinality: Many-to-One (no mandatory participation)

- Approach 1: Represent "makes" with a separate table
 - Similar to Many-to-Many but with different primary key!

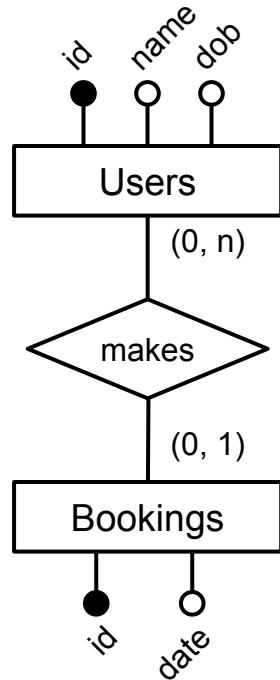


```
CREATE TABLE Makes (
    user_id    INTEGER,
    booking_id INTEGER,
    PRIMARY KEY (booking_id),
    FOREIGN KEY (user_id) REFERENCES Users (id),
    FOREIGN KEY (booking_id) REFERENCES Bookings (id)
);
```

Cardinality: Many-to-One (no mandatory participation)

Quick Quiz: What is typically the preferred approach? 1 or 2?

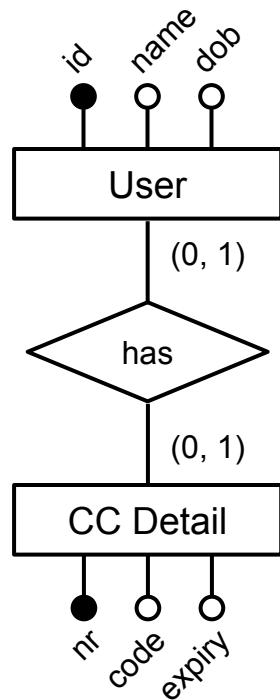
- **Approach 2: Combine "makes" and "Bookings" into one table**
 - Possible because given a booking, we can uniquely identify the user who made it



```
CREATE TABLE Bookings (
    id      INTEGER,
    date   DATE,
    user_id INTEGER,
    PRIMARY KEY (id),
    FOREIGN KEY (user_id) REFERENCES Users (id)
);
```

Cardinality: One-to-One (no mandatory participation)

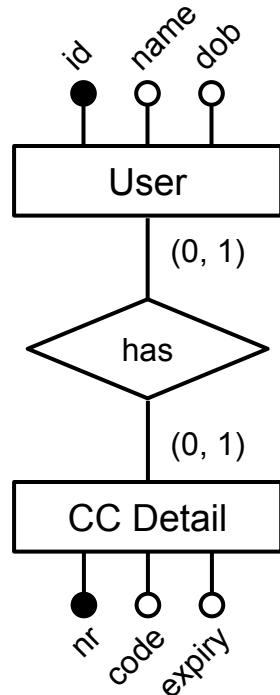
- Approach 1: Represent "has" with a separate table
 - Similar to Many-to-One but primary key can be chosen



```
CREATE TABLE Has (
    user_id      INTEGER,
    cc_nr        CHAR(16) UNIQUE,
    PRIMARY KEY (user_id),
    FOREIGN KEY (user_id) REFERENCES Users (id),
    FOREIGN KEY (cc_nr) REFERENCES CCDetails (id)
);
```

Cardinality: One-to-One (no mandatory participation)

- Approach 2: Combine "has" and "Users" or "has" and "CC Details"



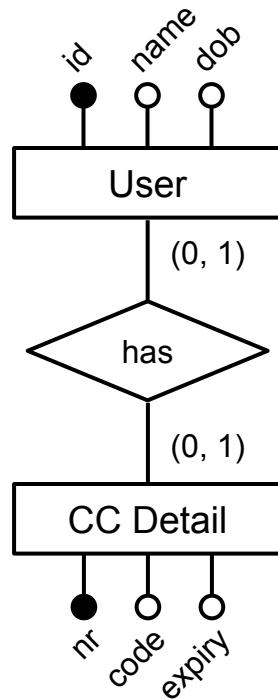
```
CREATE TABLE Users (
    id      INTEGER,
    name   VARCHAR(100),
    dob    DATE,
    cc_nr  CHAR(16) UNIQUE,
    PRIMARY KEY (id),
    FOREIGN KEY (cc_nr) REFERENCES CCDetails (nr)
);
```

```
CREATE TABLE CCdetails (
    nr      CHAR(16),
    code   CHAR(3),
    expiry DATE,
    user_id INTEGER UNIQUE,
    PRIMARY KEY (nr),
    FOREIGN KEY (user_id) REFERENCES Users (id)
);
```

Cardinality Constraints: One-to-One

Quick Quiz: What could be a downside of this approach?
(Hint: security)

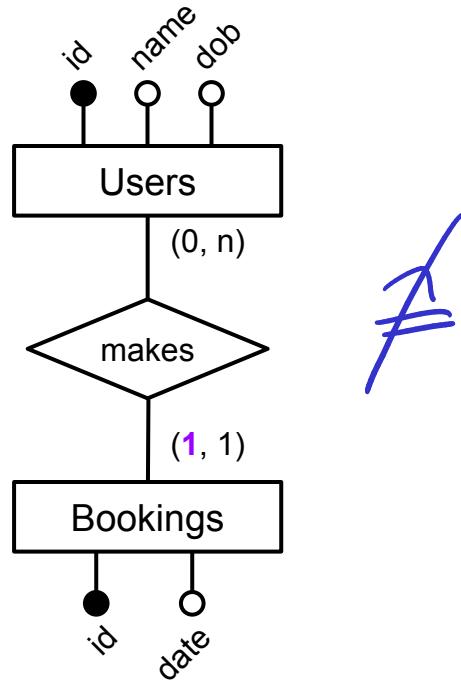
- Approach 3: Combine "has", "Users", and "CC Details"



```
CREATE TABLE Users (
    id      INTEGER,
    name   VARCHAR(100),
    dob    DATE,
    cc_nr  CHAR(16) UNIQUE,
    cc_code CHAR(3),
    cc_expiry DATE,
    PRIMARY KEY (id)
);
```

Cardinality & Participation Constraints

- Approach 1 (separate table): fails to capture mandatory participation!

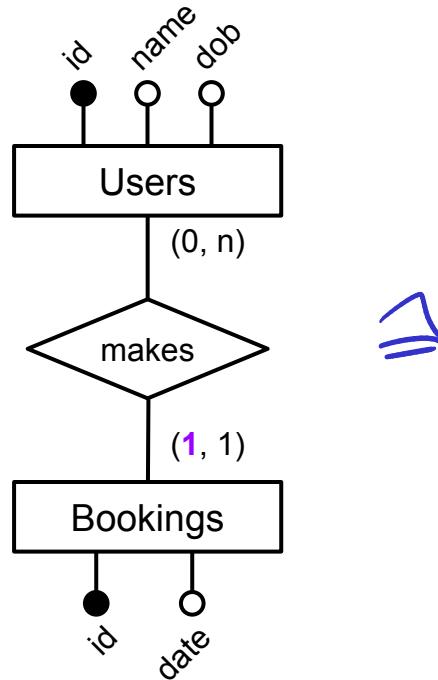


```
CREATE TABLE Makes (
    user_id      INTEGER NOT NULL,
    booking_id   INTEGER,
    PRIMARY KEY (booking_id),
    FOREIGN KEY (user_id) REFERENCES Users (id),
    FOREIGN KEY (booking_id) REFERENCES Bookings (id)
);
```

- Schema does not enforce mandatory participation of "Bookings" w.r.t. "Makes"
- e.g.: "Makes" can be empty while both "Users" and "Bookings" are non-empty

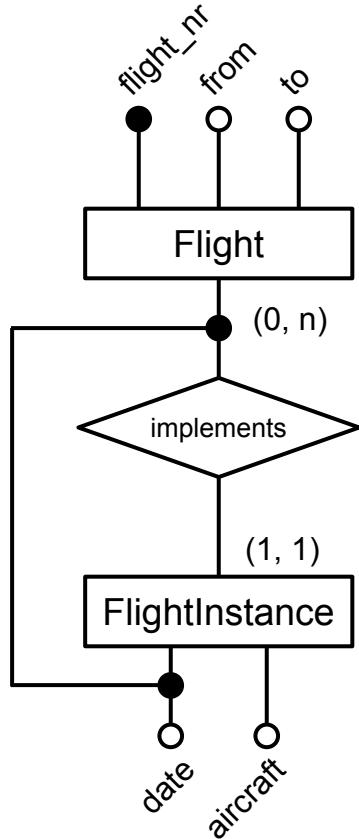
Cardinality & Participation Constraints

- **Approach 2:** Combine "makes" and "Bookings" into one table
 - Enforces total participation via NOT NULL constraint



```
CREATE TABLE Bookings (
    id      INTEGER,
    date   DATE,
    user_id INTEGER NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (user_id) REFERENCES Users (id)
);
```

Weak Entity Sets



```
CREATE TABLE Flights (
    flight_nr      VARCHAR(10),
    from           VARCHAR(10),
    to             VARCHAR(10),
    PRIMARY KEY (flight_nr)
);
```

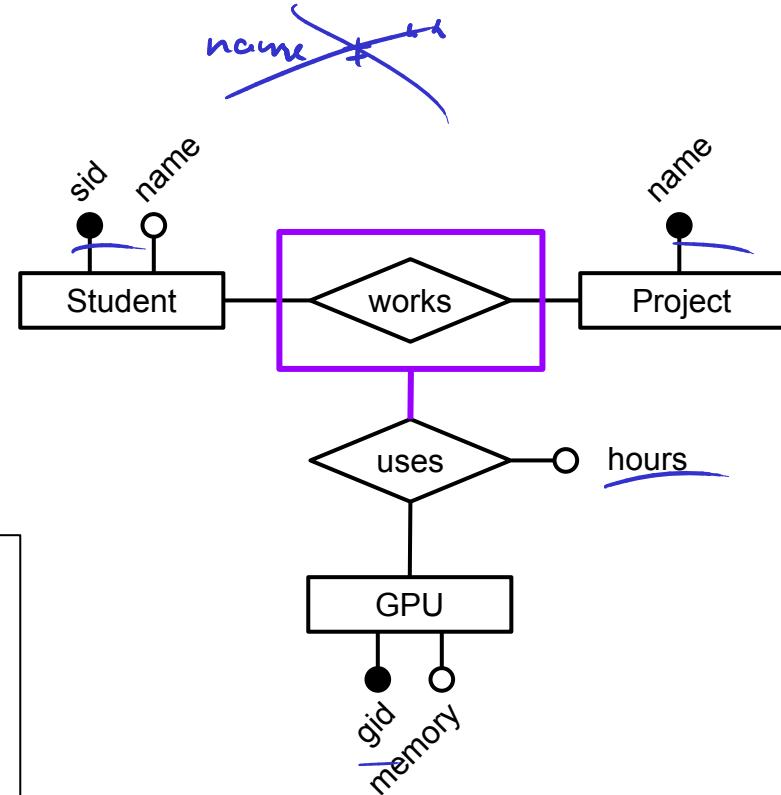
```
CREATE TABLE FlightInstances (
    flight_nr      VARCHAR(10),
    date           DATE,
    aircraft       VARCHAR(10),
    PRIMARY KEY (flight_nr, date),
    FOREIGN KEY (flight_nr) REFERENCES Flights (flight_nr)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

Aggregation — Relational Mapping

Schema definition of "uses"

- Primary key of aggregation relationship → (sid, pname)
- Primary key of associated entity set "GPUs" → gid
- Descriptive attributes of "uses" → hours

```
CREATE TABLE Uses (
    gid      INTEGER,
    sid      CHAR(20),
    pname    VARCHAR(50),
    hours   NUMERIC,
    PRIMARY KEY (gid, sid, pname),
    FOREIGN KEY (gid) REFERENCES GPUs (gid),
    FOREIGN KEY (sid, pname) REFERENCES works (sid, pname)
);
```



ER Design & Relational Mapping — Basic Guidelines

- Guidelines for ER design
 - An ER diagram should capture as many of the constraints as possible
 - An ER diagram must not impose any constraints that are not required
- Guidelines for relational mapping
 - (i.e., from ER diagram to relational database schema)
 - The relational schema should enforce as many if the constraints as possible using column and/or table constraints
 - The relational schema should not impose and constraints that are not required

Overview

- Entity Relationship Model
 - Overview + ER diagrams
 - Entity sets and attributes
 - Relationship sets
 - Cardinality & participation constraints
 - Dependency constraints: weak entity sets
 - Aggregation
- Relational Mapping
 - From ER diagram to database tables
- Summary

Summary

- Entity-Relationship (ER) model
 - Basic concepts: entity sets, relationship sets, attributes
 - Cardinality constraints and participation constraints
 - Extended concepts: (ISA hierarchies) aggregation ✓
 - Relational Mapping
 - Mapping ER diagram to database schema
 - Not all constraints of ER diagram may be captured
 - Outlook for next lecture
 - SQL for querying a database (recommendation: study RA)
- Visualized using **ER diagrams**

Quick Quiz Solutions

Quick Quiz (Slide 3)

- Solution
 - Storing the "dob" instead of "age" is arguably the preferred approach
 - The value of "age" changes each year (not really a big deal)
 - "dob" provides more detailed information compared the "age"

Quick Quiz (Slide 14)

- Solution

- Modeling "address" and "phone" as a single-values string might be OK-ish if we never use these attributes to select rows
- If we only need to get the address or all phone numbers for a given user then this solution might be good enough
- However, queries using "address" or "phone" to filter rows will become unnecessarily complicated or even impossible
- A query such as "Return all users with addresses with the ZIP code 123456" is possible since SQL supports string pattern matching and even regular expression. The performance would degrade, though.
- More intricate queries might still be formulated but the complexity of the SQL query would quickly blow up

Quick Quiz (Slide 17)

- Solution

- Using a fixed number of single-valued attributes avoids "splitting" the data across multiple tables (and avoids joining them as part of queries – costly operation)
- Two disadvantages when a fixed number of single-valued attributes
 - Unable to store then 2 phone numbers
 - Requires storing NULL values if user does not have exactly 2 phone numbers
- A separate table only stores the information needed and is more flexible, but will relies on join operations to bring the information together

Quick Quiz (Slide 18)

- Solution
 - Complex data types are generally more difficult to query
 - For example: How to check the value for an optional field in a JSON document?
Maybe be possible but often requires more complex and non-standard syntax

Quick Quiz (Slide 34)

- Solution
 - 0/1/n constraints can typically be captured using basic integrity constraints (as shown later)
 - Any more specific upper and lower bounds require more integrity checks, particularly since these constraints involve more than 1 table
 - General solution: **triggers**
 - Also not uncommon: don't use DBMS to enforce constraints

Quick Quiz (Slide 49)

- Solution
 - "Flight Instances" is weak entity set with "Flights" being the owner entity set
 - Thus, "Flight Instances" is identified by the key of "Flights" (i.e., "fnr") and its own partial key "date"

Quick Quiz (Slide 51)

- Solution
 - Approach 2 is generally the preferred approach as it leads to a smaller number of tables
 - Less tables also means that queries might need less join operations
(which are typically the more expensive operations)
 - Good rule of thumb but not a "law"

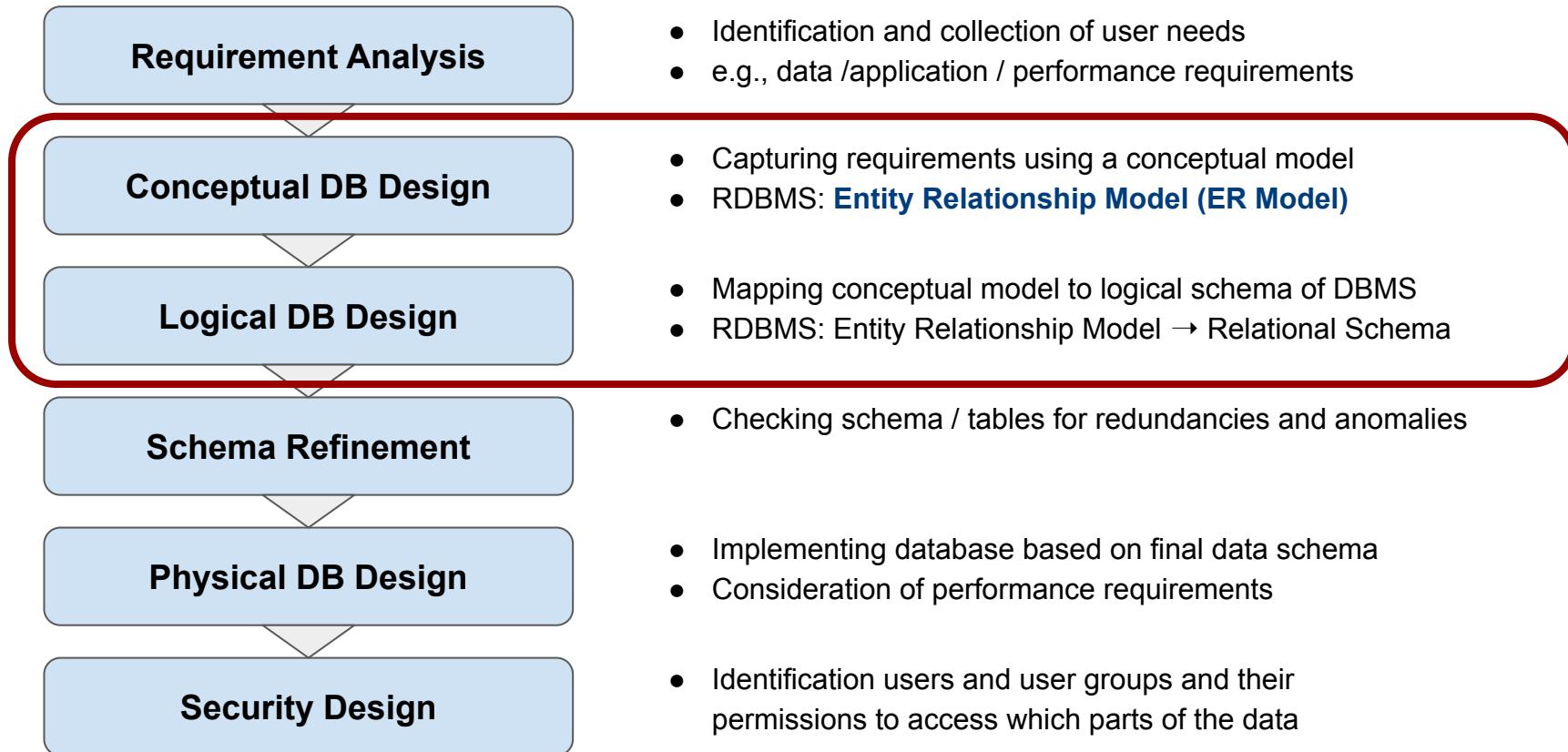
Quick Quiz (Slide 54)

- Solution
 - Access privileges can (mostly) only be set on the table level
 - Separating the basic user data and the credit card details allows assign different access privilege to different users
 - Also, separate table avoid NULL values if some users do not have a credit card

CS2102: Database Systems

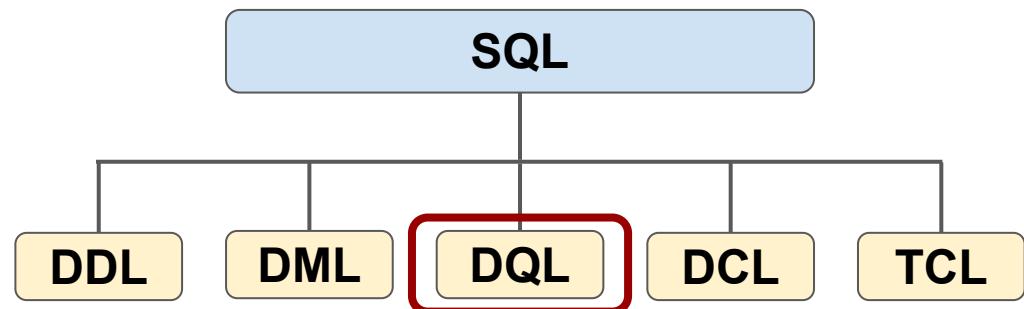
Lecture 4 — SQL (Part 2)

Quick Recap: Database Design Process



Quick Recap: Where We are Right Now

- Topics covered so far
 - Designing a database using conceptual and logical modeling
 - Creating a database using DDL (data definition language)
 - Inserting, updating and deleting data using DML (data manipulation language)
- Now: Querying a database
 - Extracting information using SQL (DQL: data query language)
 - Anything with "SELECT ..."



Overview

- **SQL – DQL**
- **SQL Queries**
 - Simple queries
 - Set operations
 - Join queries
 - Subqueries
 - Sorting & rank-based selection
- **Summary**

SQL – DQL

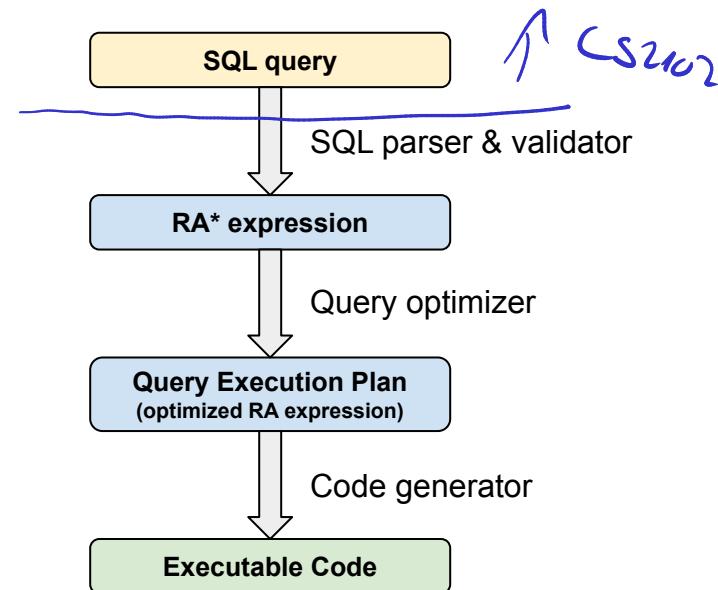
- **SQL** (more precisely: the DQL part of SQL)

- Declarative query language for RDBMS

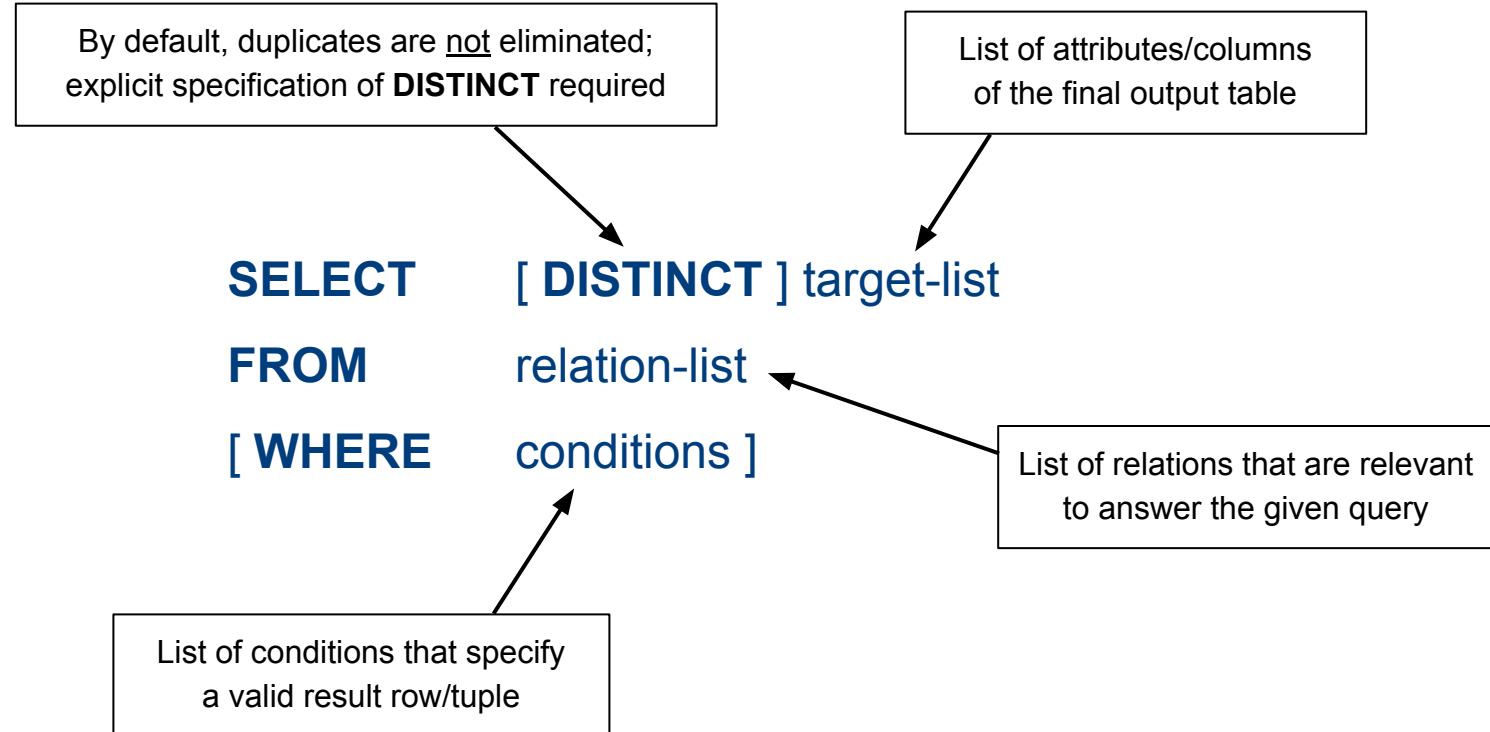
(Focus on *what* to compute, not on *how* to compute)

- Multiset / bag semantics *≠ relational model
(sets)*
- Query = SELECT statement

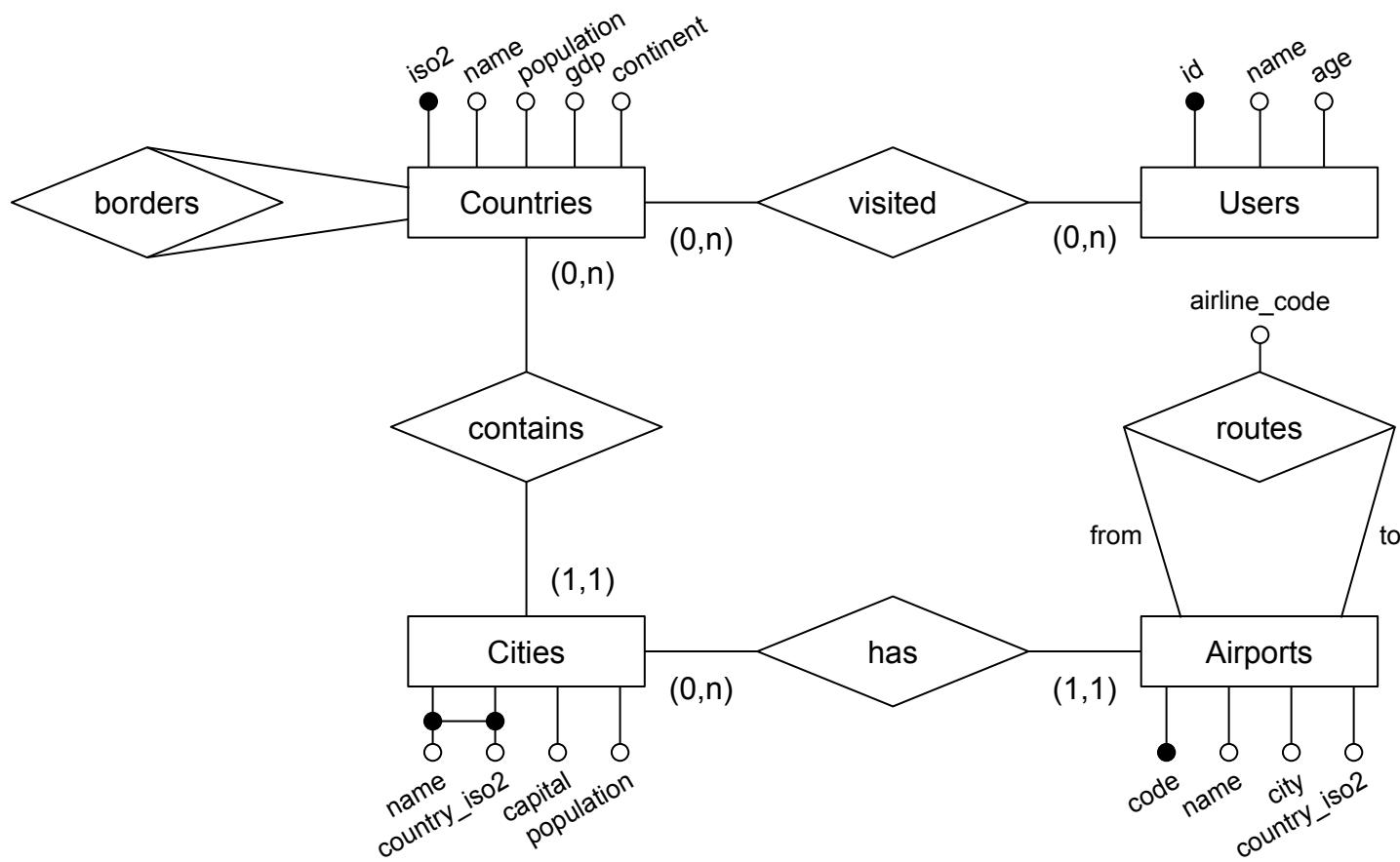
```
SELECT      [ DISTINCT ] target-list
FROM        relation-list
[ WHERE     conditions ]
```



SQL Query — Basic Form



Example Database — ER Diagram



Example Database — Data Sample

Countries (196 tuples)

iso2	name	population	gdp	continent
SG	Singapore	5781728	488000000000	Asia
AU	Australia	22992654	1190000000000	Oceania
TH	Thailand	68200824	1160000000000	Asia
DE	Germany	80722792	3980000000000	Europe
CN	China	1373541278	21100000000000	Asia
...

Borders (657 tuples)

country1_iso2	country2_iso2
SG	null
AU	null
TH	KH
TH	LA
TH	MY
...	...

Airports (3,920 tuples)

code	name	city	country_iso2
SIN	Singapore Changi Airport	Singapore	SG
XSP	Seletar Airport	Singapore	SG
SYD	Sydney Int. Airport	Sydney	AU
MEL	Melbourne Int. Airport	Melbourne	AU
FRA	Frankfurt am Main Airport	Frankfurt	DE
...

Cities (40,138 tuples)

name	country_iso2	capital	population
Singapore	SG	primary	5745000
Kuala Lumpur	MY	primary	8285000
Nanyang	CN	null	12010000
Atlanta	US	admin	5449398
Washington	US	primary	5379184
...

Routes (38,588 tuples)

from_code	to_code	airline_code
ADD	BKK	SQ
ADL	SIN	SQ
AKL	SIN	SQ
AMS	SIN	SQ
BCN	GRU	SQ
...

Users (9 tuples)

user_id	name	age
101	Sarah	25
102	Judy	35
103	Max	52
104	Marie	36
105	Sam	30
...

Visited (527 tuples)

user_id	iso2
103	AU
103	US
103	SG
103	GB
104	GB
...	...

Overview

- SQL – DQL
- **SQL Queries**
 - Simple queries
 - Set operations
 - Join queries
 - Subqueries
 - Sorting & rank-based selection
- Summary

Simple Queries (SELECT ... FROM ... WHERE)

Find the name and population of all cities with a population greater than 10 Million.

```
SELECT name, population  
FROM cities  
WHERE population > 10000000;
```

name	population
Tokyo	39105000
Jakarta	35362000
Delhi	31870000
Manila	23971000
Sao Paulo	22495000
...	...

40 tuples

Find the name and population of all countries in Asia and Europe with a population between 5 and 6 Million.

```
SELECT name, population  
FROM countries  
WHERE continent = 'Asia' OR continent = 'Europe'  
      AND (population > 5000000 AND population < 6000000);
```

>=

<=

name	population
Denmark	5873420
Finland	5536146
Ireland	5011500
Norway	5425270
Palestine	5159076
Singapore	5453600
Slovakia	5449270

Simple Queries (SELECT ... FROM ... WHERE)

- Additional language constructs
 - Wildcard '*' to include all attributes
 - 'expr **BETWEEN** <lower> **AND** <upper>' for basic value range conditions

Find all countries in Asia and Europe with a population between 5 and 6 Million.

SELECT *
FROM countries
WHERE (continent = 'Asia' **OR** continent = 'Europe')
AND (population **BETWEEN** 5000000 **AND** 6000000);

↑ ↑
inclusive

iso2	name	population	area	gdp	gini	continent
DK	Denmark	5873420	Europe
FI	Finland	5536146	Europe
IE	Ireland	5011500	Europe
NO	Norway	5425270	Europe
PS	Palestine	5159076	Asia
SG	Singapore	5453600	Asia
SK	Slovakia	5449270	Europe

SELECT Clause — Expressions

- Common use cases for SELECT clause expressions
 - Combine and process attribute values
 - Rename columns

Find the name and the GDP per capita in SGD rounded to the nearest dollar for all countries.

"||" concatenates strings
"AS" is optional
`SELECT name, 'S$ ' || ROUND((gdp / population)*1.28) AS gdp_per_capita
FROM countries;`

Convert from USD to SGD
(as of August 2025)

name	gdp_per_capita
Denmark	S\$ 90342
Germany	S\$ 66452
Kyrgyzstan	S\$ 1715
Norway	S\$ 86351
Singapore	S\$ 87872
Slovakia	S\$ 29912
Turkmenistan	S\$ 9075
United Arab Emirates	S\$ 69134
...	...

196 (all countries)

SELECT Clause — Duplicates

Quick Quiz: Why do you think does SQL not eliminate duplicates by default?

- Multiset / bad nature of SQL
 - By default, SQL does not eliminate duplicates
 - Use keyword **DISTINCT** to enforce duplicate elimination

unique

Find all country codes for which cities are available in the database.

```
SELECT country_iso2 AS code  
FROM cities;
```

40,138 tuples (all cities)

code
MX
ID
IN
IN
PH
IN
...

```
SELECT DISTINCT country_iso2 AS code  
FROM cities;
```

OR

```
SELECT DISTINCT(country_iso2) AS code  
FROM cities;
```

193 tuples

code
MX
ID
IN
PH
CN
TH
...

SELECT Clause — Duplicates with NULL Values

x	y	$x <> y$	$x \text{ IS DISTINCT FROM } y$
1	1	FALSE	FALSE
1	2	TRUE	TRUE
null	1	null	TRUE
null	null	null	FALSE

- Example: two tuples (n_1, c_1) and (n_2, c_2) are considered distinct if

" $(n_1 \text{ IS DISTINCT FROM } n_2)$ " or " $(c_1 \text{ IS DISTINCT FROM } c_2)$ "

evaluates to TRUE

SELECT name, type
FROM cities;

40,138 tuples (all cities)

name	type
Mexico City	primary
Jakarta	primary
Perth	admin
Perth	null
Perth	null
Shenzhen	minor
...	...

SELECT DISTINCT name, type
FROM cities;

39,466 tuples

name	type
Tokyo	primary
Jakarta	primary
Perth	admin
Perth	null
Shenzhen	minor
Manila	primary
...	...

WHERE Clause — Conditions for NULL Values

- Finding tuples with NULL or not-NUL as attribute value

- Correct: "attribute **IS NULL**", "attribute **IS NOT NULL**"

- False: "attribute = **NULL**", "attribute <>> **NULL**"

(CAREFUL: the conditions above do not throw an error!)

\Rightarrow evaluate to unk / null

Find all codes of countries that have no land border with another country.

```
SELECT country1_iso2 AS code  
FROM borders  
WHERE country2_iso2 IS NULL;
```

38 tuples

code
AU
SG
BH
PH
NZ
JP
...

```
SELECT country1_iso2 AS code  
FROM borders  
WHERE country2_iso2 = NULL;
```

0 tuples (but no error!)

code
-

WHERE Clause — Pattern Matching

- Basic pattern matching with (NOT) LIKE
 - "_" matches any single character
 - "%" matches any sequence of zero or more characters

Find all cities that start with "Si" and end with "re".

```
SELECT name  
FROM cities  
WHERE name LIKE 'Si%re';
```

name
Singapore
Sierre
Sierra Madre

↑
✓

- Advanced pattern matching using Regular Expression

(Out of scope here; check for full details: <https://www.postgresql.org/docs/9.3/functions-matching.html>)

Examples:

'abc' LIKE 'abc'	→	TRUE
'abc' LIKE 'a%'	→	TRUE
'abc' LIKE '_b_'	→	TRUE
'abc' LIKE '_c'	→	FALSE

Overview

- SQL – DQL
- **SQL Queries**
 - Simple queries
 - **Set operations**
 - Join queries
 - Subqueries
 - Sorting & rank-based selection
- Summary

Set Operations

columns must match
data types + order must match

- Let Q_1 and Q_2 be two SQL queries that yield union-compatible tables:

- $Q_1 \text{ UNION } Q_2 = Q_1 \cup Q_2$
- $Q_1 \text{ INTERSECT } Q_2 = Q_1 \cap Q_2$
- $Q_1 \text{ EXCEPT } Q_2 = Q_1 - Q_2$

set difference

- Attention: duplicate elimination

- UNION, INTERSECT, EXCEPT**
eliminate duplicate tuples from result
- UNION ALL, INTERSECT ALL, EXCEPT ALL**
do not eliminate duplicate tuples from result

I \sqcup T

R	S
value	value
1	2
2	2
2	3

(SELECT value FROM R)
UNION
(SELECT value FROM S);

value
2
1
3

(SELECT value FROM R)
UNION ALL
(SELECT value FROM S);

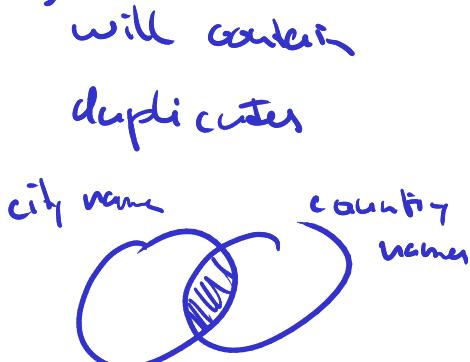
value
1
2
2
2
3

Set Operations — Example Queries

Find all names that refer to both a city and a country.

DISTINCT (but not need)

{
 SELECT name FROM cities)
INTERSECT
(SELECT name FROM countries);



name
Singapore
Mexico
Peru
Monaco
Mali
El Salvador
China
Poland
...

29 tuples

Find the codes of all the countries for which they have no city in the database.

(SELECT iso2 FROM countries)
EXCEPT
(SELECT DISTINCT(country_iso2)
FROM cities);

iso2
NA
EH
PS

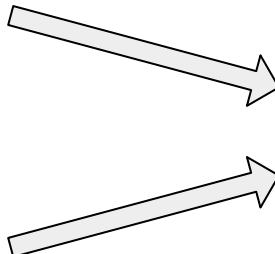


Flexibility of SQL

- Very common: Multiple ways to answer the same query
 - Note: The performance between the queries might differ significantly

Find all airports located in cities named "Singapore" or "Perth".

```
SELECT *  
FROM airports  
WHERE city = 'Singapore'  
OR city = 'Perth';
```



```
(SELECT * FROM airports  
WHERE city = 'Singapore')  
UNION  
(SELECT * FROM airports  
WHERE city = 'Perth');
```

code	name	city	country_iso2
SIN	Singapore Changi Airport	Singapore	SG
PER	Perth Int. Airport	Perth	AU
JAD	Perth Jandakot Airport	Perth	AU
PSL	Perth/Scone Airport	Perth	GB

Overview

- SQL – DQL
- **SQL Queries**
 - Simple queries
 - Set operations
 - **Join queries**
 - Subqueries
 - Sorting & rank-based selection
- Summary

Multi-Table Queries (Join Queries)

- So far: only single-table queries
 - Each SQL statement only had one table in the FROM clause
 - Some set queries contain multiple tables but in each in a different FROM clause
- Often: required information across multiple table → multi-query queries
 - Example: "*Find all countries where at least one neighboring country has a larger population.*"

Countries (196 tuples)

iso2	name	population	gdp	continent
SG	Singapore	5781728	488000000000	Asia
AU	Australia	22992654	1190000000000	Oceania
TH	Thailand	68200824	1160000000000	Asia
DE	Germany	80722792	3980000000000	Europe
CN	China	1373541278	21100000000000	Asia
...

Borders (657 tuples)

country1_iso2	country2_iso2
SG	null
AU	null
TH	KH
TH	LA
TH	MY
...	...

Multi-Table Queries (Join Queries)

- Basic SQL syntax

- Multiple table names in the same FROM clause
- Very common and always recommended: use of aliases

- Cross product / Cartesian product

- Multi-table query with WHERE clause
- Computes all possible pairs of tuples

```
SELECT c.name, n.name  
FROM cities AS c, countries AS n;
```

Returns all combinations of
city and country names

name	name
Tokyo	Albania
Tokyo	Algeria
Tokyo	Andorra
Tokyo	Angola
Tokyo	Antigua and Barbuda
Tokyo	Argentina
Tokyo	Armenia
Tokyo	Australia
Tokyo	Austria
Tokyo	Azerbaijan
Tokyo	Bangladesh
Tokyo	Belize
...	...

7,867,048 tuples

$196 \times 40,138$

Multi-Table Queries (Join Queries)

- Multi-table queries

- Most common in practice: cross product + attribute selection → join

For all cities, find their names together with the names of the countries they are located in.

equivalent queries

```
SELECT c.name, n.name  
FROM cities AS c, countries AS n  
WHERE c.country_iso2 = n.iso2;
```

join condition

```
SELECT c.name, n.name  
FROM cities c INNER JOIN countries n  
ON c.country_iso2 = n.iso2;
```

```
SELECT c.name, n.name  
FROM cities c JOIN countries n  
ON c.country_iso2 = n.iso2;
```

comparison between 2 attributes
("=", "<>", "<", "<=", ">=", ">")

"AS" is optional

name	name
Mexico City	Mexico
Jakarta	Indonesia
Delhi	India
Mumbai	India
Singapore	Singapore
Manila	Philippines
Mexico City	Mexico
Seoul	South Korea
...	...

40,138 tuples (all cities)

Multi-Table Queries (Join Queries)

*Find the names and the population of all countries with ~~directly neighboring countries~~ 1 neighbouring country
with that have a larger population. Include the neighbors and their population as well.*

```
SELECT c1.name, c1.population, c2.name, c2.population
FROM countries c1, borders b, countries c2
WHERE c1.iso2 = b.country1_iso2
AND c2.iso2 = b.country2_iso2
AND c1.population < c2.population;
```

} 2 join conditions needed here!

} or 3

name	population	name	population
Andorra	79535	Spain	47450795
Andorra	79535	France	67413000
United Arab Emirates	9282410	Saudi Arabia	34218000
Afghanistan	40218234	People's Republic of China	1412600000
Afghanistan	40218234	Iran	83183741
...

309 tuples

Multi-Table Queries (Join Queries)

- Natural Joins

- Join condition (attribute selection) implied by attribute names
- Natural joins only defined for equality comparisons ("=")
- Result does contain only one instance of matching attributes

Find all names that refer to both a city and a country.

Why?

```
SELECT DISTINCT(name)
  FROM (SELECT name FROM cities) t1
        NATURAL JOIN
    (SELECT name FROM countries) t2;
```

Quick Quiz: Why is the result of the query below empty?

```
SELECT * FROM countries NATURAL JOIN cities;
```

$t1.name = t2.name$
 $\wedge t1.pop... = t2.population$

name
Singapore
Mexico
Peru
Monaco
Mali
El Salvador
China
Poland
...

29 tuples

Multi-Table Queries (Join Queries)

- Outer Joins

- Sometimes we are interested in tuples that do not have a match in another table
- Important: this is not the same as using "<>" for the join condition

- 3 basic types:

- LEFT OUTER JOIN = INNER JOIN + all remaining tuples from the left table
- RIGHT OUTER JOIN = INNER JOIN + all remaining tuples from the right table
- FULL OUTER JOIN = INNER JOIN + all remaining tuples from both tables

Missing values
get filled with
NULL values

clueless tuples

Multi-Table Queries (Join Queries)

- Outer Joins – basic examples

R	S
x	y
1	3
2	4
3	5
4	6
5	6

```
SELECT x, y  
FROM R LEFT OUTER JOIN S  
ON x = y;  
WHERE y IS NULL
```

x	y
1	null
2	null
3	3
4	4
5	5

```
SELECT x, y  
FROM R RIGHT OUTER JOIN S  
ON x = y;
```

x	y
3	3
4	4
5	5
null	6
null	7

```
SELECT x, y  
FROM R FULL OUTER JOIN S  
ON x = y;
```

x	y
1	null
2	null
3	3
4	4
5	5
null	6
null	7

INNER JOIN tuples

Multi-Table Queries (Join Queries)

- Outer Joins – practical example

- Note: LEFT OUTER JOIN and RIGHT OUTER JOIN just mirrored version
(strictly speaking, we do not need both, but having both is consistent and flexible)

Find all the countries for which there is no city in the database.

```
SELECT n.name
FROM countries n LEFT OUTER JOIN cities c
ON n.iso2 = c.country_iso2
WHERE c.country_iso2 IS NULL;
```

optional

keep only the
dangling tuples

name
Namibia
Palestine
Western Sahara

Complex Join Queries

Find all airports in **European countries without a land border** which **cannot be reached** by plane given the existing routes in the database.

```
SELECT t1.country, t1.city, t1.airport  
FROM  
(SELECT n.name AS country, c.name AS city,  
       a.name AS airport, a.code  
     FROM borders b, countries n, cities c, airports a  
    WHERE b.country1_iso2 = n.iso2  
      AND n.iso2 = c.country_iso2  
      AND c.name = a.city  
      AND c.country_iso2 = a.country_iso2  
      AND b.country2_iso2 IS NULL  
      AND n.continent = 'Europe') t1  
  
LEFT OUTER JOIN  
  routes r  
ON t1.code = r.to_code  
WHERE r.to_code IS NULL;
```

attribute selections
for join operations

All airports in European countries
without a land border (4 tuples)

country	city	airport
Saint Lucia	Castries	George F. L. Charles Airport

Join Queries — Remarks

- In practice
 - Join condition very often along foreign key relationships
 - Most common comparison for join conditions: "=" (equality)
 - NATURAL JOIN not really needed and may yield unexpected results if you are not careful
(it is typically "safer" to specify all join conditions explicitly – even if the query gets longer)

Overview

- SQL – DQL
- **SQL Queries**
 - Simple queries
 - Set operations
 - Join queries
 - **Subqueries**
 - Sorting & rank-based selection
- Summary

Subqueries / Nested Queries

- Subqueries in FROM clause
 - Consequence of closure property
 - Must be enclosed in parentheses
 - Table alias mandatory
 - Column aliases optional

outer query

SELECT *
FROM (AS code
 {
 SELECT n.iso2, n.name
 FROM countries n, borders b
 WHERE n.iso2 = b.country1_iso2
 AND country2_iso2 IS NULL
) AS LandborderfreeCountries(code, name);

subquery inner query

iso2 name

code	name
AU	Australia
BS	Bahamas
SG	Singapore
CU	Cuba
JP	Japan
MV	Maldives
...	...

38 tuples

- Subquery expressions
 - IN subqueries
 - EXISTS subqueries
 - ANY/SOME subqueries
 - ALL subqueries

Quick Quiz: How can we rewrite the query without the column aliases but yielding the same result?

IN Subqueries

expr $\in S ?$

Quick Quiz: In the example query below, could we simply switch "countries" and "cities"?

- **(NOT) IN** subquery expressions

- Basic syntax: "*expr IN (subquery)*", "*expr NOT IN (subquery)*"
- The subquery must return exactly one column
- **IN** returns TRUE if *expr* matches with any subquery row
- **NOT IN** returns TRUE if *expr* matches with no subquery row

Find all names that refer to both a city and a country.

outer query → **SELECT name
FROM countries
WHERE name IN (SELECT name
FROM cities);** ← inner query

name $\in \subseteq$
all city names

name
Singapore
Mexico
Peru
Monaco
Mali
El Salvador
China
Poland
...

29 tuples

IN Subqueries

Find the codes of all the countries for which there is not city in the database.

```
SELECT iso2  
FROM countries  
WHERE iso2 NOT IN (SELECT country_iso2  
                 FROM cities);
```

iso2
NA
EH
PS

iso2 \notin $\{ \text{NA}, \text{EH}, \text{PS} \}$ \rightarrow iso2 codes we find in cities

- Rule of thumb (can have significant impact on query performance)
 - IN subqueries can typically be replaced with (inner) joins
 - NOT IN subqueries can typically be replaced with outer joins

IN Subquery

- Special syntax: "manual" specification of subquery result
 - Syntax: "*expression (NOT) IN (value₁, value₂, ..., value_n)*"

Find all countries in Asia and Europe with a population between 5 and 6 Million.

```
SELECT *
FROM countries
WHERE continent IN ('Asia', 'Europe')
    AND population BETWEEN 5000000 AND 6000000;
```

iso2	name	population	area	gdp	gini	continent
DK	Denmark	5873420	Europe
FI	Finland	5536146	Europe
IE	Ireland	5011500	Europe
NO	Norway	5425270	Europe
PS	Palestine	5159076	Asia
SG	Singapore	5453600	Asia
SK	Slovakia	5449270	Europe

ANY/SOME Subqueries (ANY and SOME are synonymous)

- **ANY** subquery expressions

- Basic syntax: "expr op **ANY** (subquery)"
- The subquery must return exactly one column
- Expression *expr* is compared to each subquery row using operator *op*
- **ANY** returns TRUE if comparison evaluates to TRUE for at least one subquery row

*Find all countries with a population size smaller than any city called "London"
(there are actually 3 cities called "London" on the database).*

```
SELECT name, population  
FROM countries  
WHERE population < ANY (SELECT population  
                      FROM cities  
                      WHERE name = 'London');
```

All Londons		
name	country	population
London	GB	11120000
London	CA	383822
London	US	37714

name	population
Singapore	5453600
Portugal	10344802
Sweden	10402070
Brunei	460345
Bhutan	754388
...	

113 tuples

ALL Subqueries

- **ALL** subquery expressions

- Basic syntax: "expr op ALL (subquery)"
- The subquery must return exactly one column
- Expression *expr* is compared to each subquery row using operator *op*
- **ALL** returns TRUE if comparison evaluates to TRUE for all subquery rows

*Find all countries with a population size smaller than all cities called "London"
(there are actually 3 cities called "London" on the database).*

```
SELECT name, population  
FROM countries  
WHERE population < ALL (SELECT population  
                           FROM cities  
                           WHERE name = 'London');
```

All Londons		
name	country	population
London	GB	11120000
London	CA	383822
London	US	37714

name	population
Nauru	10834
Palau	17907
San Marino	33600
Tuvalu	11900
Vatican City	453

Correlated Subqueries

Quick Quiz: Wait a minute, why is "Europe" and "Africa" missing from the result set?

- Correlated subquery
 - Subquery uses value from outer query
 - Result of subquery depends on value of outer query → potentially slow performance

For each continent, find the country with the highest GDP.

```
SELECT name, continent, gdp  
FROM countries c1  
WHERE gdp >= ALL (SELECT gdp  
                  FROM countries c2  
                  WHERE c2.continent = c1.continent);
```

name	continent	gdp
Australia	Oceania	1748000000000
Brazil	South America	1810000000000
China	Asia	1991000000000
United States	North America	2535000000000

Correlated Subqueries

- Correlated subquery
 - ALL condition must be true for all (duh!) result of the subquery
 - Problematic if subquery contains NULL value → condition never evaluates to TRUE

For each continent, find the country with the highest GDP.

```
SELECT name, continent, gdp
FROM countries c1
WHERE gdp >= ALL (SELECT gdp
                   FROM countries c2
                   WHERE c2.continent = c1.continent
                   AND c2.gdp IS NOT NULL);
```

name	continent	gdp
Australia	Oceania	1748000000000
Brazil	South America	1810000000000
China	Asia	1991000000000
Germany	Europe	4319000000000
Nigeria	Africa	498060000000
United States	North America	25350000000000

Correlated Subqueries — Scoping Rules

- Potential pitfall: naming ambiguities

- Same attribute names in inner and outer queries (here: "continent")

SELECT name, continent, gdp

FROM countries c1

WHERE gdp >= **ALL** (**SELECT** gdp

FROM countries c2

WHERE c2.continent = c1.continent

AND c2.gdp **IS NOT NULL**);

- Best approach: resolve ambiguities using table aliases (here: c1, c2)

- Otherwise: application of scoping rules

- Scoping Rules

- A table alias declared in a (sub-)query Q can only be used in Q or subqueries nested within Q
(In example above: "SELECT c1.name, c1.continent, c1.gdp ..." OK, but "SELECT c2.name, c2.continent, c2.gdp ..." fails)
 - If the same table alias is declared both in a subquery Q and in an outer query (or not at all) the declaration in Q is applied (general rule: "from inner to outer queries" in case of multiple nestings)

Scoping Rules Gone Wrong

For each continent, find the country with the highest GDP.

```
SELECT name, continent, gdp
FROM countries c1
WHERE gdp >= ALL (SELECT gdp
    FROM countries c2
    WHERE c2.continent = c1.continent
    AND c2.gdp IS NOT NULL);
```



```
SELECT name, continent, gdp
FROM countries c1
WHERE gdp >= ALL (SELECT gdp
    FROM countries c2
    WHERE c2.continent = c1.continent
    AND c2.gdp IS NOT NULL);
```



name	continent	gdp
Australia	Oceania	11900000000000
Brazil	South America	30800000000000
China	Asia	21100000000000
Egypt	Africa	11100000000000
Germany	Europe	39800000000000
United States	North America	18600000000000

name	continent	gdp
China	Asia	21100000000000

Scoping Rules Gone Wrong

Quick Quiz: Can you explain the result of the 3rd query?

Find all names that refer to both a city and a country.

**SELECT name
FROM countries
WHERE name IN (SELECT name
FROM cities);**



name
Singapore
Mexico
Peru
Monaco
Mali
El Salvador
China
Poland
...

29 tuples

**SELECT c.name
FROM countries c
WHERE name IN (SELECT c.name
FROM cities c);**



name
Singapore
Mexico
Peru
Monaco
Mali
El Salvador
China
Poland
...

29 tuples

**SELECT c1.name
FROM countries c1
WHERE name IN (SELECT c1.name
FROM cities c2);**



name
Singapore
China
Germany
Japan
Brasil
Russia
Malaysia
Vietnam
...

196 tuples (all countries)

'sin' IN
(('sin',
'sin',
...
(sin))

EXISTS Subqueries

- **(NOT) EXISTS** subquery expressions
 - Basic syntax: "EXISTS (subquery)", "NOT EXISTS (subquery)"
 - **EXISTS** returns TRUE if the subquery returns at least one tuple
 - **NOT EXISTS** returns TRUE if the subquery returns no tuple

Find all names that refer to both a city and a country.

```
SELECT n.name  
FROM countries n  
WHERE EXISTS (SELECT c.name  
               FROM cities c  
              WHERE c.name = n.name);
```

↗ ↖ ↗
c.name , c.pop

name
Singapore
Mexico
Peru
Monaco
Mali
El Salvador
China
Poland
...

29 tuples

EXISTS Subqueries

Find all the countries for which there is not city in the database.

```
SELECT n.name  
FROM countries n  
WHERE NOT EXISTS (SELECT *  
                   FROM cities c  
                   WHERE c.country_iso2 = n.iso2);
```

name
Namibia
West Sahara
Palestine

- Rule of thumb
 - (NOT) EXISTS subqueries are generally always correlated
 - Uncorrelated (NOT) EXISTS subqueries are either wrong or unnecessary

Scalar Subqueries

Quick Quiz: How do we know the subquery will return only a single value?

- **Scalar subquery — definition**
 - Subquery returns a single value (i.e., table 1 row with 1 column)
 - Can be used as a expression in queries

For all cities, find their names together with the names of the countries they are located in.

```
SELECT name AS city,  
      (SELECT name AS country  
       FROM countries n  
       WHERE n.iso2 = c.country_iso2)  
     FROM cities c;
```

city	country
Tokyo	Japan
Jakarta	Indonesia
Delhi	India
Mumbai	India
Singapore	Singapore
Manila	Philippines
Mexico City	Mexico
Seoul	South Korea
...	...

40,138 tuples

Scalar Subqueries

For Illustration Purposes Only
— Don't Try This At Home! :)

Find all cities that are located in a country with a country population smaller than the population of all cities called "London" (there are actually 3 cities called "London" on the database).

```
SELECT c.name AS city, c.country_iso2 AS country, c.population
FROM cities c
WHERE (SELECT population
      FROM countries n
      WHERE n.iso2 = c.country_iso2) < ALL (SELECT population
                                              FROM cities
                                              WHERE name = 'London');
```

population of the country for a given city which
is located in that country (single value!)

city	country	population
Funafuti	TV	6025
San Marino	SM	4040
Vatican City	VA	825
Yaren	NR	NULL
Ngerulmud	PW	271
...

15 tuples

Subqueries — Row Constructors

- So far: Requirement for IN, ANY/SOME, and ALL subqueries
 - Subquery must return exactly one attribute/column

→ Row Constructors

- Allow subqueries to return more than one attribute/column
- The number of attributes/columns in row constructor must match the one of the subquery

Attention: The semantics of comparison using row constructors can be rather unintuitive!

Subqueries — Row Constructors

name	population	gdp
France	67413000	3140000000000
Germany	83190556	4319000000000

Find all countries with a higher population or higher gdp than France or Germany

```
SELECT name, population, gdp
FROM countries
WHERE ROW(population, gdp) > ANY (SELECT population, gdp
                                     FROM countries
                                     WHERE name IN ('Germany', 'France'));
```

name	population	gdp
China	1412600000	19910000000000
Turkey	84680273	692000000000
Nigeria	211400708	498060000000
Vietnam	96208984	340602000000
United States	331893745	25350000000000
...

19 tuples

Note: For the <, <=, > and >= cases, the row elements are compared left-to-right, stopping as soon as an unequal or null pair of elements is found.
For more details: <https://www.postgresql.org/docs/current/functions-comparisons.html#ROW-WISE-COMPARISON>

Subqueries — Remarks

- Queries can contain multiple nested subqueries

Find all the airports in Denmark.

```
SELECT name, city
FROM airports
WHERE city IN (SELECT name
                FROM cities
                WHERE country_iso2 IN (SELECT iso2
                                        FROM countries
                                        WHERE name = 'Denmark'))
      );
```

```
SELECT a.name, a.city
FROM airports a, cities c, countries n
WHERE a.city = c.name
AND c.country_iso2 = n.iso2
AND n.name = 'Denmark';
```

Alternative query
using only joins

name	city
Aarhus Airport	Aarhus
Copenhagen Kastrup Airport	Copenhagen
Esbjerg Airport	Esbjerg
Odense Airport	Odense
Copenhagen Roskilde Airport	Copenhagen
Aalborg Airport	Aalborg

Subqueries — Remarks

- Not all constructs are absolutely required
 - "*expr IN* (subquery)" is equivalent to "*expr = ANY* (subquery)"
 - "*expr1 op ANY (SELECT expr2 FROM ... WHERE ...)*" is equivalent to "**EXISTS (SELECT * FROM ... WHERE ... AND *expr1 op expr2*)**"
 - ...

Overview

- SQL – DQL
- **SQL Queries**
 - Simple queries
 - Set operations
 - Join queries
 - Subqueries
 - **Sorting & rank-based selection**
- Summary

Sorting — ORDER BY

- Sorting tables

- By default, order of tuples in a table is unpredictable!
- Sorting of tuples with ORDER BY in ascending order (**ASC**) or descending order (**DESC**)
- Sorting w.r.t. multiple attributes and different orders supported

Find the GDP per capita for all countries sorted from highest to lowest.

```
SELECT name, (gdp/population) AS gdp_per_capita  
FROM countries  
WHERE gdp is NOT NULL  
ORDER BY gdp_per_capita DESC;
```

ASC (default)

name	gdp_per_capita
Monaco	193838
Liechtenstein	107612
Luxembourg	107612
Ireland	102963
Switzerland	87396
Qatar	84513
Brunei	77235
...	...

194 tuples

Sorting — ORDER BY

Find all cities sorted by country (ascending from A to Z) and for each country with respect to the cities' population size in descending order.

```
SELECT n.name AS country, c.name AS city, c.population  
FROM cities c, countries n  
WHERE c.country_iso2 = n.iso2  
AND c.population IS NOT NULL  
ORDER BY n.name ASC, c.population DESC;
```

The 2nd sorting criteria only affects result if 1st sorting criteria does not yield an unambiguous order already!

country	city	population
Afghanistan	Kabul	4273156
Afghanistan	Kandahar	614254
Afghanistan	Herat	556205
...
Albania	Tirana	418495
Albania	Vlore	130827
Albania	Kamez	126777
...
Zimbabwe	Chivhu	10263
Zimbabwe	Mazoe	9966
Zimbabwe	Plumtree	2148

39,493 tuples

LIMIT & OFFSET — Selection Based on Ranking

- Returning only a portion of the result table
 - **LIMIT** k : return the "first" k tuples of the result table
 - **OFFSET** i : specify the position of the "first" tuple to be considered
 - Typically only meaningful in combination with **ORDER BY**

Find the top-5 countries regarding their GDP per capita for all countries.

```
SELECT name, (gdp/population) AS gdp_per_capita
FROM countries
WHERE gdp IS NOT NULL
ORDER BY gdp_per_capita DESC
LIMIT 5;
```

name	gdp_per_capita
Monaco	193838
Liechtenstein	176676
Luxembourg	107612
Ireland	102963
Switzerland	87396

LIMIT & OFFSET — Selection Based on Ranking

Find the "second" top-5 countries regarding their GDP per capita for all countries.

```
SELECT name, (gdp/population) AS gdp_per_capita  
FROM countries  
WHERE gdp IS NOT NULL  
ORDER BY gdp_per_capita DESC  
OFFSET 5  
LIMIT 5;
```

name	gdp_per_capita
Qatar	84513
Brunei	77235
United States	76379
Denmark	70580
Singapore	68650

- Typical use case: Pagination on websites

Previous [1](#) [2](#) [3](#) [4](#) [5](#) Next

« [1](#) [2](#) [3](#) [4](#) [5](#) 7 »

Items per Page: [3](#) ▾

[◀](#) [1](#) [2](#) [3](#) [4](#) [5](#) ... [11](#) [▶](#)

Summary

Find all names that refer to both a city and a country.

(**SELECT** name **FROM** cities)

INTERSECT

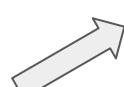
(**SELECT** name **FROM** countries);



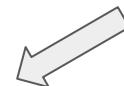
name
Singapore
Mexico
Peru
Monaco
Mali
El Salvador
China
Poland
...

29 tuples

SELECT DISTINCT(name)
FROM (**SELECT** name **FROM** cities) t1
NATURAL JOIN
(**SELECT** name **FROM** countries) t2;



SELECT n.name
FROM countries n
WHERE EXISTS (**SELECT** c.name
FROM cities c
WHERE c.name = n.name);



SELECT name
FROM countries
WHERE name **IN** (**SELECT** name
FROM cities);

Summary

- **Querying relational databases with SQL (DQL)**
 - Declarative query language
 - Built on top of Relational Algebra (Lecture 6)
- **This lecture**
 - Basic queries (SELECT ... FROM ... WHERE)
 - Set queries and join queries
 - Subqueries
 - Sorting, rank-based selection
- **Next lecture**
 - Aggregation, grouping, conditional expressions, extended concepts

Quick Quiz Solutions

Quick Quiz (Slide 13)

- Solution
 - Duplicate elimination is quite expensive operation particularly over very large data
 - Only eliminate duplicates if really needed

Quick Quiz (Slide 26)

- Solution
 - Tables "countries" and "cities" share two attributes: "name" and "population"
 - There's no country-city pair where both name and population match
(not even for city states like Singapore)

Quick Quiz (Slide 33)

- Solution

- We can define aliases in any of the 2 SELECT clauses
- For example, for the inner SELECT clause: [...] **SELECT n.iso2 AS code, n.name AS name [...]**

```
SELECT *
FROM (
    SELECT n.iso2 AS code, n.name AS name
        not really needed
    FROM countries n, borders b
    WHERE n.iso2 = b.country1_iso2
    AND country2_iso2 IS NULL
) AS LandborderfreeCountries;
```

Quick Quiz (Slide 34)

- Solution
 - Yes we can, but we then need a "SELECT DISTINCT name ..." for the outer query
 - Reason: there are a few duplicate city names that are also a country

Quick Quiz (Slide 39)

- Solution
 - There are some European and African countries with unknown (NULL) values for the GDP
 - The ALL subquery requires the condition to be TRUE for all results of the subquery
 - Any comparison with NULL does not evaluate to TRUE

Quick Quiz (Slide 43)

- Solution

- The inner query contains 196 tuples (all countries)
- All values in the result of the inner query are the same value: the current country name from the outer query
- Visualized using "Singapore" as example

WHERE 'Singapore' IN ('Singapore', 'Singapore', 'Singapore', ...)

40k+ times (#cities)

```
SELECT c1.name  
FROM countries c1  
WHERE name IN (SELECT c1.name  
                FROM cities c2);
```

WHERE 'Germany' IN ('Germany', 'Germany', ...)

40k+

Quick Quiz (Slide 46)

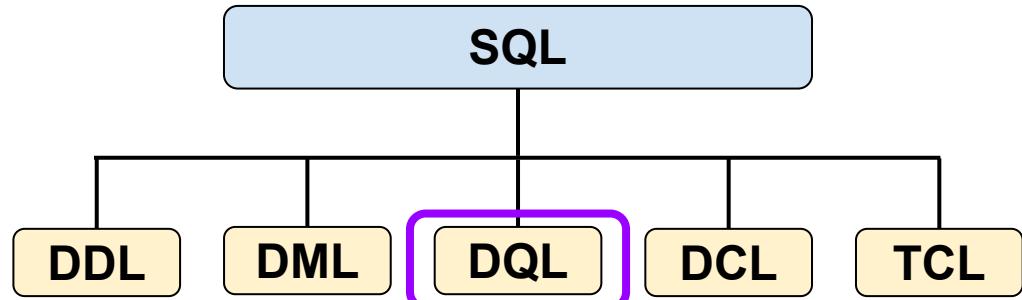
- Solution
 - The DBMS doesn't know ahead of time
 - An error raised when executing the query and the subquery returns more than 1 row or column

CS2102: Database Systems

Lecture 5 — SQL (Part 3)

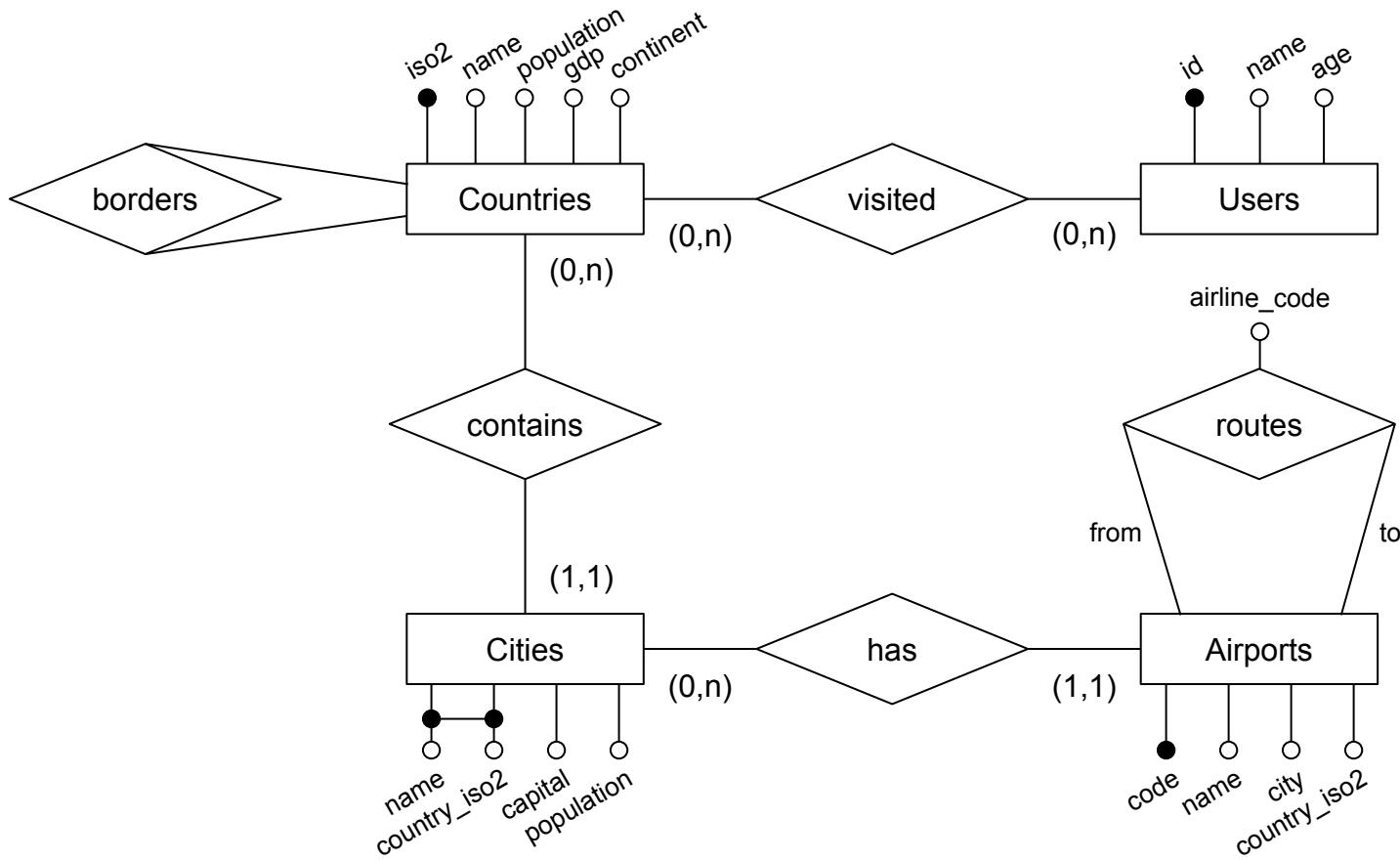
Quick Recap: Where We are Right Now

- **Querying a database**
 - Extracting information using SQL (DQL: data query language)
 - Anything with "**SELECT ...**"



- **Covered constructs**
 - Basic queries: **SELECT [DISTINCT] ... FROM [WHERE]**
 - Multirelational queries / join queries: **(INNER) JOIN**, **NATURAL JOIN**, **OUTER JOIN**, etc
 - Subquery expressions: **(NOT) IN**, **(NOT) EXISTS**, **ANY/SOME**, **ALL**
 - Sorting & rank-based selection: **ORDER BY**, **LIMIT**, **OFFSET**

Example Database — ER Diagram



Example Database — Data Sample

Countries (225 tuples)

iso2	name	population	gdp	continent
SG	Singapore	5781728	488000000000	Asia
AU	Australia	22992654	1190000000000	Oceania
TH	Thailand	68200824	1160000000000	Asia
DE	Germany	80722792	3980000000000	Europe
CN	China	1373541278	21100000000000	Asia
...

Borders (699 tuples)

country1_iso2	country2_iso2
SG	null
AU	null
TH	KH
TH	LA
TH	MY
...	...

Airports (3,372 tuples)

code	name	city	country_iso2
SIN	Singapore Changi Airport	Singapore	SG
XSP	Seletar Airport	Singapore	SG
SYD	Sydney Int. Airport	Sydney	AU
MEL	Melbourne Int. Airport	Melbourne	AU
FRA	Frankfurt am Main Airport	Frankfurt	DE
...

Cities (24,567 tuples)

name	country_iso2	capital	population
Singapore	SG	primary	5745000
Kuala Lumpur	MY	primary	8285000
Nanyang	CN	null	12010000
Atlanta	US	admin	5449398
Washington	US	primary	5379184
...

Routes (47,076 tuples)

from_code	to_code	airline_code
ADD	BKK	SQ
ADL	SIN	SQ
AKL	SIN	SQ
AMS	SIN	SQ
BCN	GRU	SQ
...

Users (9 tuples)

user_id	name	age
101	Sarah	25
102	Judy	35
103	Max	52
104	Marie	36
105	Sam	30
...

Visited (585 tuples)

user_id	iso2
103	AU
103	US
103	SG
103	GB
104	GB
...	...

Overview

- **Common SQL constructs**
 - Aggregation
 - Grouping
 - Conditional Expressions
- Structuring Queries
 - Common Table Expressions
 - Views
- Extended concepts
 - Universal Quantification
 - Recursive Queries
- Summary

Aggregation

- Aggregate functions
 - Compute a single value from a set of tuples
 - Examples: **MIN()**, **MAX()**, **AVG()**, **COUNT()**, **SUM()**

Find find the lowest and highest population sizes among all countries, as well as the global population size (= sum over all countries).

```
SELECT MIN(population) AS lowest,  
       MAX(population) AS highest,  
       SUM(population) AS global  
FROM countries;
```

lowest	highest	global
453	1412600000	7712195627

Aggregation — Interpretation of NULL values

- Let R be a non-empty relation with attribute A

...	A	...
...	3	...
...	null	...
...	42	...
...	0	...
...	3	...

Query	Interpretation	Result
SELECT MIN(A) FROM R;	Minimum non-null value in A	0
SELECT MAX(A) FROM R;	Maximum non-null value in A	42
SELECT AVG(A) FROM R;	Average of non-null values in A	12
SELECT SUM(A) FROM R;	Sum of non-null values in A	48
SELECT COUNT(A) FROM R;	Count of non-null values in A	4
SELECT COUNT(*) FROM R;	Count of rows in R	5
SELECT AVG(DISTINCT A) FROM R;	Average of distinct non-null values in A	15
SELECT SUM(DISTINCT A) FROM R;	Sum of distinct non-null values in A	45
SELECT COUNT(DISTINCT A) FROM R;	Count of distinct non-null values in A	3

Aggregation — Interpretation of NULL values

- Let R, S be two relations with an attribute A
 - Let R be an empty relation
 - Let S be a non-empty relation with n tuples but only null values for A

Table R			Table S		
...	A	A	...
...	null	null	...
...	null	null	...
...	null	null	...
...

Query	Result
SELECT MIN(A) FROM R;	null
SELECT MAX(A) FROM R;	null
SELECT AVG(A) FROM R;	null
SELECT SUM(A) FROM R;	null
SELECT COUNT(A) FROM R;	0
SELECT COUNT(*) FROM R;	0

Query	Result
SELECT MIN(A) FROM S;	null
SELECT MAX(A) FROM S;	null
SELECT AVG(A) FROM S;	null
SELECT SUM(A) FROM S;	null
SELECT COUNT(A) FROM S;	0
SELECT COUNT(*) FROM S;	<i>n</i>

Aggregation — More Examples

*Find the first and last city in the United States
with respect to their lexicographic sorting.*

```
SELECT MIN(name) AS lexi_first, MAX(name) AS lexi_last  
FROM cities  
WHERE country_iso2 = 'US';
```

lexi_first	lexi_last
Abbeville	Zuni Pueblo

*Find the number countries with at least 10% of the population
compared to the country with the largest population size.*

```
SELECT COUNT(*) AS num_big_countries  
FROM countries  
WHERE population >= 0.1 * (SELECT MAX(population)  
                           FROM countries);
```

(population)

num_big_countries
9

Scalar subquery!

Aggregate Functions — Signatures

- Data type of attribute/column of a table affects:
 - Applicability of aggregate functions
 - Return data type of aggregate functions
- Examples
 - **MIN()**, **MAX()** defined for all data types; return data type same as input data type
 - **SUM()** defined for all numeric data types; **SUM(INTEGER)**→BIGINT, **SUM(REAL)**→REAL, ...
 - **COUNT()** defined for all data types; **COUNT(...)**→BIGINT
 - ...

Overview

- **Common SQL constructs**
 - Aggregation
 - **Grouping**
 - Conditional Expressions
- Structuring Queries
 - Common Table Expressions
 - Views
- Extended concepts
 - Universal Quantification
 - Recursive Queries
- Summary

Grouping — GROUP BY Clause

- Aggregation so far
 - Application of aggregate functions over all tuples of a relation
 - Result relation has only one tuple

→ Grouping using GROUP BY

- Logical partition of relation into groups based on values for specified attributes
- In principle, always applied together with aggregation
(GROUP BY without aggregation valid but typically not meaningful)
- Application of aggregation functions over each group
- One result tuple for each group

GROUP BY — Example

For each continent, find the lowest and highest population sizes among all countries, as well as the overall population size for that continent.

Logical partition of "Countries" w.r.t. "continent"

iso2	name	population	area	gdp	gini	continent
DZ	Algeria	44700000	Africa
AO	Angola	33086278	Africa
...
AF	Afghanistan	40218234	Asia
BH	Bahrain	1569446	Asia
...
AR	Argentina	45605826	South America
BO	Bolivia	11428245	South America
...
BS	Bahamas	400516	North America
CA	Canada	38526760	North America
...
...	Europe



SELECT continent,
MIN(population) **AS** lowest,
MAX(population) **AS** highest,
SUM(population) **AS** overall
FROM countries
GROUP BY continent;

continent	lowest	highest	overall
Africa	99331	211400708	1354025807
Asia	579330	1412600000	4554731303
South America	575990	212688125	430763036
North America	52441	331893745	585036622
Europe	453	145478097	745055194
Oceania	10834	25997100	42583665

GROUP BY — Example

For each route, find the number of airlines that serve that route.

Logical partition of "Routes" w.r.t. "from_code" and "to_code"

from_code	to_code	airline_code
SIN	FRA	SQ
SIN	FRA	LH
SIN	FRA	US
PEK	SIN	CA
PEK	SIN	SQ
MNL	SIN	3K
MNL	SIN	5J
MNL	SIN	PR
MNL	SIN	SQ
MNL	SIN	TR
SIN	ADL	ET
SIN	ADL	SQ
SIN	ADL	VA
SIN	HEL	AY
...

```
SELECT from_code, to_code,  
       COUNT(*) AS num_airlines  
FROM routes  
GROUP BY from_code, to_code;
```

from_code	to_code	num_airlines
SIN	FRA	3
PEK	SIN	2
MNL	SIN	5
SIN	ADL	3
SIN	HEL	1
MNL	KLO	6
ATL	JFK	10
KUL	BKK	9
...

20,326 tuples

GROUP BY Clause — Defining Groups

- Given "**GROUP BY** a_1, a_2, \dots, a_n ", 2 tuples t and t' belong to the same group if
 - " $(t.a_1 \text{ IS NOT DISTINCT FROM } t'.a_1)$ " and
 - " $(t.a_2 \text{ IS NOT DISTINCT FROM } t'.a_2)$ " and
 - ... and
 - " $(t.a_n \text{ IS NOT DISTINCT FROM } t'.a_n)$ "

evaluates to "true"

- Example:
 - Table R with three attributes A, B, C

A	B	C
null	4	19
6	1	null
20	2	10
1	1	2
1	18	2
null	21	19
6	20	null

SELECT
FROM R
GROUP BY $A, C;$

A	B	C
null	4	19
null	21	19
6	1	null
6	20	null
20	2	10
1	1	2
1	18	2

GROUP BY Clause — Restrictions to SELECT Clause

- If column A_i of table R appears in the **SELECT** clause, one of the following conditions must hold:
 - A_i appears in the **GROUP BY** clause
 - A_i appears as input of an aggregation function in the **SELECT** clause
 - The primary key ~~or a candidate key~~ of R appears in the **GROUP BY** clause



Valid in standard SQL but not supported by PostgreSQL.
In this module, we follow PostgreSQL's tighter restriction

Example of an **invalid** query:

SELECT continent, gdp, **SUM**(population)
FROM countries
GROUP BY continent;

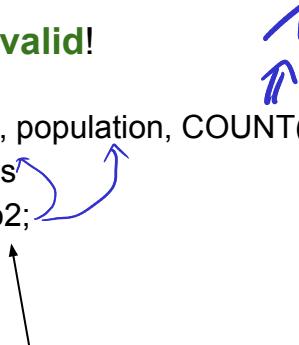
GROUP BY — Grouping over Primary Key

- Assume table "Countries" was created as shown on the right

```
CREATE TABLE Countries (
    iso2      CHAR(2) PRIMARY KEY,
    name      VARCHAR(255) UNIQUE,
    population INTEGER,
    gdp       BIGINT,
    continent VARCHAR(255)
);
```

This query is **valid!**

```
SELECT name, population, COUNT(*)
FROM countries
GROUP BY iso2;
```



Quick Quiz: What is the "problem" with this query?

This query is **valid** SQL standard but **invalid** PostgreSQL!

```
SELECT name, population, COUNT(*)
FROM countries
GROUP BY name;
```



GROUP BY — Grouping over Primary Key

- Assume table "Countries" was created as shown on the right
 - No key constraints on "Cities"

```
CREATE TABLE Countries (
    iso2      CHAR(2) PRIMARY KEY,
    name      VARCHAR(255) UNIQUE,
    population  INTEGER,
    gdp       BIGINT,
    continent  VARCHAR(255)
);
```

This query is **valid!**

```
SELECT n.name, n.population, COUNT(*)
FROM cities c, countries n
WHERE c.country_iso2 = n.iso2
GROUP BY n.iso2;
```

This query is **invalid!**

```
SELECT n.name, c.name, COUNT(*)
FROM cities c, countries n
WHERE c.country_iso2 = n.iso2
GROUP BY n.iso2;
```

This query is **valid!**

```
SELECT n.name, n.population, COUNT(*)  
FROM cities c, countries n  
WHERE c.country_iso2 = n.iso2  
GROUP BY n.iso2;
```

This query is **invalid!**

```
SELECT n.name, c.name, COUNT(*)  
FROM cities c, countries n  
WHERE c.country_iso2 = n.iso2  
GROUP BY n.iso2;
```

n.iso2	n.name	n.population	...	c.name	c.country_iso2	c.population	...
BS	Bahamas	400516	...	Nassau	BS	274400	...
BS	Bahamas	400516	...	Freeport City	BS	45945	...
BS	Bahamas	400516	...	Marsh Harbour	BS	6283	...
SG	Singapore	5453600	...	Singapore	SG	5271000	...
DJ	Djibouti	921804	...	Djibouti	DJ	562000	...
DJ	Djibouti	921804	...	Arta	DJ	null	...
DJ	Djibouti	921804	...	Ali Sabieh	DJ	37939	...
DJ	Djibouti	921804	...	Dikhil	DJ	35000	...
DJ	Djibouti	921804	...	Obock	DJ	21200	...
DJ	Djibouti	921804	...	Tadjourah	DJ	14820	...
AU	Australia	25997100	...	Sydney	AU	4840600	...
AU	Australia	25997100	...	Melbourne	AU	4529500	...

HAVING Clause — Conditions over Groups

WHERE → tuples
HAVING → groups

- **HAVING** conditions

- Conditions check for each group defined by **GROUP BY** clause
- **HAVING** clause cannot be used without a **GROUP BY** clause
- Conditions typically involve aggregate functions

Find all routes that are served by more than 12 airlines.

```
SELECT from_code, to_code,  
       COUNT(*) AS num_airlines  
FROM routes  
GROUP BY from_code, to_code  
HAVING COUNT(*) > 12;
```

from_code	to_code	num_airlines
ORD	ATL	20
ATL	ORD	19
ORD	MSY	13
HKT	BKK	13

HAVING Clause — Conditions over Groups

Find all countries that have at least one city with a population size larger than the average population size of all European countries

```
SELECT n.name, n.continent  
FROM cities c, countries n  
WHERE c.country_iso2 = n.iso2  
GROUP BY n.name, n.continent  
HAVING MAX(c.population) > (SELECT AVG(population)  
    - FROM countries  
    WHERE continent = 'Europe');
```



Scalar
Sub query

name	continent
Bangladesh	Asia
Japan	Asia
Mexico	North America
India	Asia
Egypt	Africa
Philippines	Asia
Russia	Europe
Thailand	Asia
China	Asia
Brazil	South America
Argentina	South America
South Korea	Asia
Indonesia	Asia
United States	North America

GROUP BY Clause — Restrictions to HAVING Clause

- If column A_i of table R appears in the **HAVING** clause, one of the following conditions must hold:
 - A_i appears in the **GROUP BY** clause
 - A_i appears as input of an aggregation function in the **HAVING** clause
 - The primary key ~~or a candidate key~~ of R appears in the **GROUP BY** clause

Valid Queries

```
SELECT continent, COUNT(*)  
FROM countries  
GROUP BY continent  
HAVING AVG(population) > 25000000;
```

Invalid Query

```
SELECT continent, COUNT(*)  
FROM countries  
GROUP BY continent  
HAVING name = 'China';
```

```
SELECT continent, COUNT(*)  
FROM countries  
GROUP BY continent  
HAVING continent = 'Asia';
```

Asia , 1

```
SELECT continent, COUNT(*)  
FROM countries  
GROUP BY iso2  
HAVING name = 'China';
```

Quick Quiz: What is the result of this query?

Conceptual Evaluation of Queries

Unit 3

5	7
4	
3	
3	
3	
2	

FROM

Compute cross-product of all tables in FROM clause

WHERE

Filter tuples that evaluate to true on the WHERE condition(s)

GROUP BY

Partition table into groups w.r.t. to the grouping attribute(s)

HAVING

Filter groups that evaluate to true on the HAVING condition(s)

SELECT

Remove all attributes not specified in SELECT clause

ORDER BY

Sort tables based on specified attribute(s)

LIMIT/OFFSET

Filter tuples based on their order in the table

Final output

Overview

- **Common SQL constructs**
 - Aggregation
 - Grouping
 - **Conditional Expressions**
- Structuring Queries
 - Common Table Expressions
 - Views
- Extended concepts
 - Universal Quantification
 - Recursive Queries
- Summary

CASE — Conditional Expressions

- CASE expression
 - Generic conditional expression
 - Similar to case or if/else statements in programming languages
- Two basic ways for formulating CASE expressions

CASE

WHEN *condition₁*, **THEN** *result₁*,

WHEN *condition₂*, **THEN** *result₂*

...

WHEN *condition_n*, **THEN** *result_n*

ELSE *result₀*

END

CASE expression

WHEN *value₁*, **THEN** *result₁*,

WHEN *value₂*, **THEN** *result₂*

...

WHEN *value_n*, **THEN** *result_n*

ELSE *result₀*

END

CASE — Conditional Expressions

Find the number of all cities regarding the classification (defined by a cities population size).

```
SELECT class, COUNT(*) AS city_count
FROM
  (SELECT name, CASE
    WHEN population > 10000000 THEN 'Super City'
    WHEN population > 5000000 THEN 'Mega City'
    WHEN population > 1000000 THEN 'Large City'
    WHEN population > 500000 THEN 'Medium City'
    ELSE 'Small City' END AS class
  FROM cities) t
GROUP BY class;
```

Subquery

City Size	Urban Population (Million)
Super city	>10
Megacity	5–10
Large city	1–5
Medium city	0.5–1
Small city	<0.5

class	city_count
Medium City	576
Large City	546
Small City	38872
Mega City	104
Super City	40

CASE — Conditional Expressions

Find all countries and return the continent in Tamil.

```
SELECT name, CASE continent
    WHEN 'Africa' THEN 'ஆப்பிரிக்கா'
    WHEN 'Asia' THEN 'ஆசியா'
    WHEN 'Europe' THEN 'ஐரோப்பா'
    WHEN 'North America' THEN 'வட அமெரிக்கா'
    WHEN 'South America' THEN 'தென் அமெரிக்கா'
    WHEN 'Oceania' THEN 'ஓசியானியா'
    ELSE NULL END AS continent
FROM countries;
```

name	continent
Afghanistan	ஆசியா
Albania	ஐரோப்பா
Algeria	ஆப்பிரிக்கா
Andorra	ஐரோப்பா
Angola	ஆப்பிரிக்கா
Antigua and Barbuda	வட அமெரிக்கா
Argentina	தென் அமெரிக்கா
...	...

COALESCE — Conditional Expressions for NULL Values

- COALESCE(value1, value2, value3, ...)
 - Returns the first non-NULL value in the list of input arguments
 - Returns NULL if all values in the list of input arguments are NULL
 - Example: **SELECT COALESCE(null, null, 1, null, 2)** →

val
1

*Find the number of cities for each city type;
consider cities with NULL for column "capital" as "other".*

```
SELECT type, COUNT(*) AS city_count
FROM
    (SELECT COALESCE(type, 'other') AS type
     FROM cities) t
GROUP BY type;
```

*('primary', 'other') → 'primary'
(NULL, 'other') → 'other'*

type	city_count
primary	206
other	30573
admin	5852
minor	3507

NULLIF — Conditional Expressions for NULL Values

- **NULLIF(*value₁*, *value₂*)**

- Returns NULL if *value₁*=*value₂*; otherwise returns *value₁*,

- Examples:

SELECT NULLIF(1, 1) AS val; →

val
null

SELECT NULLIF(1, 2) AS val; →

val
1

- Common use case: convert “special” values (zero, empty string) to NULL values

Find the minimum and average Gini Coefficients across all countries (unknown values are represented by 0)

(0, 0) → null

```
SELECT MIN(gini) AS min_gini,  
       AVG(gini) AS avg_gini  
FROM countries;
```

min_gini	avg_gini
0.0	33.08

```
SELECT MIN(NULLIF(gini, 0)) AS min_gini,  
       AVG(NULLIF(gini, 0)) AS avg_gini  
FROM countries;
```

min_gini	avg_gini
22.8	37.92

Overview

- Common SQL constructs
 - Aggregation
 - Grouping
 - Conditional Expressions
- Structuring Queries
 - Common Table Expressions
 - Views
- Extended concepts
 - Universal Quantification
 - Recursive Queries
- Summary

Common Table Expressions (CTEs)

• Motivation

- SQL can quickly become complex and unreadable
- CTEs allow to structure SQL queries to improve readability

→ Common Table Expression CTE

- Temporary named query
- One or more CTEs can be used within an SQL statement

country	city	airport
Saint Lucia	Castries	George F. L. Charles Airport

Example from last lecture:

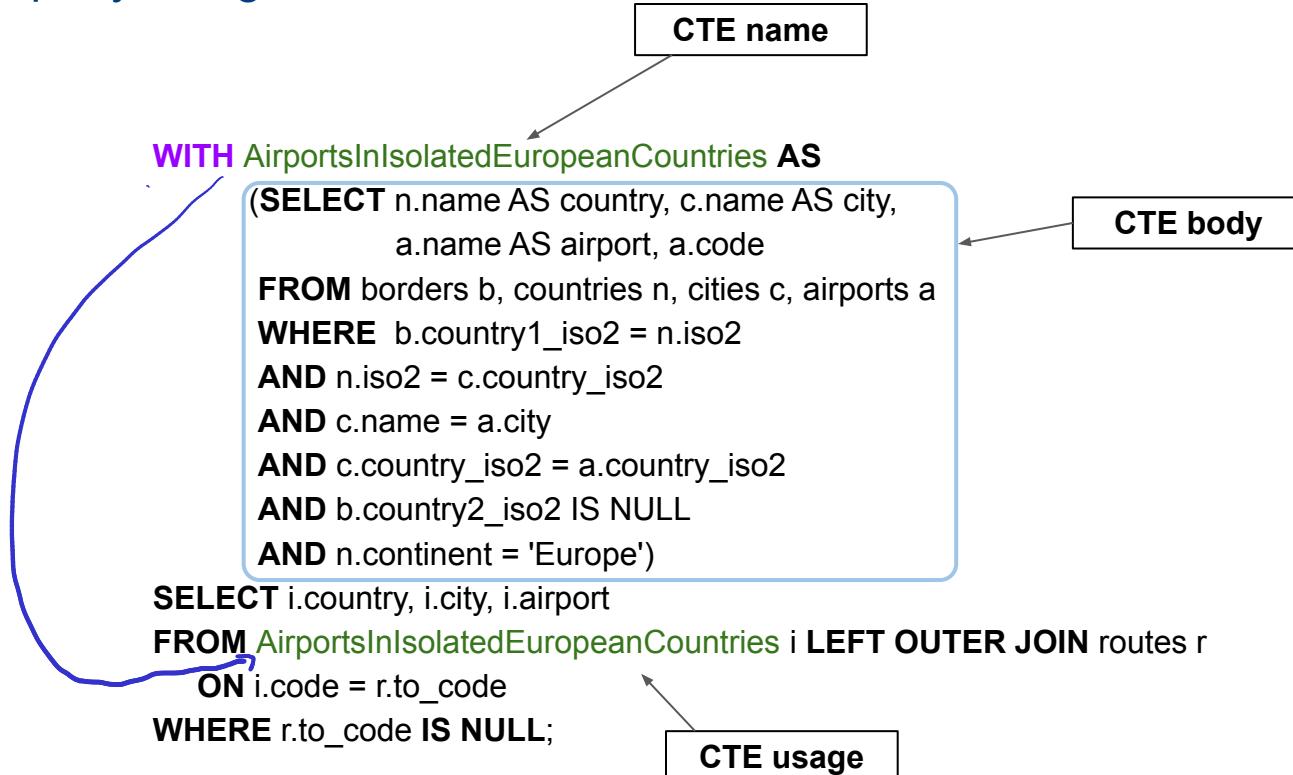
Find all airports in European countries without a land border which cannot be reached by plane given the existing routes in the database.

```
SELECT t1.country, t1.city, t1.airport  
FROM  
(SELECT n.name AS country, c.name AS city,  
       a.name AS airport, a.code  
     FROM borders b, countries n, cities c, airports a  
    WHERE b.country1_iso2 = n.iso2  
      AND n.iso2 = c.country_iso2  
      AND c.name = a.city  
      AND c.country_iso2 = a.country_iso2  
      AND b.country2_iso2 IS NULL  
      AND n.continent = 'Europe') t1  
LEFT OUTER JOIN  
  routes r  
ON t1.code = r.to_code  
WHERE r.to_code IS NULL;
```

Common Table Expressions (CTEs)

country	city	airport
Saint Lucia	Castries	George F. L. Charles Airport

- Same query using a CTE



Common Table Expressions (CTEs)

- General syntax

- Each C_i is the name of a temporary table defined by query Q_i
- Each C_i can reference any other C_j that has been declared before C_i
- SQL statement S can reference any possible subset of all C_i

WITH

$C_1 \text{ AS } (Q_1),$

$C_2 \text{ AS } (Q_2),$

...,

$C_n \text{ AS } (Q_n)$

SQL statement $S;$

- Note

- The goal of using CTEs is not to write less code
- CTEs help to improve readability, debugging, maintenance

Common Table Expressions (CTEs)

country	city	airport
Saint Lucia	Castries	George F. L. Charles Airport

- Extended example

- Multiples CTEs
- CTE referencing previously declared CTE
- CTEs are not required to be referenced

```
WITH IsolatedEuropeanCountries AS (
    SELECT n.iso2, n.name AS country
    FROM borders b, countries n
    WHERE b.country1_iso2 = n.iso2
        AND b.country2_iso2 IS NULL
        AND n.continent = 'Europe'),
AirportsInIsolatedEuropeanCountries AS (
    SELECT n.country, c.name AS city, a.code, a.name AS airport
    FROM IsolatedEuropeanCountries n, cities c, airports a
    WHERE n.iso2 = c.country_iso2
        AND c.name = a.city
        AND c.country_iso2 = a.country_iso2),
UnusedJustForFun AS (
    SELECT COUNT(*)
    FROM IsolatedEuropeanCountries)
SELECT i.country, i.city, i.airport
FROM AirportsInIsolatedEuropeanCountries i LEFT OUTER JOIN routes r
    ON i.code = r.to_code
WHERE r.to_code IS NULL;
```

Overview

- Common SQL constructs
 - Aggregation
 - Grouping
 - Conditional Expressions
- Structuring Queries
 - Common Table Expressions
 - Views
- Extended concepts
 - Universal Quantification
 - Recursive Queries
- Summary

Views — Virtual Tables

- Common observations when querying databases

(beyond the case of increasing complexity of SQL queries)

- Often only parts of a table (rows/columns) are of interest
- Often not all parts of a table (rows/columns) should be accessible to all users
- Often the same queries or subqueries are regularly and frequently used

→ View

- Permanently named query (= virtual table)
- Can be used like normal tables
(with some restrictions; discussed later)
- The result of a query is not permanently stored!
(query is executed each time the view is used)

```
CREATE VIEW <name> AS  
SELECT ...  
FROM ...
```

...

Views — Example

Assumption: Finding all European countries without a land border is a very frequent query.

*Find all airports in **European countries without a land border** which cannot be reached by plane given the existing routes in the database.*

CREATE VIEW IsolatedEuropeanCountries **AS**

```
SELECT n.iso2, n.name AS country  
FROM borders b, countries n  
WHERE b.country1_iso2 = n.iso2  
AND b.country2_iso2 IS NULL  
AND n.continent = 'Europe';
```

WITH AirportsInIsolatedEuropeanCountries **AS** (

```
SELECT n.country, c.name AS city, a.code, a.name AS airport  
FROM IsolatedEuropeanCountries n, cities c, airports a  
WHERE n.iso2 = c.country_iso2  
AND c.name = a.city)
```

SELECT i.country, i.city, i.airport

```
FROM AirportsInIsolatedEuropeanCountries i LEFT OUTER JOIN routes r  
ON i.code = r.to_code  
WHERE r.to_code IS NULL;
```

country	city	airport
Saint Lucia	Castries	George F. L. Charles Airport

Views — Example

CREATE VIEW CountryUrbanizationStats AS

SELECT

n.iso2, n.name, n.population **AS** overall_population, **SUM**(c.population) **AS** city_population,
SUM(c.population) / **CAST**(n.population **AS NUMERIC**) **AS** urbanization_rate

FROM cities c, countries n

WHERE c.country_iso2 = n.iso2

GROUP BY n.iso2, n.name, n.population;

Quick Quiz: Why do we need this?

Find all countries with a urbanization rate below 10%.

```
SELECT name, urbanization_rate  
FROM CountryUrbanizationStats  
WHERE urbanization_rate < 0.1  
ORDER BY urbanization_rate ASC;
```

($\text{alias}_1, \text{alias}_2, \dots$)

integer division $\rightarrow 15 \rightarrow 0$

name	urbanization_rate
Grenada	0.039
Micronesia	0.061
Ethiopia	0.073
Burundi	0.087
Uganda	0.099

Views — Usability

- No restriction when used in SQL queries (**SELECT** statements)
 - But what about **INSERT**, **UPDATE**, **DELETE** statements?

→ Updatable View — requirements

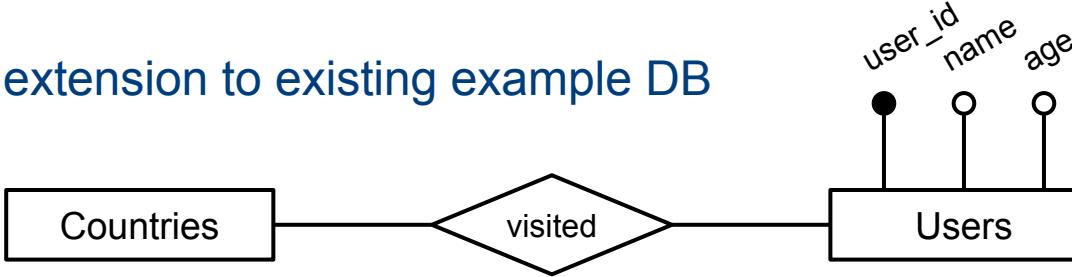
- Only one entry in **FROM** clause (table or updatable view)
- No **WITH**, **DISTINCT**, **GROUP BY**, **HAVING**, **LIMIT**, or **OFFSET**
- No **UNION**, **INTERSECT** or **EXCEPT**
- No aggregate functions
- etc. (incl. no constraint violations)

Overview

- Common SQL constructs
 - Aggregation
 - Grouping
 - Conditional Expressions
- Structuring Queries
 - Common Table Expressions
 - Views
- Extended concepts
 - Universal Quantification
 - Recursive Queries
- Summary

Universal Quantification

- Small extension to existing example DB



- Query with universal quantification

- "Find the names of all users that have visited all countries."*

→ Problem: SQL directly supports only existential quantification (**EXISTS**)



Visited	
user_id	iso2
101	SG
101	DE
103	SG
103	CN
103	FR
...	...

Users		
user_id	name	age
101	Sarah	25
102	Judy	35
103	Max	52
...

Universal Quantification

- "Transformation" of query using logical equivalences
 - "user who visited all countries" → "there does not exists a country the user has not visited"
- Useful subquery
 - All countries a user with user_id = x has not visited

```
SELECT n.iso2  
FROM countries n  
WHERE NOT EXISTS (SELECT 1  
                   FROM visited v  
                   WHERE v.iso2 = n.iso2  
                     AND v.user_id = x);
```

TRUE only for countries that do not have a match
in "Visited" for all tuples where the user_id = x

Universal Quantification

"Find the names of all users that have visited all countries."

```
SELECT user_id, name  
FROM users u  
WHERE NOT EXISTS (SELECT n.iso2  
                   FROM countries n  
                   WHERE NOT EXISTS (SELECT 1  
                                     FROM visited v  
                                     WHERE v.iso2 = n.iso2  
                                       AND v.user_id = u.user_id)  
);
```

user_id	name
103	Max
107	Emma

→ While not overly common, SQL queries requiring universal quantification can get "ugly".

Universal Quantification

- Alternative interpretation
 - "user who visited all countries" → "the number of tuples in "Visited" for that user must match the total number of countries"

"Find the names of all users that have visited all countries."

```
SELECT u.user_id, u.name  
FROM users u, visited v  
WHERE u.user_id = v.user_id  
GROUP BY u.user_id  
HAVING COUNT(*) = (SELECT COUNT(*) FROM countries);
```

user_id	name
103	Max
107	Emma

Overview

- Common SQL constructs
 - Aggregation
 - Grouping
 - Conditional Expressions
- Structuring Queries
 - Common Table Expressions
 - Views
- Extended concepts
 - Universal Quantification
 - **Recursive Queries**
- Summary

Recursive Queries

- Small extension to existing example DB
 - Create table "Connections" as shown
 - Eliminates duplicate routes served by multiple airlines

- Interesting queries

- *"Find all airports that can be reached from SIN non-stop."*

```
SELECT to_code  
FROM connections  
WHERE from_code = 'SIN';
```

90 tuples

```
CREATE TABLE connections AS  
SELECT DISTINCT(from_code, to_code)  
FROM routes;
```

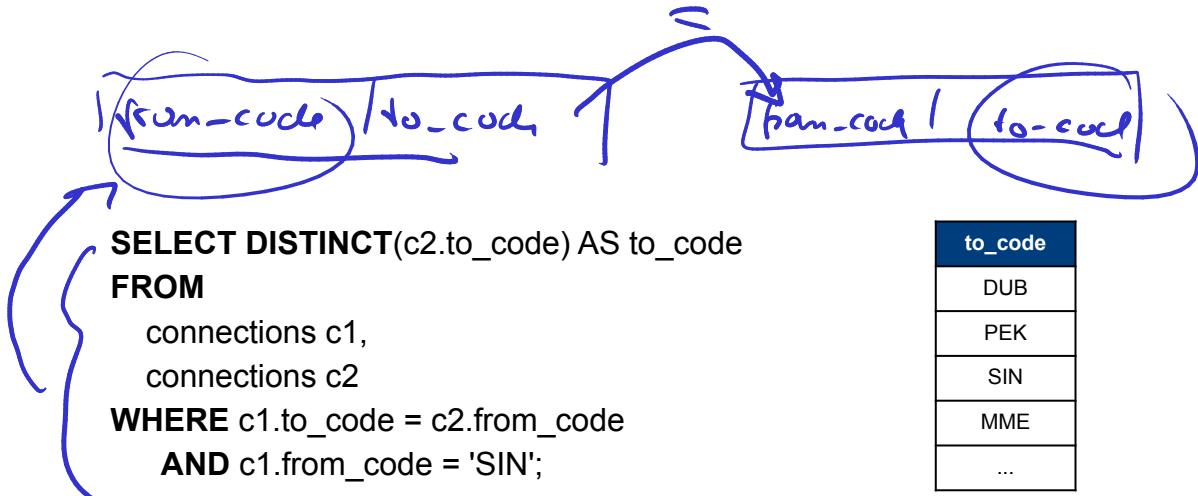
to_code
PEK
BKK
FRA
KUA
...

- *"Find all airports that can be reached from SIN with 1/2/3/... stops." → ???*

Recursive Queries

Find all airports that can be reached from SIN with 1 stop.

825 tuples



Find all airports that can be reached from SIN with 2 stop.

1,561 tuples

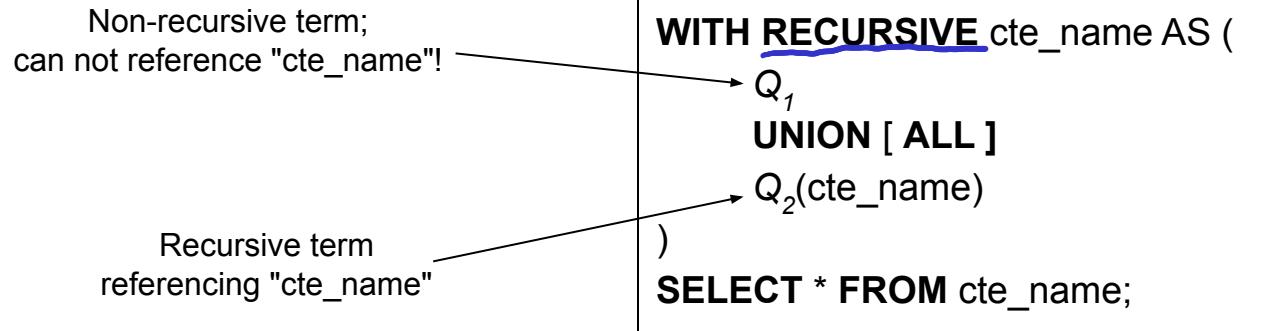
```
SELECT DISTINCT(c3.to_code) AS to_code
FROM
    connections c1,
    connections c2,
    connections c3
WHERE c1.to_code = c2.from_code
    AND c2.to_code = c3.from_code
    AND c1.from_code = 'SIN';
```

to_code
DUB
PEK
SIN
MME
...

Recursive Queries

- Observation: X stops requires query with X joins
 - Requires to write a separate query for each X

→ Recursive Queries using CTEs



Recursive Queries

Find all airports that can be reached from SIN with 0..2 stops.

(limitation to max. 2 stops purely for performance reasons)

```
WITH RECURSIVE flight_path AS (
    SELECT from_code, to_code, 0 AS stops
    FROM connections
    WHERE from_code = 'SIN'
    UNION ALL
    SELECT c.from_code, c.to_code, p.stops+1
    FROM flight_path p, connections c
    WHERE p.to_code = c.from_code
    AND p.stops < 2
)
SELECT DISTINCT to_code, stops
FROM flight_path
ORDER BY stops ASC;
```

base case }
= *no stop* }

to_code	stops	
PEK	0	
BKK	0	
FRA	0	
...	...	
KUA	0	
DUB	1	
PEK	1	
SIN	1	
...	...	
MME	1	
AMS	2	
BKK	2	
PER	2	
...	
ZYL	2	

90 tuples }
825 tuples }
1,561 tuples }

Find all airports that can be reached from SIN with 0..2 stops, including the exact paths.

(limitation to max. 2 stops purely for performance reasons)

WITH RECURSIVE flight_path (airport_codes, stops, is_visited) **AS** (

SELECT

ARRAY[from_code, to_code],

 0 AS stops,

 from_code = to_code

FROM connections

WHERE from_code = 'SIN'

UNION ALL

SELECT

 (airport_codes || to_code)::char(3)[],

 p.stops + 1,

 c.to_code = **ANY**(p.airport_codes)

FROM

 connections c,

 flight_path p

WHERE p.airport_codes[**ARRAY_LENGTH**(airport_codes, 1)] = c.from_code

AND NOT p.is_visited

AND p.stops < 2

)

SELECT DISTINCT airport_codes, stops

FROM flight_path

ORDER BY stops;

basis
case

airport_codes	stops
{SIN, PEK}	0
{SIN, BKK}	0
{SIN, FRA}	0
...	...
{SIN, KUA}	0
{SIN, BKK, PEK}	1
{SIN, FRA, PEK}	1
{SIN, DOH, PEK}	1
...	...
{SIN, MFM, DMK}	1
{SIN, ADL, HKG, PEK}	2
{SIN, ADL, KUL, PEK}	2
{SIN, ADL, SYD, PEK}	2
...
{SIN, TPE, FRA, CSS}	2

90 tuples

4,058 tuples

184,059 tuples

Dealing with the Limitations of (Basic) SQL

- Other types of queries poorly or not support by basic SQL
 - "Sorted by GDP, are there somewhere in the ranking 5 Asian countries listed in a row."
 - Queries/tasks common for time series: moving average, sliding window, etc.
 - Common approaches
 - Keep or move logic into the application
 - Use features that make SQL turing-complete*
(e.g. using SQL/PSM — Persistent Stored Modules)
 - Use a different data model / DBMS
(e.g., a graph database for recursive queries, or time series databases)
- 
- Covered in later lectures

*strictly speaking the support of Recursive CTEs already made SQL turing-complete

Summary

- **Covered: SQL (DQL)**
 - Most common vocabulary for writing queries
 - Basic means to "organize" complex queries (CTEs, Views)

- **Limitations of SQL** (more general: Relational Model)

- Universal quantification
- Recursive queries
- Sequential data
- Graph data
- ...

RDBMS & SQL not the solution for everything

Quick Quiz Solutions

Quick Quiz (Slide 17)

- Solution
 - The query on the left is kind of boring as we have only one table
 - The result will be the name, population, and 1 for each country
- Additional comments
 - Grouping by table's primary key is generally only meaningful if multiple tables are involved (see follow-up example on Slides 18/19)

Quick Quiz (Slide 22)

- Solution
 - The result will be 1 tuple: ('Asia', 1)
 - Again, grouping by a primary key and only one table is generally not meaningful
- Additional comments
 - The full name of China in the database is: "People's Republic of China"
 - If you copy-&-pasted the query, the result will be empty :)

Quick Quiz (Slide 38)

- Solution
 - Both "population" columns are of type Integer, so Integer division is performed
 - For example, with Integer division, $1/3 = 0$
- Additional comments
 - This behavior might differ across different RDBMS implementations

CS2102: Database Systems

Lecture 6 — Relational Algebra

Quick Recap: The Relational Model

- Basic concept: **relations**

- Unified representation of all data using tables with rows and tables
- Relation = set of tuples (row of table) filled with atomic values (or *null*)

- Structural integrity constraints

(condition that restricts what constitutes valid data)

- Domain constraints
- Key constraints
- Foreign key constraints

Table "Movies"

id	title	genre	opened
101	Aliens	action	1986
102	Logan	drama	2017
103	Heat	crime	1995
104	Terminator	action	1984

references relation

Table "Cast"

movie_id	actor_id	role
101	20	Ellen Ripley
101	23	Private Hudson
102	21	Logan
104	23	Punk Leader

referencing relation

Missing: formal method to process and query relations → **Relational Algebra**

Quick Recap: (Structural) Integrity Constraints

- Possible misconception
 - (Foreign) key constraints are not an intrinsic property of a relation
 - Constraints are specified by the DB designer to define what constitutes valid data
- Example from Lecture 1
 - Without any key constraints, relation on the right is perfectly valid → DBMS does not complain
 - Problematic semantics from an application perspective (e.g., CS2102 gives different credits, with and without an exam???)
 - Goal: avoid different values for "mc" and "exam" for the same course → Pick {code} as primary key

code	mc	exam
cs2102	2	yes
cs2102	2	no
cs2102	4	yes
cs2102	4	no
cs3223	2	yes
...
null	4	no
null	null	no
null	null	null

Overview

- **Relation Algebra (RA)**
 - Motivation & overview
 - Closure property
- **Basic operators**
 - Unary operators: selection, projection, renaming
 - Set operators
 - Cross product
- **Join operators**
 - Inner joins
 - Outer joins
- **Complex RA expressions**

Relational Algebra

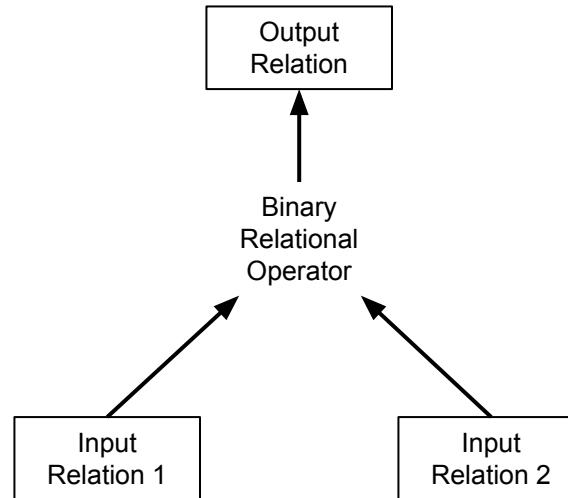
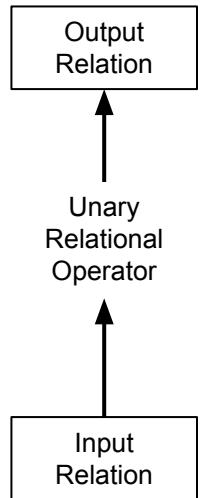
- **Algebra** — mathematical system consisting of
 - Operands: variables or values from which new values can be constructed
 - Operators: symbols denoting procedures that construct new values from given values
- **Relational Algebra** — procedural query language
 - Operands = relations (or variables representing relations)
 - Operators = transform one or more input relations into one output relation
- **Basic operators of the Relational Algebra**
 - Unary operators: selection σ , projection π , (renaming ρ)
 - Binary operators: cross-product \times , union \cup , set difference –

All other RA operators can be expressed using these basic operators

no grouping
no aggregation

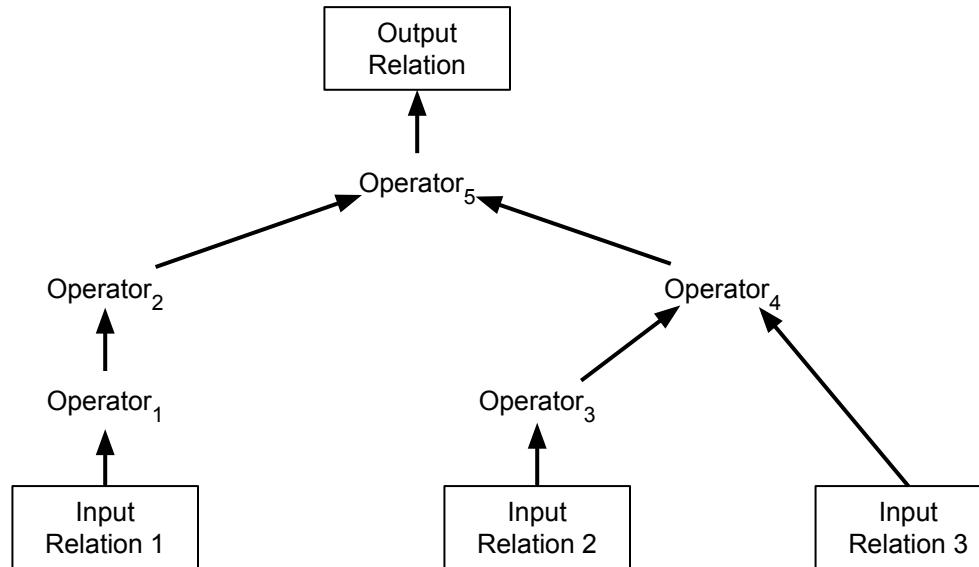
Closure Property

- **Closure:** relations are *closed* under the Relational Algebra
 - All input operands and the outputs of all operators are relations
 - The output of one operator can serve as input for subsequent operators



Closure Property

- Closure property allows for the nesting of relational operators
→ **relational algebra expressions / query trees**



Example Database

- Simplified company database schema (primary keys are underlined)

Employees (name: **text**, age: **integer**, role: **text**)

Managers (name: **text**, office: **text**)

Teams (ename: **text**, pname: **text**, hours: **integer**)

Projects (name: **text**, manager: **text**, start_year: **integer**, end_year: **integer**)

- Foreign key constraints

- Manager.name → Employees.name
- Teams.ename → Employees.name
- Teams.pname → Projects.name
- Projects.manager → Manager.name

Example Database



Teams

ename	pname	hours
Sarah	BigAI	10
Sam	BigAI	5
Bill	BigAI	15
Judy	GlobalDB	20
Max	GlobalDB	5
Sarah	GlobalDB	10
Emma	GlobalDB	35
Max	CoreOS	40
Bill	CoreOS	30
Sam	CoolCoin	40
Sarah	CoolCoin	25
Emma	CoolCoin	10

Projects

name	manager	start_year	end_year
BigAI	Judy	2025	2030
FastCash	Judy	2023	2030
GlobalDB	Jack	2024	2028
CoreOS	Judy	2025	2025
CoolCoin	Jack	2020	2025

Employees

name	age	role
Sarah	25	dev
Judy	35	sales
Max	52	dev
Marie	36	hr
Sam	30	sales
Bernie	19	null
Emma	28	dev
Jack	40	dev
Bill	45	dev

Managers

name	office
Judy	#03-20
Jack	#03-10

Overview

- Relation Algebra (RA)
 - Motivation & Overview
 - Closure Property
- Basic operators
 - **Unary operators:** renaming, selection, projection
 - Set operators
 - Cross product
- Join operators
 - Inner joins
 - Outer joins
- Complex RA expressions

Renaming

- Motivation

- Different relations may share the same attribute
(attributes only unique within same relation)
- RA expression often rely on multiple relations

→ How to ensure unambiguous RA expression?

- Useful: dot notation

- Specify attribute using relation and attribute name
(e.g., Projects.name, Employees.name, Managers.name)
- Problem: insufficient when combining relations
(we will see this more clearly when talking about joins)
- Annoyance: potentially a lot to type

Projects

name	manager	start_year	end_year
BigAI	Judy	2025	2030
FastCash	Judy	2023	2030
...

Employees

name	age	role
Sarah	25	dev
Judy	35	sales
...

Managers

name	office
Judy	#03-20
Jack	#03-10

Renaming $\underline{\rho}$

- 3 ways for renaming

- Renaming a relation: $\rho(R, S)$ (reflect aliases in SQL!)
- Renaming attributes: $\rho(R, R(a_1 \rightarrow b_1, a_1 \rightarrow b_1, \dots))$
- Renaming a relation and attributes: $\rho(R, S(a_1 \rightarrow b_1, a_1 \rightarrow b_1, \dots))$

}
new names only valid
within the scope of the
RA expression!

$$\rho(\text{Managers}, \underline{m}(\text{office} \rightarrow \text{room}))$$

Managers	
name	office
Judy	#03-20
Jack	#03-10

Managers.name Managers.office



m	
name	room
Judy	#03-20
Jack	#03-10

m.name m.room

Selection σ_c

- $\sigma_c(R)$ selects all tuples from a relation R (i.e., rows from a table) that satisfy the *selection condition c*
 - For each tuple $t \in R$, $t \in \sigma_c(R)$ iff c evaluates to **true** on t
 - Input and output relation have the same schema

Example: Find all projects where Judy is the manager

Projects

name	manager	start_year	end_year
BigAI	Judy	2025	2030
FastCash	Judy	2023	2030
GlobalDB	Jack	2024	2028
CoreOS	Judy	2025	2025
CoolCoin	Jack	2020	2025



Uncary option
 \downarrow
 $\sigma_{\text{manager}='Judy'}(\text{Projects})$

name	manager	start_year	end_year
BigAI	Judy	2025	2030
FastCash	Judy	2023	2030
CoreOS	Judy	2025	2025

Selection Conditions \triangleq WHERE clause

- A **selection condition** is boolean expression of one of the following forms:

attribute **op** constant $\sigma_{\text{start_year}=2025}(\text{Projects})$ **constant selection**

attribute₁ **op** attribute₂ $\sigma_{\text{start_year}=\text{end_year}}(\text{Projects})$ **attribute selection**

expr₁ \wedge expr₂ $\sigma_{\text{start_year}=2025 \wedge \text{manager}='Judy'}(\text{Projects})$

expr₁ \vee expr₂ $\sigma_{\text{start_year}=2025 \vee \text{manager}='Judy'}(\text{Projects})$

\neg expr $\sigma_{\neg(\text{start_year}=2025)}(\text{Projects})$

(expr)

- **with**

- **op** $\in \{=, <>, <, \leq, \geq, >\}$

- Operator precedence: (), **op**, \neg , \wedge , \vee

Selection with *null* Values

- Rules of handling *null* values
 - The result of a comparison operation with *null* is *unknown*
 - The result of an arithmetic operation with *null* is *null*
- Examples: assume that the value of x is *null*

$x < 2025$ → *unknown*

$x = \text{null}$ → *unknown*

$x <> \text{null}$ → *unknown*

$x + 5$ → *null*

Employees

name	age	role
...
Sam	30	sales
Bernie	19	<i>null</i>
Emma	28	dev
...

Three-Valued Logic: true, false, unknown

c_1	c_2	$c_1 \wedge c_2$	$c_1 \vee c_2$	$\neg c_1$
false	false	false	false	true
false	unknown	false	unknown	true
false	true	false	true	true
unknown	false	false	unknown	unknown
unknown	unknown	unknown	unknown	unknown
unknown	true	unknown	true	unknown
true	false	false	true	false
true	unknown	unknown	true	false
true	true	true	true	false

Recall: For each tuple $t \in R$, $t \in \sigma_c(R)$ iff c evaluates to true on t

Selection — Example

Example: Find all employees that (a) do not work in Sales and are younger than 30 or (b) work in HR.

Beckie (link 1 true) ∨ false
 $\sigma_{(role <> 'sales' \wedge age < 30) \vee (role = 'hr')}(\text{Employees})$

Employees

name	age	role
Sarah	25	dev
Judy	35	sales
Max	52	dev
Marie	36	hr
Sam	30	sales
Bernie	19	null
Emma	28	dev
Jack	40	dev
Bill	45	dev



name	age	role
Sarah	25	dev
Marie	36	hr
Emma	28	dev

Projection π_ℓ $\hat{=}$ SELECT clause

- $\pi_\ell(R)$ projects all the attributes of a relation specified in list ℓ
 - i.e., projects all columns of a table specified in list ℓ
 - The order of attributes in ℓ matters

Example: Find all projects and their team members

Teams

ename	pname	hours
Sarah	BigAI	10
Sam	BigAI	5
Bill	BigAI	15
Judy	GlobalDB	20
Max	GlobalDB	5
Sarah	GlobalDB	10
Emma	GlobalDB	35
Max	CoreOS	40
Bill	CoreOS	30
Sam	CoolCoin	40
Sarah	CoolCoin	25
Emma	CoolCoin	10

$\pi_{\text{pname}, \text{ename}}(\text{Teams})$



pname	ename
BigAI	Sarah
BigAI	Sam
BigAI	Bill
GlobalDB	Judy
GlobalDB	Max
GlobalDB	Sarah
GlobalDB	Emma
CoreOS	Max
CoreOS	Bill
CoolCoin	Sam
CoolCoin	Sarah
CoolCoin	Emma

Projection π_ℓ

- Relation = set of tuples → duplicate tuples are removed from output relation

Example: Find all projects that have team members.

Teams

ename	pname	hours
Sarah	BigAI	10
Sam	BigAI	5
Bill	BigAI	15
Judy	GlobalDB	20
Max	GlobalDB	5
Sarah	GlobalDB	10
Emma	GlobalDB	35
Max	CoreOS	40
Bill	CoreOS	30
Sam	CoolCoin	40
Sarah	CoolCoin	25
Emma	CoolCoin	10

$\pi_{pname}(\text{Teams})$

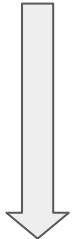


pname
BigAI
GlobalDB
CoreOS
CoolCoin

Quick Quiz

Projects

name	manager	start_year	end_year
BigAI	Judy	2025	2030
FastCash	Judy	2023	2030
GlobalDB	Jack	2024	2028
CoreOS	Judy	2025	2025
CoolCoin	Jack	2020	2025



Which **algebra expression** results in the output below?

manager	name
Judy	FastCash
Jack	GlobalDB
Jack	CoolCoin



A $\sigma_{\text{start_year} \leq 2024}(\pi_{\text{manager}, \text{name}}(\text{Projects}))$

B $\checkmark \pi_{\text{manager}, \text{name}}(\sigma_{\text{start_year} < 2025}(\text{Projects}))$

C $\pi_{\text{manager}, \text{name}}(\sigma_{\text{manager} = \text{'Jack'}}(\text{Projects}))$

D $\pi_{\text{name}, \text{manager}}(\sigma_{\text{start_year} = 2024}(\text{Projects}))$



Overview

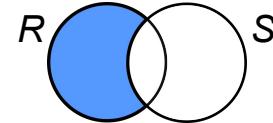
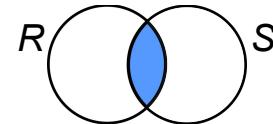
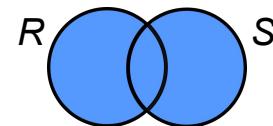
- Relation Algebra (RA)
 - Motivation & Overview
 - Closure Property
- Basic operators
 - Unary operators: selection, projection, renaming
 - **Set operators**
 - Cross product
- Join operators
 - Inner joins
 - Outer joins
- Complex RA expressions

Set Operators

- Recall: a relation is a set of tuples
- Three set operators (given two relation R and S)
 - **Union** $R \cup S$ returns a relation with all tuples that are in both R or S
 - **Intersection** $R \cap S$ returns a relation with all tuples that are in both R and S
 - **Set difference** $R - S$ returns a relation with all tuples that are in R but not in S
- Requirement for all set operators: R and S must be **union-compatible**

Note: The intersection operator is not fundamentally required as it can be expressed with union & difference

$$R \cap S = (R \cup S) - ((R - S) \cup (S - R))$$



Union Compatibility (also: type compatibility)

Note: Just because two relations are union-compatible does not mean that a set operation is semantically meaningful.

- Two relations R and S are **union-compatible** if
 - R and S have the same number of attributes and
 - The corresponding attributes have the same or compatible domains
 - But: R and S do not have to use the same attribute names

Employees (name: **text**, age: **integer**, role: **text**)

Teams (ename: **text**, pname: **text**, hours: **integer**)

Teams

ename	pname	hours
Sarah	BigAI	10
Sam	BigAI	5
...

Employees

name	age	role
Sarah	25	dev
Judy	35	sales
...

not union-compatible

Employees (name: **text**, role: **text**, age: **integer**)

Teams (ename: **text**, pname: **text**, hours: **integer**)

Teams

ename	pname	hours
Sarah	BigAI	10
Sam	BigAI	5
...

Employees

name	role	age
Sarah	dev	25
Judy	sales	35
...

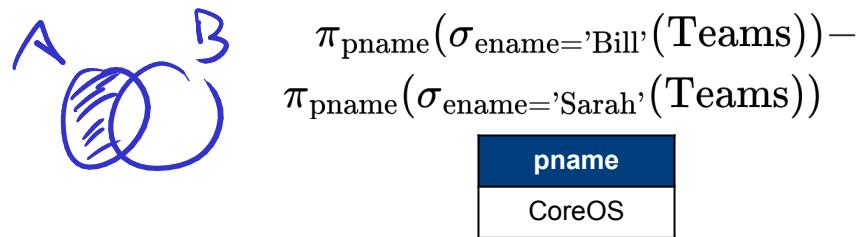
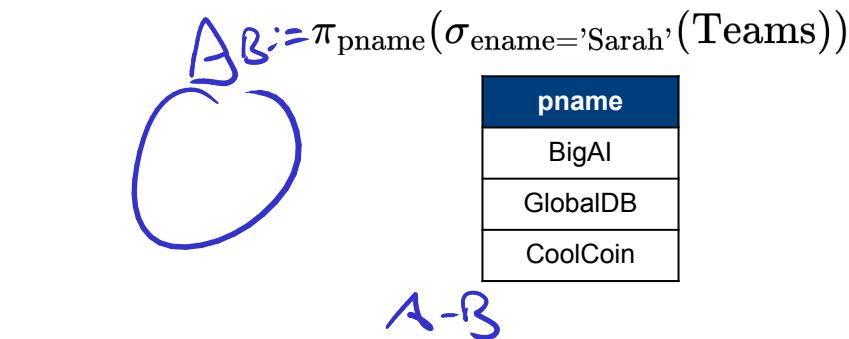
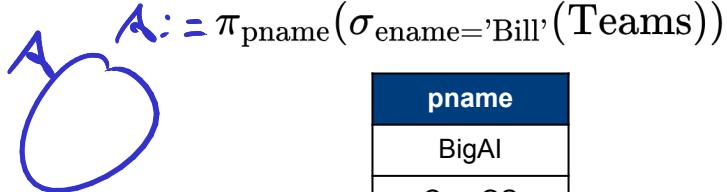
union-compatible

Set Operators — Example

Example: Find all projects that Bill but not Sarah are working on

Teams

ename	pname	hours
Sarah	BigAI	10
Sam	BigAI	5
Bill	BigAI	15
Judy	GlobalDB	20
Max	GlobalDB	5
Sarah	GlobalDB	10
Emma	GlobalDB	35
Max	CoreOS	40
Bill	CoreOS	30
Sam	CoolCoin	40
Sarah	CoolCoin	25
Emma	CoolCoin	10

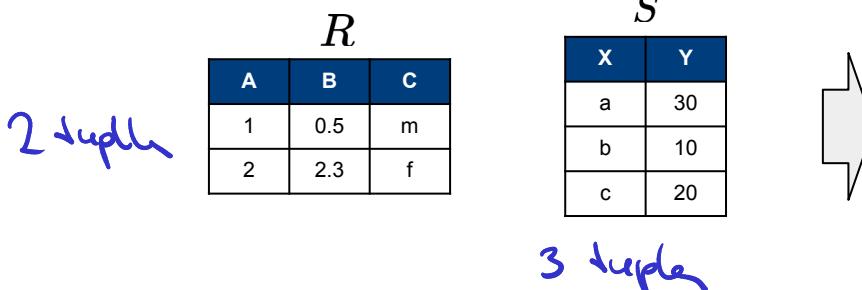


Overview

- Relation Algebra (RA)
 - Motivation & Overview
 - Closure Property
- Basic operators
 - Unary operators: selection, projection, renaming
 - Set operators
 - **Cross product**
- Join operators
 - Inner joins
 - Outer joins
- Complex RA expressions

Cross Product \times (also: Cartesian Product) $A \times B \Rightarrow$ "From A, B"

- The **cross product** combines two relations R and S by forming all pairs of tuples from the two relations
- More formally, given two relations R(A, B, C) and S(X, Y)
 - $R \times S$ returns a relation with schema (A, B, C, X, Y) defined as
 - $R \times S = \{(a, b, c, x, y) \mid (a, b, c) \in R, (x, y) \in S\}$
- Example:



$R \times S$

2x3=6 tuples



A	B	C	X	Y
1	0.5	m	a	30
1	0.5	m	b	10
1	0.5	m	c	20
2	2.3	f	a	30
2	2.3	f	b	20
2	2.3	f	c	10

Cross Product — Example

Example: Find all pairs of senior employees (age ≥ 45) and junior employees (age ≤ 25)

Employees

name	age	role
Sarah	25	dev
Judy	35	sales
Max	52	dev
Marie	36	hr
Sam	30	sales
Bernie	19	null
Emma	28	dev
Jack	40	dev
Bill	45	dev

$$S := \pi_{\text{name}}(\sigma_{\text{age} \geq 45}(\rho(\text{Employees}, S)))$$

↑
Senior

name
Max
Bill

$$J := \pi_{\text{name}}(\sigma_{\text{age} \leq 25}(\rho(\text{Employees}, J(\text{name} \rightarrow \text{jname}))))$$

↑
Junior

name
Sarah
Bernie

$$S \times J$$

name	jname
Max	Sarah
Max	Bernie
Bill	Sarah
Bill	Bernie

Cross Product — Example

Example: For all the projects, find the offices of the managers

join = cross
prob. +
attr. selection

Managers

name	office
Judy	#03-20
Jack	#03-10

Projects

name	manager	start_year	end_year
BigAI	Judy	2025	2030
FastCash	Judy	2023	2030
GlobalDB	Jack	2024	2028
CoreOS	Judy	2025	2025
CoolCoin	Jack	2020	2025

$$M := \text{Projects} \times \rho(\text{Managers}, M(\text{name} \rightarrow \text{mname}))$$

name	manager	start_year	end_year	mname	office
BigAI	Judy	2025	2030	Judy	#03-20
BigAI	Judy	2025	2030	Jack	#03-10
FastCash	Judy	2023	2030	Judy	#03-20
FastCash	Judy	2023	2030	Jack	#03-10
GlobalDB	Jack	2024	2028	Judy	#03-20
GlobalDB	Jack	2024	2028	Jack	#03-10
CoreOS	Judy	2025	2025	Judy	#03-20
CoreOS	Judy	2025	2025	Jack	#03-10
CoolCoin	Jack	2020	2025	Judy	#03-20
CoolCoin	Jack	2020	2025	Jack	#03-10

$$\pi_{\text{name}, \text{office}}(\sigma_{\text{manager} = \text{mname}}(M))$$

name	office
BigAI	#03-20
FastCash	#03-20
GlobalDB	#03-10
CoreOS	#03-20
CoolCoin	#03-10

attribute
selection



Cross Product — Discussion

- **Observation:**

- Given two relations R and S , the size of the cross product is $|R|^*|S|$
- In practice, many to most queries requiring a cross product also require a attribute selection that removes formed pairs of tuples

$$\pi_{\text{name}, \text{office}}(\sigma_{\text{manager}=\text{mname}}(\text{Projects} \times \rho(\text{Managers}, M(\text{name} \rightarrow \text{mname}))))$$

- **Goal: Make use of this observation to**

- simplify Relational Algebra expressions and
- avoid generating all $|R|^*|S|$ output tuples
(when implementing all algebra operators within a DBMS)



→ **Join operators**

Overview

- Relation Algebra (RA)
 - Motivation & Overview
 - Closure Property
- Basic operators
 - Unary operators: selection, projection, renaming
 - Set operators
 - Cross product
- Join operators
 - Inner joins
 - Outer joins
- Complex RA expressions

Join Operators

- **Join operator — basic idea**
 - Combines cross product, attribute selection and (possibly projection into a single operator)
 - Typically results in simpler relational algebra expressions when formulating queries

- **Base types**

Only the tuples that satisfy matching criteria are included in the final result

Join

Tuples that do and do not satisfy the matching criteria are included in the final result

Inner Join

- \bowtie_θ θ -join
- \bowtie equi join
- \bowtie natural join

Outer Join

- \bowtie_{left} left outer join
- \bowtie_{right} right outer join
- \bowtie_{full} full outer join

Inner Joins — θ -Join

- The θ -join $R \bowtie_{\theta} S$ of two relations R and S is defined as

$$R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$$

Managers

name	office
Judy	#03-20
Jack	#03-10

Example: For all the projects,
find the offices of the managers

Note: final projection omitted
here to show result of θ -join

Projects

name	manager	start_year	end_year
BigAI	Judy	2025	2030
FastCash	Judy	2023	2030
GlobalDB	Jack	2024	2028
CoreOS	Judy	2025	2025
CoolCoin	Jack	2020	2025



Projects $\bowtie_{\text{manager}=\text{mname}} \rho(\text{Managers}, M(\text{name} \rightarrow \text{mname}))$

name	manager	start_year	end_year	mname	office
BigAI	Judy	2025	2030	Judy	#03-20
FastCash	Judy	2023	2030	Judy	#03-20
GlobalDB	Jack	2024	2028	Jack	#03-10
CoreOS	Judy	2025	2025	Judy	#03-20
CoolCoin	Jack	2020	2025	Jack	#03-10

Inner Joins — Equi Join $\bowtie_{=}$

- Difference between θ -join and equi join is only w.r.t. matching criteria
 - The θ -join \bowtie_{θ} allows arbitrary comparison operators for the attribute selection (e.g., $=$, $<>$, $<$, \leq , \geq , $>$)
 - The equi join \bowtie is a special case of θ -join by defined over the equality operator ($=$) only
 - Dedicated term since attribute selections using the equality operator are most common (particularly when joining along foreign key constraints)
- Example
 - see previous slide

Natural Join

- Same as equi join (i.e., only equality operator) but:
 - The join is performed over all attributes that R and S have in common (this means that no explicit matching criteria has to be specified)
 - The output relations contains the common attributes of R and S only once (compared with the θ -join and equi join that contain all attributes of both relations)
- More formally, the **natural join** of two relation R and S is defined as

$$R \bowtie S = \pi_{\underline{\ell}}(R \bowtie_c \rho_{b_i \leftarrow a_i, \dots, b_k \leftarrow a_k}(S))$$

- $A = \{a_i, \dots, a_k\}$ is the set of attributes that R and S have in common
- $c = ((a_i = b_i) \wedge \dots \wedge (a_k = b_k))$
- $\ell = \text{list of all attributes of } R + \text{list of all attributes in } S \text{ that are not in } R$

Natural Join — Example

Quick Quiz: What would be the result of
Projects \bowtie Managers

Example: For all the projects,
find the offices of the managers

Managers

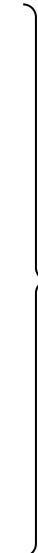
name	office
Judy	#03-20
Jack	#03-10

$\rho(\text{Managers}, m(\text{name} \rightarrow \text{manager}))$



Projects

name	manager	start_year	end_year
BigAI	Judy	2020	2025
FastCash	Judy	2018	2025
GlobalDB	Jack	2019	2023
CoreOS	Judy	2020	2020
CoolCoin	Jack	2015	2020



Note: final projection omitted
to show result of natural join

Projects \bowtie $\rho(\text{Managers}, m(\text{name} \rightarrow \text{manager}))$

name	manager	start_year	end_year	office
BigAI	Judy	2020	2025	#03-20
FastCash	Judy	2018	2025	#03-20
GlobalDB	Jack	2019	2023	#03-10
CoreOS	Judy	2020	2020	#03-20
CoolCoin	Jack	2015	2020	#03-10

Outer Joins

Note: This simple example can easily be solved using projection and set difference.

- **Motivation**

- Inner joins eliminate all tuples that do not satisfy matching criteria (i.e., attribute selection)
- Sometimes the tuples in R or S that do not match with tuples in the other relation are of interest
 - **dangling tuples**

Example: Find all employees that are not assigned to any project.

An inner join can only find all employees that are assigned to at least one project.

Employees

name	age	role
Sarah	25	dev
Judy	35	sales
Max	52	dev
Marie	36	hr
Sam	30	sales
Bernie	19	null
Emma	28	dev
Jack	40	dev
Bill	45	dev

Teams

ename	pname	hours
Sarah	BigAI	10
Sam	BigAI	5
Bill	BigAI	15
Judy	GlobalDB	20
Max	GlobalDB	5
Sarah	GlobalDB	10
Emma	GlobalDB	35
Max	CoreOS	40
Bill	CoreOS	30
Sam	CoolCoin	40
Sarah	CoolCoin	25
Emma	CoolCoin	10

Outer Joins

Quick Quiz: Why is there
 $\pi_{ename}(\text{Teams})$
and do we really need it?

- Processing steps of an outer join between R and S (informal)

- Perform inner join $M = R \bowtie_{\theta} S$

- To M , add dangling tuples to result of

R in case of a **left outer join** \bowtie_{θ}

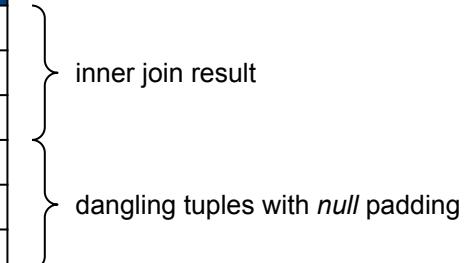
S in case of a **right outer join** \bowtie_{θ}

R and S in case of a **full outer join** \bowtie_{θ}

- "Pad" missing attribute values of dangling tuples with *null*

$\text{Employees} \bowtie_{name=ename} (\pi_{ename}(\text{Teams}))$

name	age	role	ename
...
Emma	28	dev	Emma
Bill	45	dev	Bill
Marie	36	hr	<i>null</i>
Bernie	19	<i>null</i>	<i>null</i>
Jack	40	dev	<i>null</i>



Example: Find all employees that are not assigned to any project.

Outer Joins — Formal Definitions

- **Auxiliary definitions**

- $dangle(R \bowtie_{\theta} S)$ = set of dangling tuples in R w.r.t. $R \bowtie_{\theta} S$

$$\rightarrow dangle(R \bowtie_{\theta} S) \subseteq R$$

- $null(R)$ = n -component tuple of null values where n is the number of attributes of R
e.g., $null(\text{Teams}) = (\text{null}, \text{null}, \text{null})$

- **Definitions (outer joins)**

Left outer join $R \bowtie_{\theta} S = R \bowtie_{\theta} S \cup (dangle(R \bowtie_{\theta} S) \times \{null(S)\})$

Right outer join $R \bowtie_{\theta} S = R \bowtie_{\theta} S \cup (\{null(R)\} \times dangle(S \bowtie_{\theta} R))$

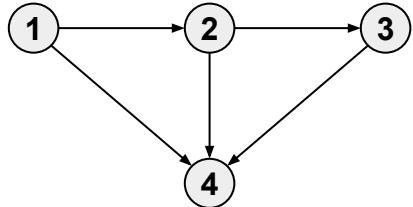
Full outer join $R \bowtie_{\theta} S = R \bowtie_{\theta} S \cup (dangle(R \bowtie_{\theta} S) \times \{null(S)\}) \cup (\{null(R)\} \times dangle(S \bowtie_{\theta} R))$


inner join


dangling tuples with *null* padding

name	age	role	ename
...
Jack	40	dev	Jack
Bill	45	dev	Bill
Marie	36	hr	<i>null</i>
Bernie	19	<i>null</i>	<i>null</i>

Full Outer Join — Example



Example: Find all nodes with no incoming or outgoing edge

$\text{Edges} \bowtie_{t=\text{in}} \rho(\text{Edges}, e(s \rightarrow \text{in}, t \rightarrow \text{out}))$

Edges

s	t
1	2
1	4
2	3
2	4
3	4

no incoming {

s	t	in	out
null	null	1	2
null	null	1	4
1	2	2	3
1	2	2	4
2	3	3	4
1	4	null	null
2	4	null	null
3	4	null	null

} no outgoing

Natural Outer Joins

- Analog to natural (inner) join
 - Only the equality operator is used for the join condition
 - The join is performed over all attributes that R and S have in common
 - The output relations contains the common attributes of R and S only once

Natural left outer join $R \bowtie S$

Natural right outer join $R \bowtie S$

Natural full outer join $R \bowtie S$

Edges	
s	t
1	2
1	4
2	3
2	4
3	4

Quick Quiz: What is the result of the expression:
 $\text{Edges} \bowtie \text{Edges}$

Quick Quiz

Teams

ename	pname	hours
Sarah	BigAI	10
Sam	BigAI	5
Bill	BigAI	15
Judy	GlobalDB	20
Max	GlobalDB	5
Sarah	GlobalDB	10
Emma	GlobalDB	35
Max	CoreOS	40
Bill	CoreOS	30
Sam	CoolCoin	40
Sarah	CoolCoin	25
Emma	CoolCoin	10

Managers

name	office
Judy	#03-20
Jack	#03-10

How many **rows & columns** has the result of the algebra expression below?

$$\sigma_{ename \equiv null}(\text{Managers} \bowtie_{name=ename} \text{Teams})$$

A

1 row, 5 cols

B

2 rows, 5 cols

C

1 row, 3 cols

D

2 rows, 3 cols

Note: We use \equiv to check if a value is null or not; recall that the basic comparison with $=$ will yield "unknown" and thus not return the desired result. SQL has the special comparison "... IS (NOT) NULL" (cf. later lectures) which the basic Relational Algebra lacks.

Overview

- Relation Algebra (RA)
 - Motivation & Overview
 - Closure Property
- Basic operators
 - Unary operators: selection, projection, renaming
 - Set operators
 - Cross product
- Join operators
 - Inner joins
 - Outer joins
- Complex RA expressions

Complex Relational Expressions (Queries)

Example: Find all managers (with their offices) of projects that started 2025 or later, where at least one member of the project team has to work 30h or more on that project per week!

$$P := \sigma_{\text{start_year} \geq 2025}(\text{Projects})$$

name	manager	start_year	end_year
BigAI	Judy	2025	2030
CoreOS	Judy	2025	2030

$$M := \pi_{\text{name}, \text{manager}, \text{office}}(P \bowtie \rho(\text{Managers}, M(\text{name} \rightarrow \text{manager})))$$

name	manager	office
BigAI	Judy	#03-20
CoreOS	Judy	#03-20

$$W := \sigma_{\text{hours} \geq 30}(\text{Teams})$$

ename	pname	hours
Emma	GlobalDB	35
Max	CoreOS	40
Bill	CoreOS	30
Sam	CoolCoin	40

$$T := W \bowtie_{\text{pname} = \text{name}} M$$

ename	pname	hours	name	manager	office
Max	CoreOS	40	CoreOS	Judy	#03-20
Bill	CoreOS	30	CoreOS	Judy	#03-20

Managers

name	office
Judy	#03-20
Jack	#03-10

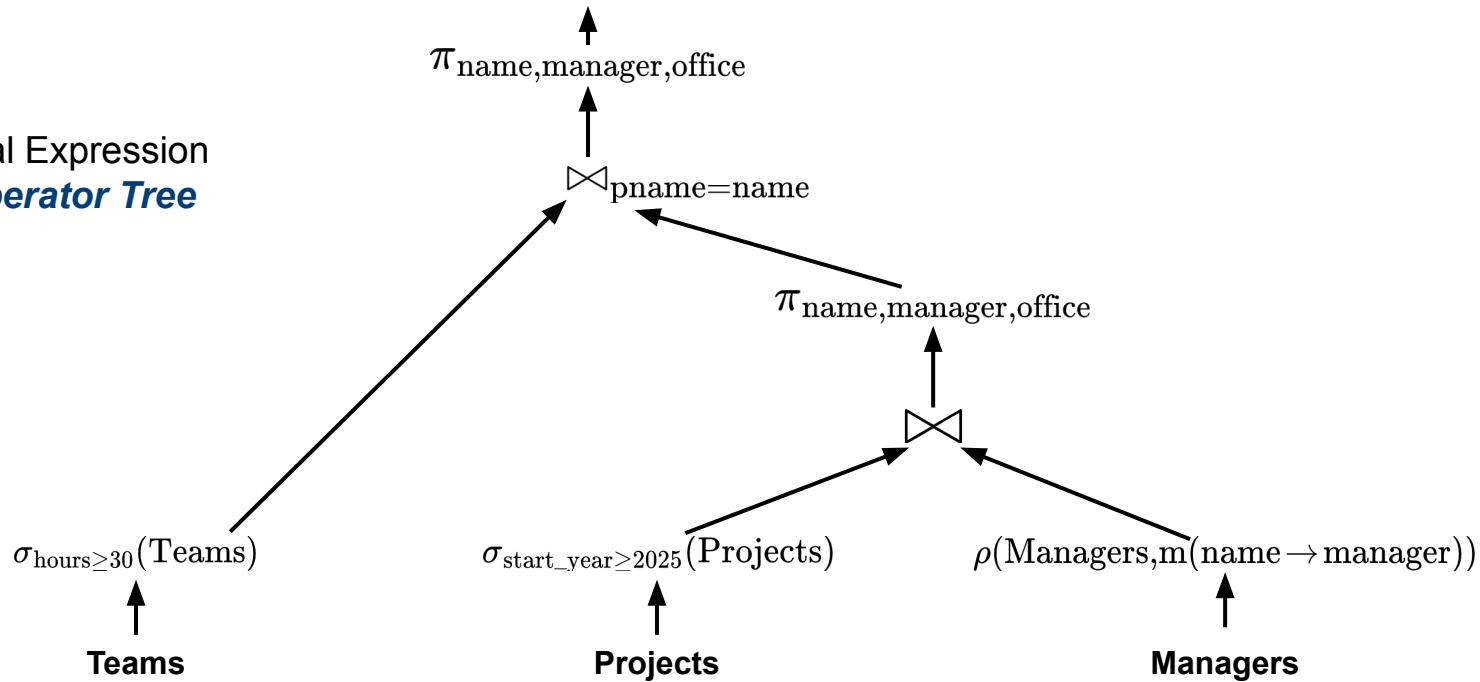
$$\pi_{\text{name}, \text{manager}, \text{office}}(T)$$

name	manager	office
CoreOS	Judy	#03-20

Complex Relational Expressions (Queries)

Example: Find all managers (with their offices) of projects that started 2025 or later, where at least one member of the project team has to work more 30h or more on that project per week!

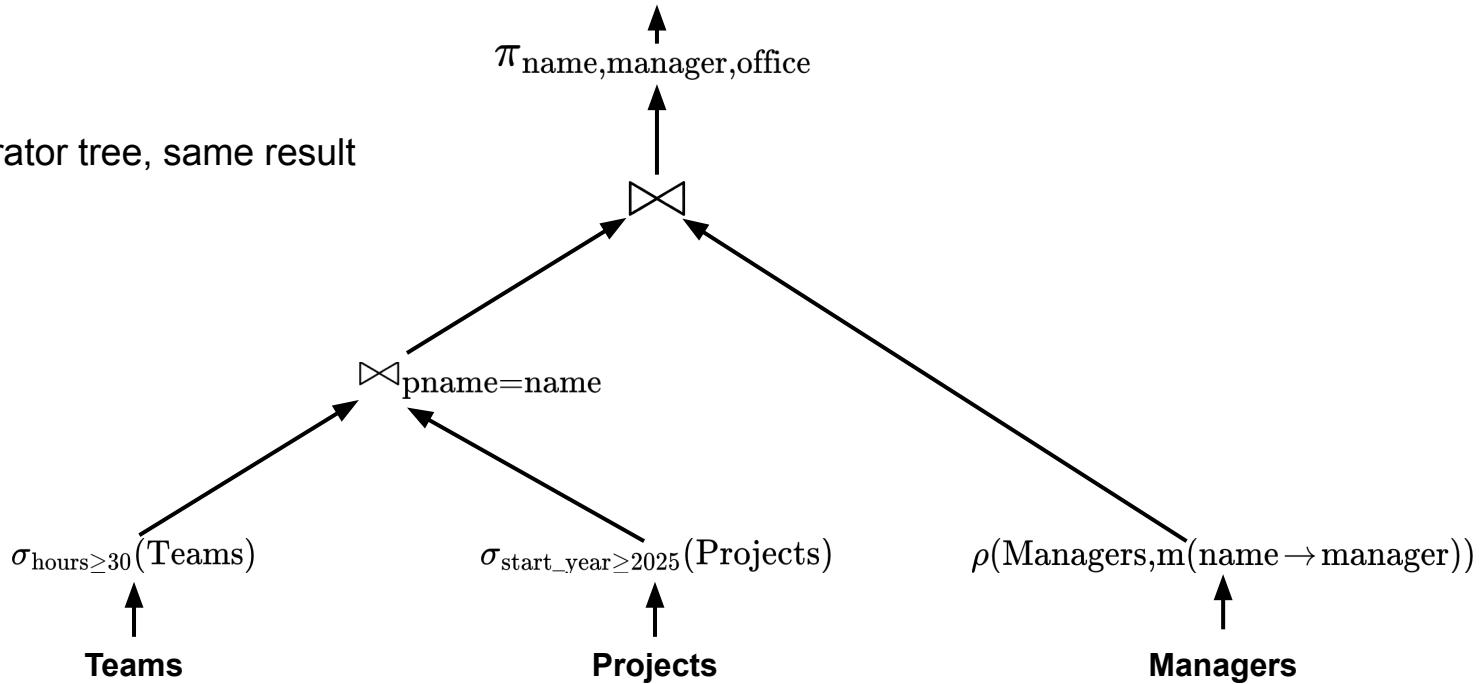
Relational Expression
as an *Operator Tree*



Complex Relational Expressions (Queries)

Example: Find all managers (with their offices) of projects that started 2025 or later, where at least one member of the project team has to work more 30h or more on that project per week!

Different operator tree, same result



Complex Relational Expressions — Observation

- In general, multiple ways to formulate a query to get the same result, e.g.,
 - Order in which join operations are performed
 - Order in which selection operations are performed (e.g., before or after join operators)
 - Inserting additional projections to minimize intermediate results
 - ...and many more
- Finding the "best" operator tree → query optimization
 - Handled by the DBMS transparent to the user
 - Covered in, e.g., CS3223

Invalid Relational Expressions — Examples

- Attribute no longer available after projection

$$\sigma_{\text{role}='dev'}(\pi_{\text{name}, \text{age}}(\text{Employees}))$$

- Attribute no longer available after renaming

$$\sigma_{\text{role}='dev'}(\rho(\text{Employees}, \text{emp}(\text{role} \rightarrow \text{position})))$$

- Incompatible attribute types

$$\sigma_{\text{age}=\text{role}}(\text{Employees})$$

Valid but not "Smart" Expressions — Examples

- Cross product + attribute selection instead join

$$\sigma_{\text{manager}=\text{m.name}}(\text{Projects} \times \rho(\text{Managers}, \text{m})) \rightarrow \text{Projects} \bowtie_{\text{manager}=\text{m.name}} \rho(\text{Managers}, \text{m})$$

- Unnecessary operators

$$\pi_{\text{name}}(\pi_{\text{name}, \text{age}}(\text{Employees})) \rightarrow \pi_{\text{name}}(\text{Employees})$$

- Query optimization (performance)

$$\sigma_{\text{start_year}=2025}(\text{Projects} \bowtie_{\text{manager}=\text{m.name}} \rho(\text{Managers}, \text{m}))$$

$$\rightarrow (\sigma_{\text{start_year}=2025}(\text{Projects})) \bowtie_{\text{manager}=\text{m.name}} \rho(\text{Managers}, \text{m})$$

Note: query optimization is beyond the scope of CS2102 and covered in other courses (e.g., CS3223). A solid grasp of the Relational Algebra is very important for this topic.

Relational Algebra vs. SQL

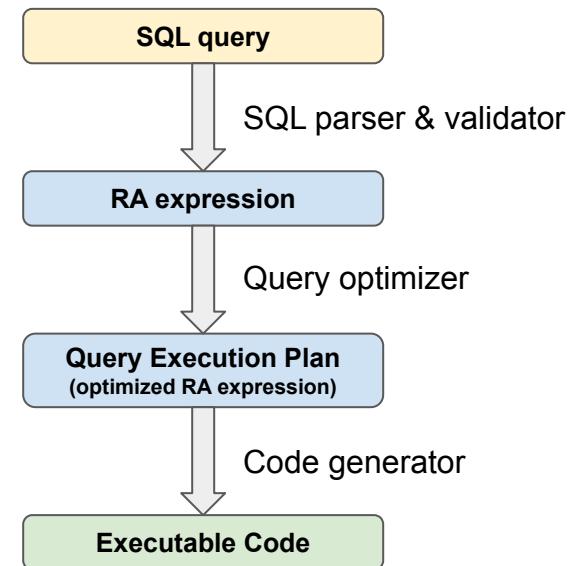
- **Relation Algebra (RA)**

- Procedural query language using operators to perform queries
- Query = relational algebra expression (operator tree)

- **SQL** (more precisely: the DQL part of SQL)

- **Declarative** query language built on top of RA
(Focus on *what* to compute, not on *how* to compute)
- Multiset / bag semantics (unlike sets in RA!)
- Query = SELECT statement

```
SELECT      [ DISTINCT ] target-list
FROM        relation-list
[ WHERE     conditions ]
```



Relational Algebra vs. SQL

- Mapping between basic SQL query to corresponding RA expression
 - Conceptual mapping; no consideration of performance

SELECT DISTINCT a_1, a_2, \dots, a_m
FROM r_1, r_2, \dots, r_n
WHERE c



$\pi_{a_1, a_2, \dots, a_m}(\sigma_c(r_1 \times r_2 \times \dots \times r_n))$

- Practical use when working with database
 - Database tuning (e.g., if different queries yield the same result, which runs faster)
 - Relevant command: **EXPLAIN** (shows query plan with operators and specific implementations)

Recall: Flexibility of SQL

Find all names that refer to both a city and a country.

(**SELECT** name **FROM** cities)

INTERSECT

(**SELECT** name **FROM** countries);



name
Singapore
Mexico
Peru
Monaco
Mali
El Salvador
China
Poland
...

29 tuples

SELECT DISTINCT(name)
FROM (**SELECT** name **FROM** cities) t1
NATURAL JOIN
(**SELECT** name **FROM** countries) t2;

SELECT n.name
FROM countries n
WHERE EXISTS (**SELECT** c.name
FROM cities c
WHERE c.name = n.name);

SELECT name
FROM countries
WHERE name **IN** (**SELECT** name
FROM cities);

Recall: Flexibility of SQL

EXPLAIN SELECT n.name

```
FROM countries n  
WHERE EXISTS (SELECT c.name  
              FROM cities c  
              WHERE c.name = n.name);
```

EXPLAIN SELECT name

```
FROM countries  
WHERE name IN (SELECT name  
                FROM cities);
```

	QUERY PLAN	text	🔒
1	Nested Loop Semi Join	(cost=0.29..549.36 rows=196 width=9)	
2	-> Seq Scan on countries n	(cost=0.00..3.96 rows=196 width=9)	
3	-> Index Only Scan using cities_pkey on cities c	(cost=0.29..2.82 rows=1 width...)	
4	Index Cond:	(name = (n.name)::text)	

Recall: Flexibility of SQL

EXPLAIN

(SELECT name FROM cities)

INTERSECT

(SELECT name FROM countries);

	QUERY PLAN text	🔒
1	HashSetOp Intersect (cost=0.00..1358.19 rows=196 width=520)	
2	-> Append (cost=0.00..1257.35 rows=40334 width=520)	
3	-> Subquery Scan on "*SELECT* 2" (cost=0.00..5.92 rows=196 width=13)	
4	-> Seq Scan on countries (cost=0.00..3.96 rows=196 width=9)	
5	-> Subquery Scan on "*SELECT* 1" (cost=0.00..1049.76 rows=40138 width...)	
6	-> Seq Scan on cities (cost=0.00..648.38 rows=40138 width=10)	

EXPLAIN SELECT DISTINCT(name)

FROM (SELECT name FROM cities) t1

NATURAL JOIN

(SELECT name FROM countries) t2;

	QUERY PLAN text	🔒
1	HashAggregate (cost=558.70..560.73 rows=203 width=10)	
2	Group Key: cities.name	
3	-> Nested Loop (cost=0.29..558.19 rows=203 width=10)	
4	-> Seq Scan on countries (cost=0.00..3.96 rows=196 width=9)	
5	-> Index Only Scan using cities_pkey on cities (cost=0.29..2.82 rows=1 width...)	
6	Index Cond: (name = (countries.name)::text)	

Summary

- **Relational Algebra**
 - Formal method to query relational data
 - Closure property for arbitrarily complex relational expressions
 - Basis for DB query languages such as SQL
 - Important for query (automated) optimization
- **Most common operators**
 - Unary operators: selection, projection, renaming
 - Binary operators: set operators, (cross product), joins

Quick Quiz Solutions

Quick Quiz (Slide 20)

- Solution: **B**
 - A: Invalid expression since projection removes attribute "start_year"
 - C: Output relation still contains projects with manager "Judy"
 - D: Order of attributes of the output relation are flipped

Quick Quiz (Slide 35)

- Solution
 - The output relation will be empty
 - Without the renaming, "Projects" and "Managers" still have an attribute "name" in common
 - However, "Projects.name" and "Managers.name" are semantically different
(names of projects vs names of people)

- Additional comments
 - If a project would be called, say, "Judy", then the result wouldn't be empty anymore
(note that the attributes are still semantically different just happened to have shared attribute values)

Quick Quiz (Slide 37)

- Solution

- The projection is not needed to get the important information; it just makes the output simpler
- Of course, strictly speaking, the output would change without the projection
(it just provided any additional information required to answer the query)
- Without the projection, the output relation would look like:

name	age	role	ename	pname	hours
...
Emma	28	dev	Emma	CoolCoin	10
Bill	45	dev	Bill	CoreOS	30
Marie	36	hr	<i>null</i>	<i>null</i>	<i>null</i>
Bernie	19	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
Jack	40	dev	<i>null</i>	<i>null</i>	<i>null</i>

{ inner join result
dangling tuples with *null* padding }

Quick Quiz (Slide 40)

- Solution
 - The output relation will look the same as "Edges"
 - We do a Natural Join of table with itself → "both" input relations have all attributes in common
 - Recall that any outer join also includes the result of the inner join,
and in this case, each tuples matches with itself
 - Lastly, the result of Natural Join only retains one set of the share attributes which is just (s, t) here

Quick Quiz (Slide 41)

- **Solution: A (1 row, 5 cols)**

- The expression returns all dangling tuples of the Left Outer Join between "Managers" and "Teams" (in other words, this expression returns all managers who are not directly working on a project themselves)
- This is only true for manager Jack, so the output relation has only 1 row/tuple.
- Since the expression does not contain any projection,
the output relation contains all $3+2=5$ attributes of both input relations