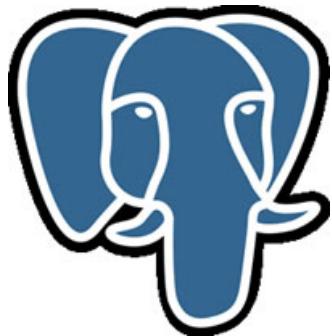


# COMP3311 Course Overview

- Why Study Databases?
- Databases: Important Themes
- What is Data? What is a Database?
- Studying Databases in CSE
- Syllabus Overview
- Your Background
- Teaching/Learning
- Assignments
- Quizzes
- Exam
- Supplementary Assessment Policy
- Assessment Summary
- Textbook (options)
- Database Management Systems
- Further Reading Material
- Home Computing
- Overview of the Databases Field
- Database Application Development
- Database System Architecture

# COMP3311 Database Systems



Lecturer: *John Shepherd* ([cs3311@cse.unsw.edu.au](mailto:cs3311@cse.unsw.edu.au))

Web Site: <http://webcms3.cse.unsw.edu.au/COMP3311/21T3/>  
or <http://www.cse.unsw.edu.au/~cs3311/>

(If Webcms3 is unavailable, try <http://www.cse.unsw.edu.au/~cs3311/21T3/>)

## ❖ Why Study Databases?

Every significant computer application involves Large Data.

This needs to be:

- **stored** (typically on a disk device)
- **manipulated** (efficiently, usefully)
- **shared** (by many users, concurrently)
- **transmitted** (all around the Internet)

**Green** stuff handled by databases; **blue** by networks.

Challenges in building effective databases: efficiency, security, scalability, maintainability, availability, integration, new media types (e.g., music), ...

## ❖ Databases: Important Themes

The field of **databases** deals with:

- **data** ... representing application scenarios
- **relationships** ... amongst data items
- **constraints** ... on data and relationships
- **redundancy** ... one source for each data item
- **data manipulation** ... declarative, procedural
- **transactions** ... multiple actions, atomic effect
- **concurrency** ... multiple users sharing data
- **scale** ... massive amounts of data

## ❖ What is Data? What is a Database?

According to the Elmasri/Navathe textbook ...

- **Data** = known recorded facts, with implicit meaning
  - e.g. a student's name, a product id, a person's address or birthday
- **Database** = collection of related data, satisfying constraints
  - e.g. a student *is enrolled in* a course, a product *is sold at* a store
- **DBMS** = database management system
  - software to manage data, control access, enforce constraints
- **RDBMS** = relational database management system
  - e.g. PostgreSQL, SQLite, Oracle, SQL Server, MySQL, ...

## ❖ Studying Databases in CSE

COMP3311 introduces foundations & technology of databases

- skills: how to build database-backed applications
- theory: how do you know that what you built was good

After COMP3311 you can go on to study ...

- COMP9313: managing Big Data (Hadoop, Spark, NoSQL techniques)
- COMP9315: how to build relational DBMSs (write your own PostgreSQL)
- COMP9318: techniques for data mining (discovering patterns in DB)
- COMP9319: Web data compression and search (XML data)
- COMP6714: information retrieval, web search (dealing with text data)
- COMP9321: data services (making data available via a network)

## ❖ Syllabus Overview

Core syllabus ...

- Data modelling and database design
  - ER model, **ODL**, ER-to-relational
  - Relational model (design theory, algebra)
- Database application development
  - SQL, views, stored procedures, triggers, aggregates
  - SQLite: `sqlite3` (an SQL shell)
  - PostgreSQL: `psql` (an SQL shell), `PLpgSQL` (procedural),
  - Programming language access to databases (Python, **ORMs**)

The **brown stuff** is not covered in tutes/pracs and is not examinable

## ❖ Syllabus Overview (cont)

More syllabus ...

- Database management systems (DBMSs)
  - DBMS architecture: query processing, index structures
  - Transaction processing: transactions, concurrency control, recovery
- Future of Databases
  - Limitations of RDBMS's, potential future technologies

Blue and green stuff is covered only briefly, and is not examinable

To learn more about the green stuff, take COMP9313, ...

To learn more about the blue stuff, take COMP9315, ...

## ❖ Your Background

We assume that you ...

- have experience with procedural programming
- have some background in data structures
- hopefully, have some knowledge of Python

You might have acquired this background in

- COMP1511, COMP1531, **COMP2521**

If you don't know Python, look at some online tutorials soon.

e.g. <https://www.python.org/about/gettingstarted/>

## ❖ Teaching/Learning

Stuff that is available for you:

- **Textbooks**: describe **most** syllabus topics in detail
- **Topic Videos**: summarize **all** syllabus topics
- **Problem-solving sessions**: work through examples
- **Tutorial sessions**: theory/prac questions (+ solutions)
- **Prac exercises**: lab-like exercises
- **Assignments**: more detailed practical exercises
- **Quizzes**: periodic progress check

All online. If you want on-campus, wait for ~~COMP3311 21T1 ???~~

## ❖ Teaching/Learning (cont)

On the course website, you can:

- find out the latest course news  
(important announcements will also be emailed)
- view the topic-based slides/videos
- get details of tute/prac exercises
- get assignment specs/material
- do the quizzes
- get your questions answered (via the Forums)

URL: <https://webcms3.cse.unsw.edu.au/COMP3311/21T3/>

(If Webcms3 is unavailable, try <http://www.cse.unsw.edu.au/~cs3311/21T3/>)

## ❖ Assignments

Two assignments, which are **critical** for **learning**

1. SQL/PLpgsql, 15%, due end week 5
2. Python/SQL/psycopg2, 20%, due end week 9

All assignments are done **individually**, and ...

- submitted via **give** or Webcms3
- automarked (so you must follow specification exactly)
- plagiarism-checked (copying solutions  $\Rightarrow$  **0** mark for assignment)
- rent-a-coder monitored (buying solutions  $\Rightarrow$  **exclusion**)

## ❖ Quizzes

Six quizzes, each worth 4 marks

- cover material in previous few weeks lectures
- aim to check your understanding of recent material
- done via Webcms3 in your own time
- primarily multiple-choice
- held in weeks 2, 3, 4, 7, 8, 10
- released Monday, due Friday 9pm
- can be submitted multiple times

$6 \times 4 = 24$ , which is mapped into a mark out of 15

Heavy penalties for late submission

## ❖ Exam

The Final Exam includes questions on ...

- SQL, PLpgSQL, (Python), design exercises, analyses
- 60% prac questions, 40% "theory" questions

Online, open-web exam during exam period

- exam is open for 4 hours
- content is what I'd put in a 3-hour in-lab exam
- can work on home machine, or via **ssh**, or via **vlab**
- all questions typed in and submitted online (**give**)

Sample exam will be available on the course website in Week 10

## ❖ Supplementary Assessment Policy

Everyone gets **exactly one chance** to pass the Exam

If you attempt the Exam

- I assume that you are fit/healthy enough to take it
- no 2nd chance exams, even with a medical certificate

All Special Consideration requests:

- must **document** how **you** were affected
- must be submitted to UNSW (useful to email lecturer as well)

Supplementary Exams are held ... (maybe) in mid-January!

## ❖ Assessment Summary

Your final mark/grade will be determined as follows:

```
quizzes = mark for on-line quizzes      (out of 15)
ass1    = mark for assignment 1        (out of 15)
ass2    = mark for assignment 2        (out of 20)
exam    = mark for final exam         (out of 50)
okExam  = exam >= 20                  (after scaling)

mark    = ass1 + ass2 + quizzes + exam

grade   = HD|DN|CR|PS    if mark >= 50 && okExam
          = FL           if mark < 50 && okExam
          = UF           if !okExam
```

## ❖ Textbook (options)

- Elmasri, Navathe  
[Fundamentals of Database Systems](#) (7th ed, 2016)
- Garcia-Molina, Ullman, Widom  
[Database Systems: The Complete Book](#) (2nd ed, 2008)
- Ramakrishnan, Gehrke  
[Database Management Systems](#) (3rd ed, 2003)
- Silberschatz, Korth, Sudarshan  
[Database System Concepts](#) (7th ed, 2019)
- Kifer, Bernstein, Lewis  
[Database Systems: Application-Oriented Approach](#) (2nd ed, 2006)

Earlier editions of texts are ok

## ❖ Database Management Systems

Two example DBMSs for prac work:

- **SQLite** (open-source, free, no server needed)
- **PostgreSQL** (open-source, free, full-featured)

Comments on using a specific DBMS:

- the primary goal is to learn SQL (a standard)
- the specific DBMS is not especially important \*\*
- but, each DBMS implements non-standard features
- we will use standard SQL as much as possible
- PG docs describe all deviations from standard

\*\* Unless it seriously violates SQL standards ... I mean you, MySQL

## ❖ Further Reading Material

The on-line documentation and manuals provided with:

- [SQLite](#) are reasonably good
- [PostgreSQL](#) are very good
- [Python](#) are similarly comprehensive

Some comments on technology books:

- tend to be expensive and short-lived
- many provide just the manual, plus some examples
- generally, anything published by O'Reilly is useful

Aside: once you understand the concepts, the manual is sufficient

## ❖ Home Computing

Software versions that we'll be running this semester (TBC):

- PostgreSQL 12/13, SQLite 3.x, Python 3.7+, psycopg2 2.8+

If you install them at home:

- get versions "close to" these
- **test all work at CSE before submitting**

Alternative to installing at home:

- run them on the new AWS CSE server in a terminal window
- use **v1ab** to log in to the AWS server with a window manager

Details on setting up a PostgreSQL server are in the first Prac Exercise.

## ❖ Home Computing (cont)

To access new server via **ssh**:

```
$ ssh d.cse.unsw.edu.au
```

To access via VLab, use a VNC service with:

```
d.cse.unsw.edu.au:5920
```

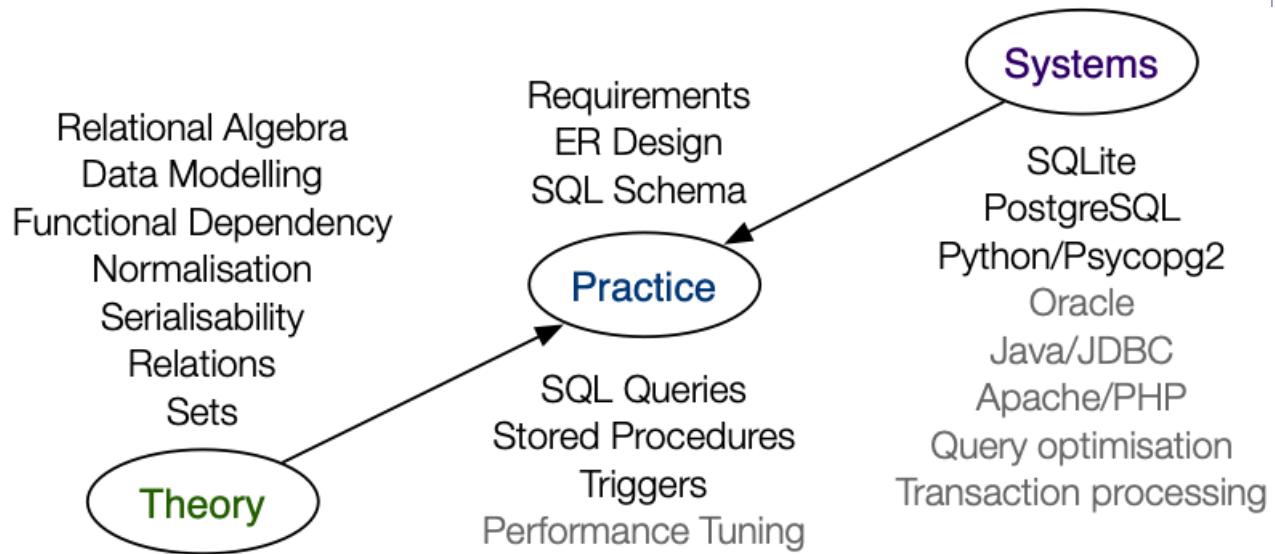
Hostname is **nw-syd-vxdb**

Your CSE home directory is accessible on this server

You have a special directory under **/localstorage**

More details on how to set up PostgreSQL in first Prac Exercise

## ❖ Overview of the Databases Field



## ❖ Database Application Development

A variation on standard software engineering process:

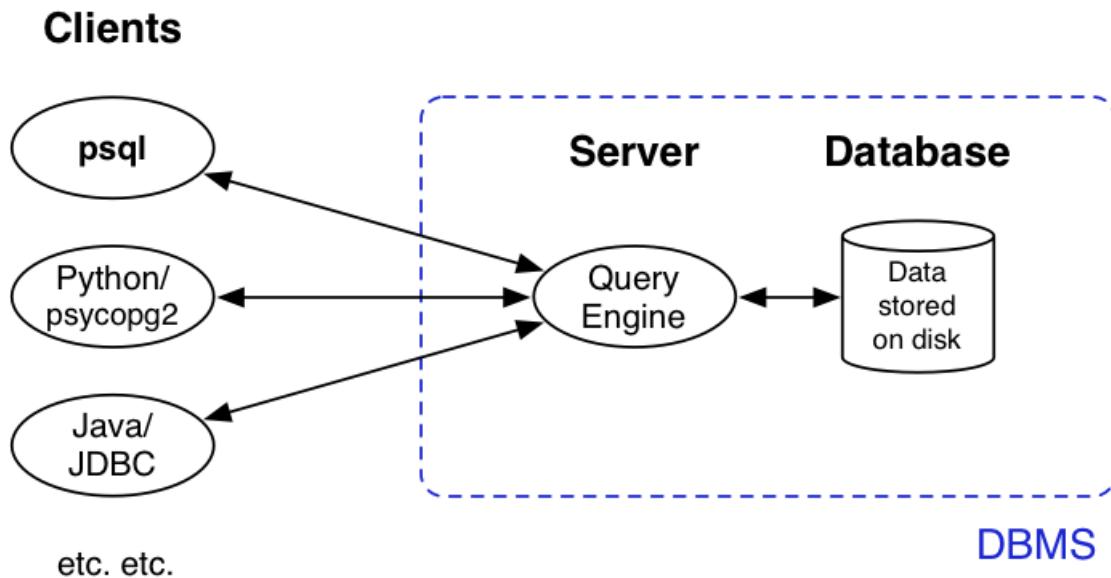
1. analyse application requirements
2. develop a data model to meet these requirements
3. check data model for redundancy (using relational theory)
4. implement the data model as relational schema
5. define operations (transactions) on this model
6. implement operations via SQL and procedural PLs
7. construct a program interface to these operations
8. monitor performance and "tune" the schema/operations

At some point, populate the database (may be via interface)

During the course, we consider these in the order 2, 4, 6, 7, 3

## ❖ Database System Architecture

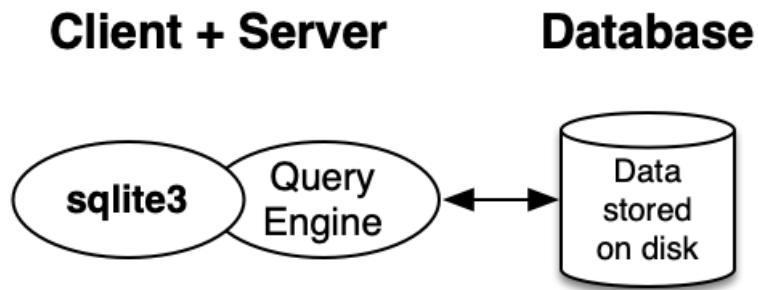
The typical environment for a modern DBMS is:



SQL queries and results travel along the client ↔ server links

## ❖ Database System Architecture (cont)

SQLite is not a client-server system:



Although it does have an API for use from programming languages.



# Data Models

---

- Data Modelling
- Some Design Ideas
- Exercise: GMail Data Model
- Quality of Designs

## ❖ Data Modelling

---

Aims of data modelling:

- describe what **information** is contained in the database  
(e.g., entities: students, courses, accounts, branches, patients, ...)
- describe **relationships** between data items  
(e.g., John is enrolled in COMP3311, Tom's account is held at Coogee)
- describe **constraints** on data  
(e.g., 7-digit IDs, students can enrol in no more than 3 courses per term)

Data modelling is a **design** process

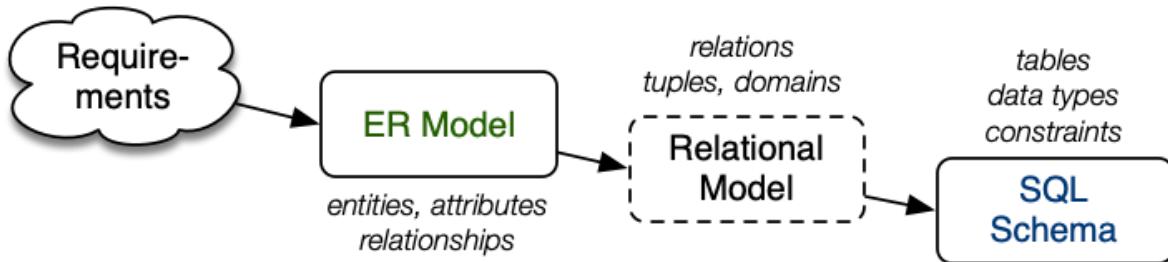
- converts requirements into a data model

## ❖ Data Modelling (cont)

Kinds of data models:

- **logical**: abstract, for conceptual design, e.g., ER, ODL, UML
- **physical**: record-based, for implementation, e.g., relational, SQL

Strategy: design using abstract model; map to physical model



## ❖ Some Design Ideas

Consider the following while working through exercises:

- start simple ... evolve design as problem better understood
- identify objects (and their properties), then relationships
- most designs involve kinds (classes) of people
- keywords in requirements suggest data/relationships  
(rule-of-thumb: nouns → data, verbs → relationships)
- don't confuse operations with relationships  
(operation: he **buys** a book; relationship: the book **is owned** by him)
- consider all possible data, not just what is available

## ❖ Exercise: GMail Data Model

Consider the GMail system (or any other modern mail client)

Develop an informal data model for it by identifying:

- the data items involved (objects and their attributes)
- relationships between these data items
- constraints on the data and relationships

## ❖ Exercise: GMail Data Model (cont)

Objects in GMail data model:

```

users
    gmail-address, name, password, ...

messages
    timestamp, sender*, title, content, ...

tags
    owner, name, colour parent*

settings
    name, value, user*
  
```

Relationships in GMail data model:

```

recipients
    user - message

sent
    user - message

tag-hierarchy
    child-tag - parent-tag

settings
    user - setting
  
```

Constraints in GMail data model:

```

gmail-address values are unique

users must have a password (strong?)

every message has a sender

every message has a non-empty title and content
  
```

values for each setting are valid for that setting

## ❖ Quality of Designs

There is no single "best" design for a given application.

Most important aspects of a design (data model):

- correctness (satisfies requirements accurately)
- completeness (all reqs covered, all assumptions explicit)
- consistency (no contradictory statements)

Potential **inadequacies** in a design:

- omits information that needs to be included
- contains redundant information ( $\Rightarrow$  inconsistency)
- leads to an inefficient implementation
- violates syntactic or semantic rules of data model



# ER Model

---

- Entity-Relationship Data Modelling
- Entity-Relationship (ER) Diagrams
- Entity Sets
- Keys
- Example: Identifying Keys
- Relationship Sets
- Example: Relationship Semantics
- Weak Entity Sets
- Subclasses and Inheritance
- Design Using the ER Model
- Large ER Diagrams
- Summary of ER

## ❖ Entity-Relationship Data Modelling

The world is viewed as a collection of **inter-related entities**.

ER has three major modelling constructs:

- **attribute**: **data item** describing a property of interest
- **entity**: collection of attributes describing **object** of interest
- **relationship**: **association** between entities (objects)

The ER model is not a standard, so notational variations exist

Lecture notes use notation from SKS and GUW books (simple)

## ❖ Entity-Relationship (ER) Diagrams

ER diagrams are a graphical tool for data modelling.

An ER diagram consists of:

- a collection of entity set definitions
- a collection of relationship set definitions
- attributes associated with entity and relationship sets
- connections between entity and relationship sets

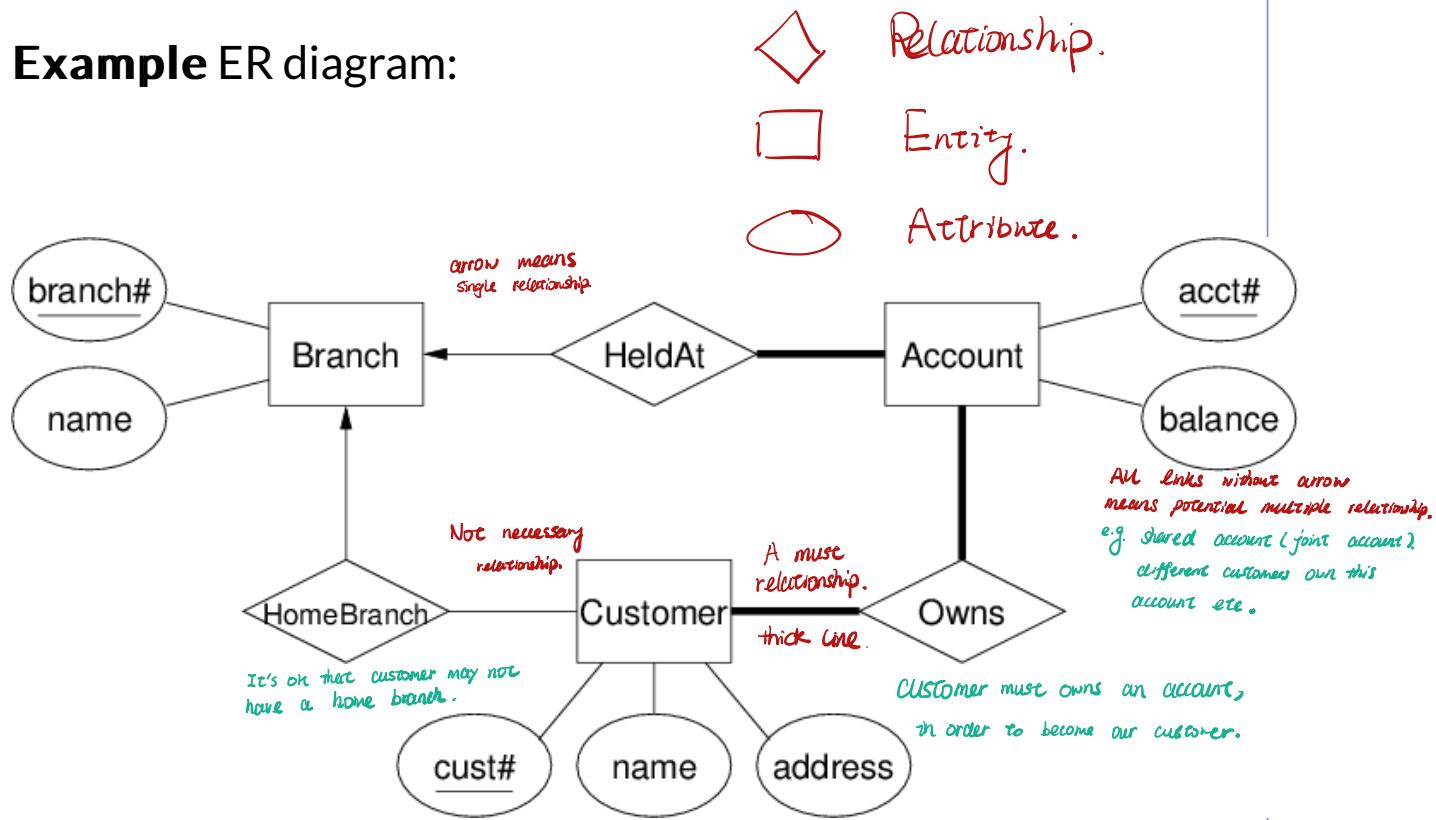
Terminology abuse:

- we say "entity" when we mean "entity set"
- we say "relationship" when we mean "relationship sets"
- we say "entity instance" to refer to a particular entity

# ❖ Entity-Relationship (ER) Diagrams

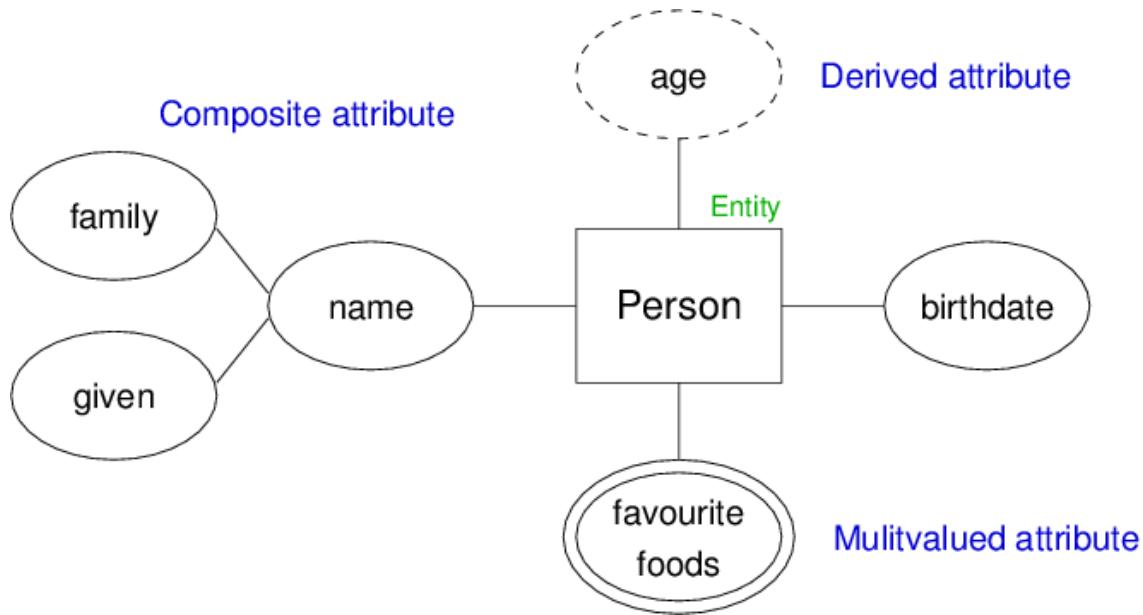
(cont)

**Example ER diagram:**



## ❖ Entity-Relationship (ER) Diagrams (cont)

**Example** of attribute notations:



## ❖ Entity Sets

An **entity set** can be viewed as either:

- a set of entities with the same set of attributes (extensional)
- an abstract description of a class of entities (intensional)

**Key (superkey)**: any set of attributes 独一无二的验证码.

- whose set of values are distinct over entity set
- natural (e.g., name+address+birthday) or artificial (e.g., SSN)

**Candidate key** = minimal superkey (no subset is a key) *which is unique, and can be used to distinguish certain person.*  
*eg { student ID } { DOB + Address }.*

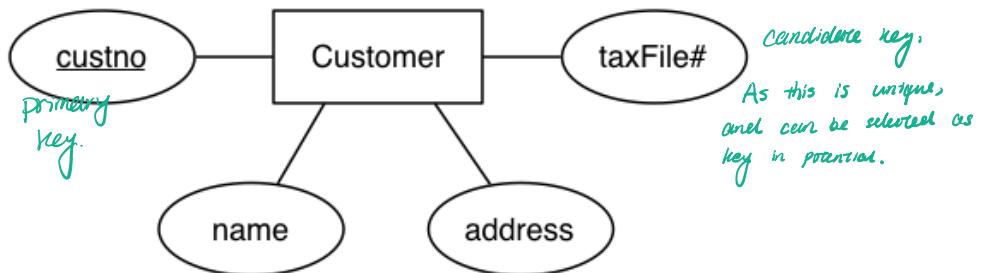
**Primary key** = candidate key chosen by DB designer

Keys are indicated in ER diagrams by underlining

## ❖ Keys

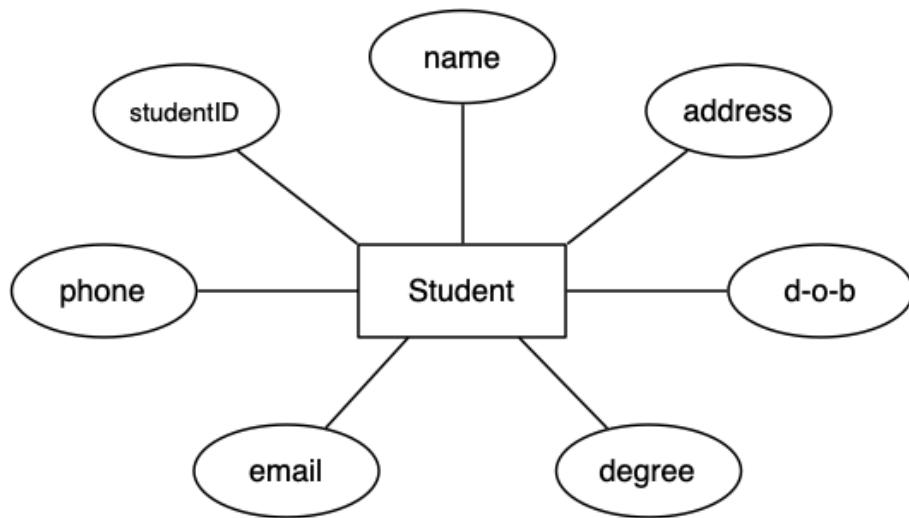
Sometimes primary keys are obvious ...

*Primary key: The attribute which can distinguish the object.*



## ❖ Example: Identifying Keys

Candidate keys in the following ER diagram ...



Possibilities: {studentID}, {phone}, {email},  
{name,address,d-o-b}?

## ❖ Relationship Sets

**Relationship:** an association among several entities

- e.g., Customer(9876) **is the owner of** Account(12345)

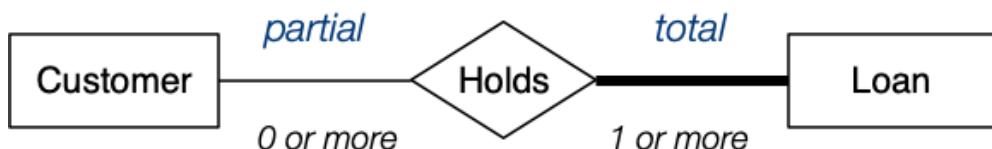
**Relationship set:** collection of relationships of the same type

**Degree** = # entities involved in reln (in ER model,  $\geq 2$ )

**Cardinality** = # associated entities on each side of reln

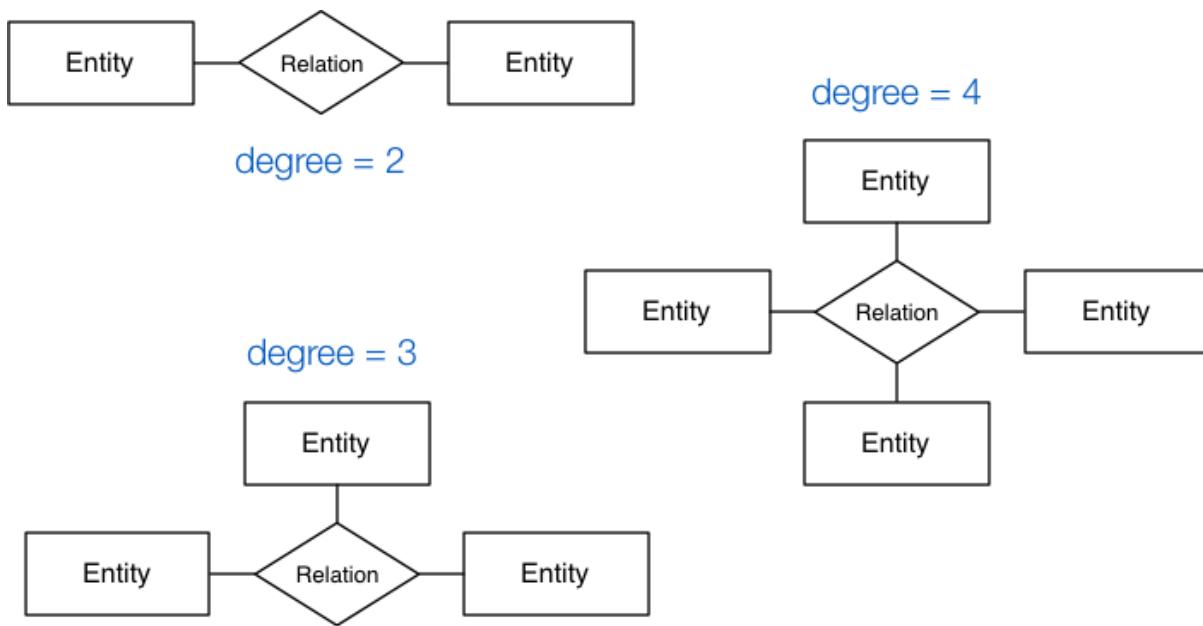
**Participation** = must every entity be in the relationship

**Example:** relationship participation



## ❖ Relationship Sets (cont)

**Examples:** relationship degree



## ❖ Relationship Sets (cont)

**Examples:** relationship cardinality

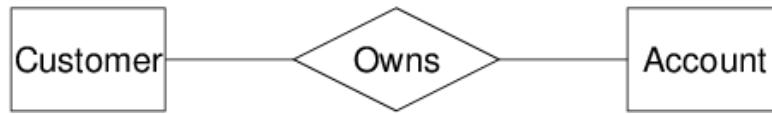
one-to-one



one-to-many



many-to-many



## ❖ Example: Relationship Semantics

Semantics of the following relationships ...



*A lecturer may teach one course  
Every course is taught by 1 or more lecturers*



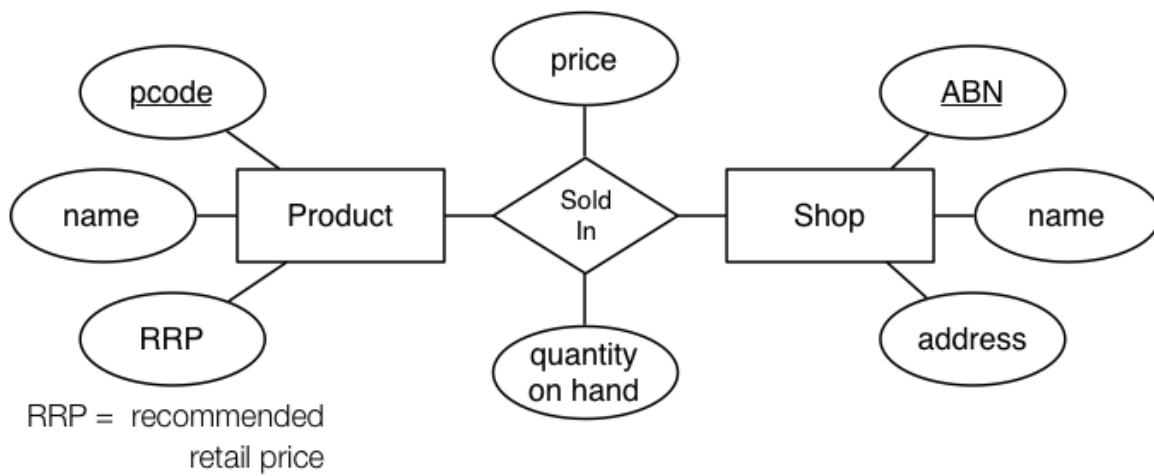
*Every lecturer teaches one or more courses  
Every course is taught by 1 or more lecturers*



*Every lecturer may teach one course  
Every course is taught by one lecturer*

## ❖ Example: Relationship Semantics (cont)

In some cases, a relationship needs associated attributes.



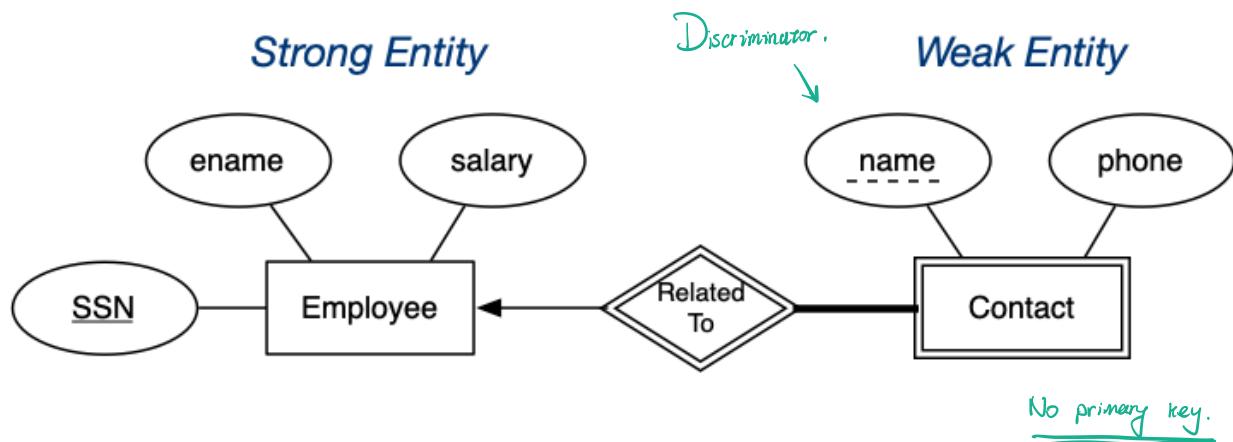
(Price and quantity are related to products in a particular shop)

## ❖ Weak Entity Sets

Weak entities

- exist only because of association with strong entities.
- have no key of their own; have a **discriminator**

Example:



## ❖ Subclasses and Inheritance

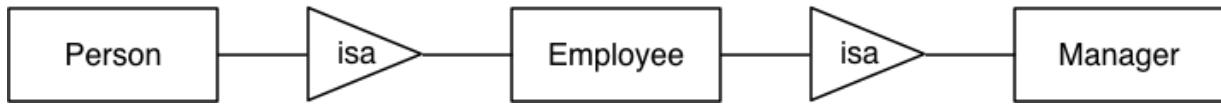
A **subclass** of an entity set  $A$  is a set of entities:

- with all attributes of  $A$ , plus (usually) its own attributes
- that is involved in all of  $A$ 's relationships, plus its own

Properties of subclasses:

- **overlapping** or **disjoint** (can an entity be in multiple subclasses?)
- **total** or **partial** (does every entity have to also be in a subclass?)

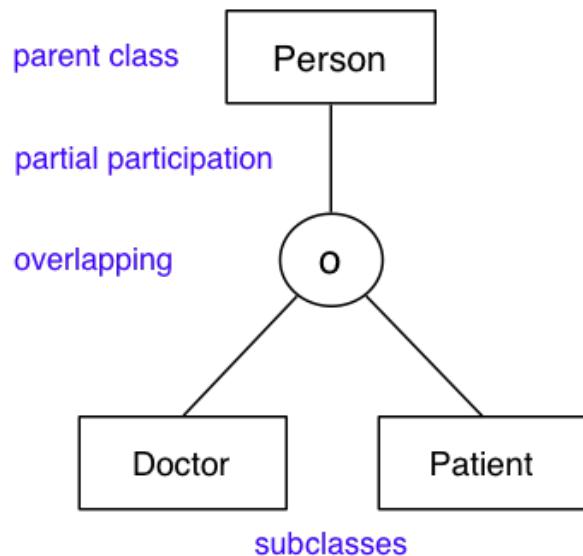
Special case: entity has one subclass ("B **is-a** A" specialisation)



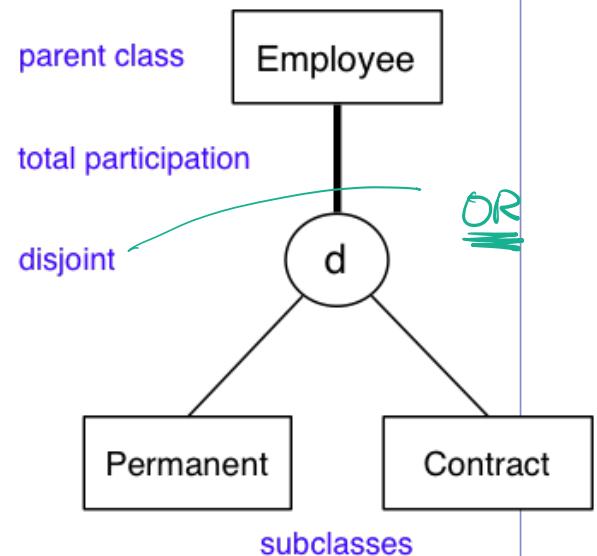
## ❖ Subclasses and Inheritance (cont)

Example:

*A person may be a doctor and/or may be a patient or may be neither*



*Every employee is either a permanent employee or works under a contract*



## ❖ Design Using the ER Model

ER model: simple, powerful set of data modelling tools

Some considerations in designing ER models:

- should an "object" be represented by an attribute or entity?
- is a "concept" best expressed as an entity or relationship?
- should we use  $n$ -way relationship or several 2-way relationships?
- is an "object" a strong or weak entity? (usually strong)
- are there subclasses/superclasses within the entities?

Answers to above are worked out by *thinking* about the application domain.

## ❖ Large ER Diagrams

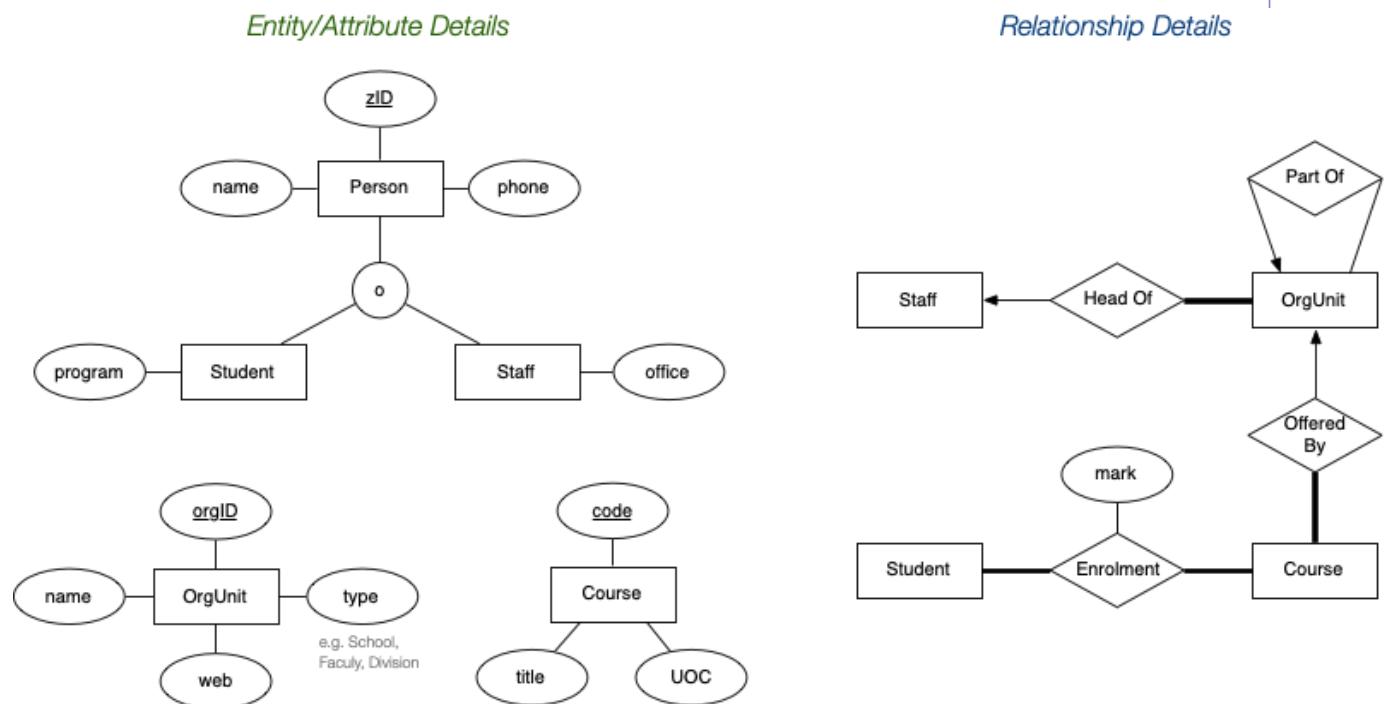
ER diagrams are typically too large to fit on a single screen  
(or a single sheet of paper, if printing)

One commonly used strategy:

- define entity sets separately, showing attributes
- combine entities and relationships on a single diagram  
(but without showing entity attributes)
- if very large design, may use several linked diagrams

## ❖ Large ER Diagrams (cont)

Example of drawing large ER diagram:



## ❖ Summary of ER

ER model is popular for doing conceptual design

- high-level, models relatively easy to understand
- good expressive power, can capture many details

Basic constructs: entities, relationships, attributes

Relationship constraints: total / partial, n:m / 1:n / 1:1

Other constructs: inheritance hierarchies, weak entities

Many notational variants of ER exist  
(especially in the expression of constraints on relationships)

---

Produced: 13 Sep 2020

# Relational Model

---

- Relational Data Model
- Example Database Schema
- Example Database (Instance)
- Integrity Constraints
- Referential Integrity
- Relational Databases
- Describing Relational Schemas

## ❖ Relational Data Model

The **relational data model** describes the world as

- a collection of inter-connected **relations** (or **tables**)

The relational model has one structuring mechanism:  
**relations**

- relations are used to model both entities and relationships

Each **relation** (denoted  $R, S, T, \dots$ ) has:

- a **name** (unique within a given database)
- a set of **attributes** (which can be viewed as column headings)

Each **attribute** (denoted  $A, B, \dots$  or  $a_1, a_2, \dots$ ) has:

- a **name** (unique within a given relation)
- an associated **domain** (set of allowed values)

## ❖ Relational Data Model (cont)

Consider relation  $R$  with attributes  $a_1, a_2, \dots, a_n$

Relation schema of  $R$ :  $R(a_1:D_1, a_2:D_2, \dots, a_n:D_n)$

Tuple of  $R$ : an element of  $D_1 \times D_2 \times \dots \times D_n$  (i.e. list of values)

Instance of  $R$ : subset of  $D_1 \times D_2 \times \dots \times D_n$  (i.e. set of tuples)

Note: tuples:  $(2,3) \neq (3,2)$  relation:  $\{ (a,b), (c,d) \} = \{ (c,d), (a,b) \}$

Domains are comprised of atomic values (e.g. integer, string, date)

A distinguished value **NULL** belongs to all domains

Each relation has a **key** (subset of attributes unique for each tuple)

## ❖ Relational Data Model (cont)

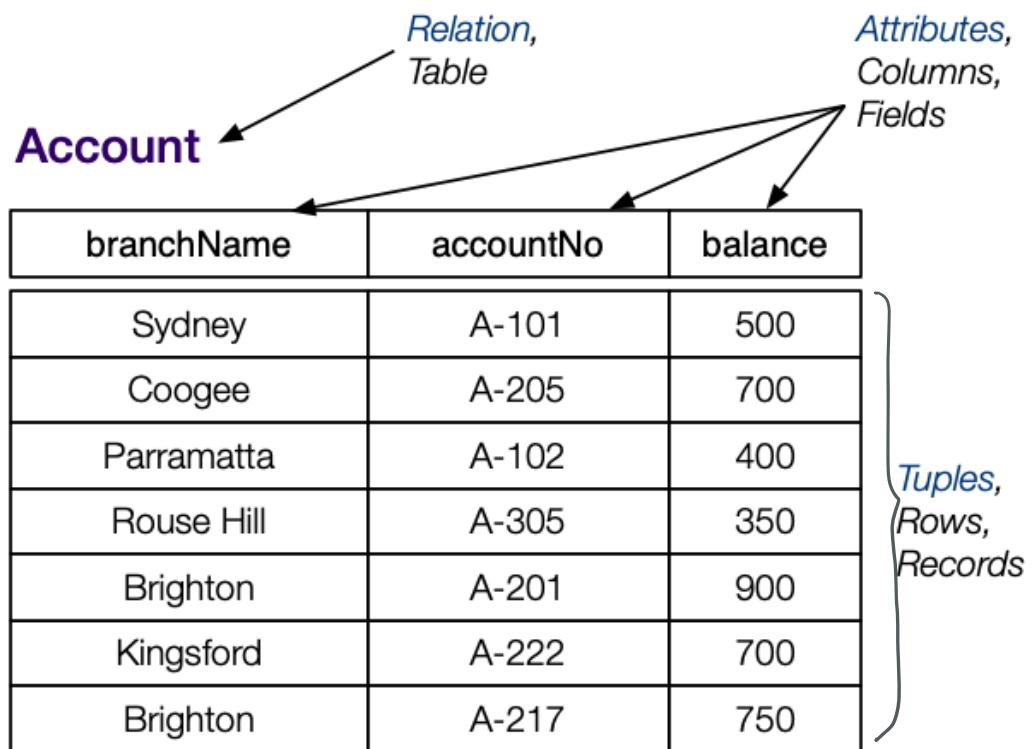
A relation: **Account**(**branchName**, **accountNo**, **balance**)

And an *instance* of this relation:

```
{  
    (Sydney, A-101, 500),  
    (Coogee, A-215, 700),  
    (Parramatta, A-102, 400),  
    (Rouse Hill, A-305, 350),  
    (Brighton, A-201, 900),  
    (Kingsford, A-222, 700)  
    (Brighton, A-217, 750)  
}
```

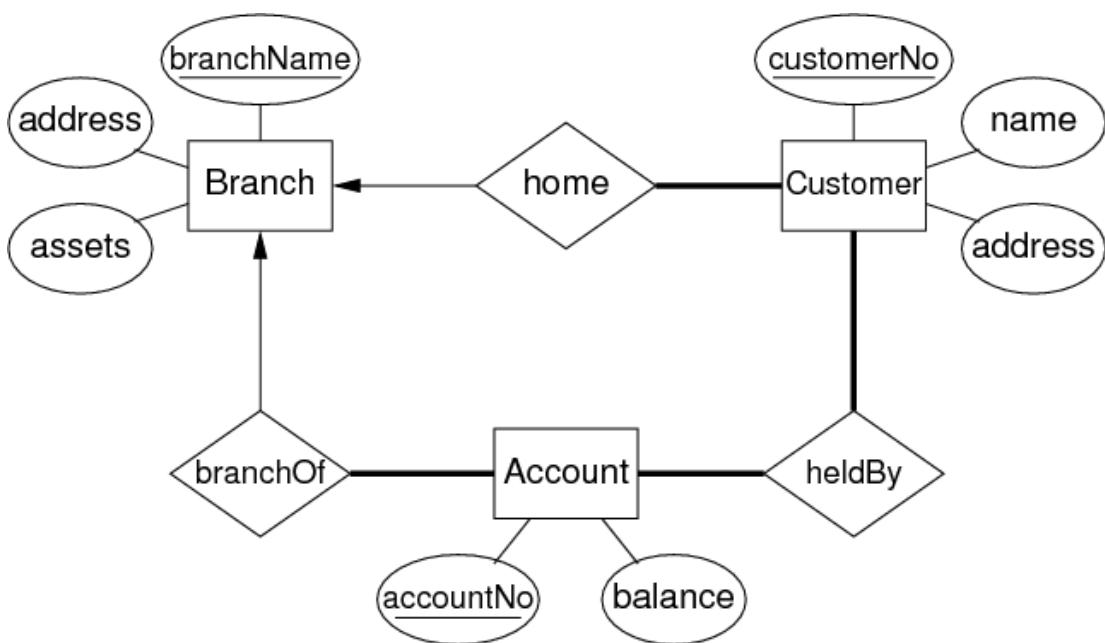
## ❖ Relational Data Model (cont)

**Account** relation as a table:



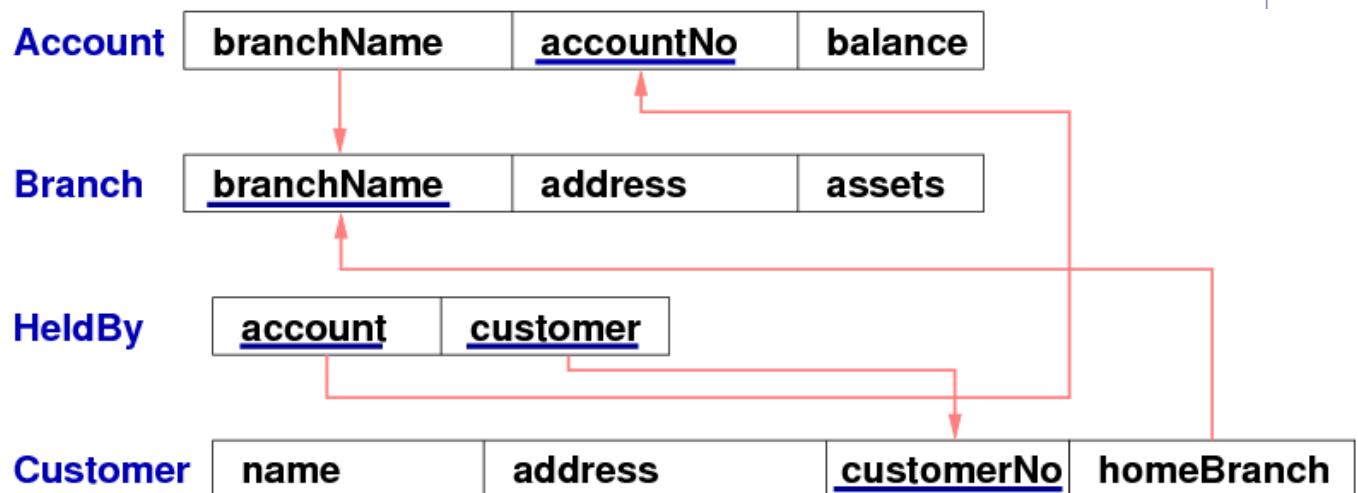
## ❖ Example Database Schema

Consider the following ER data model:



## ❖ Example Database Schema (cont)

Relational schema derived from this ER model:



Note: distinguish attributes via e.g. **Branch.address** vs  
**Customer.address**

# ❖ Example Database (Instance)

## Account

branchName	accountNo	balance
Sydney	A-101	500
Coogee	A-205	700
Parramatta	A-102	400
Rouse Hill	A-305	350

...

## Branch

branchName	address	assets
Sydney	Pitt St	9000000
Coogee	Coogee Bay Rd	750000
Parramatta	Church St	888000

...

## Customer

name	address	custNo	homeBranch
John Smith	Liverpool	11234	Sydney
Wei Wang	Randwick	74665	Coogee
Arun Shah	Liverpool	99987	Parramatta
Dave Dobbin	Penrith	35012	Rouse Hill

...

## HeldBy

account	customer
A-101	11234
A-205	74665
A-102	99987
A-999	11234

...

## ❖ Integrity Constraints

To represent real-world problems, need to describe

- what values are/are not allowed
- what combinations of values are/are not allowed

Constraints are logical statements that do this:

- **domain constraints:**  
limit the set of values that attributes can take
- **key constraints:**  
identify attributes that uniquely identify tuples
- **entity integrity constraints:** require keys to be fully-defined
- **referential integrity constraints:**  
require references to other tables to be valid

## ❖ Integrity Constraints (cont)

Domain constraints example:

- **Employee.age** attribute is typically defined as **integer**
- better modelled by adding extra constraint **(15<age<66)**

Note: **NULL** satisfies all domain constraints (except (NOT NULL))

Key constraints example:

- **Student(id, ...)** is guaranteed unique
- **Class(...,day,time,location,...)** is unique

Entity integrity example:

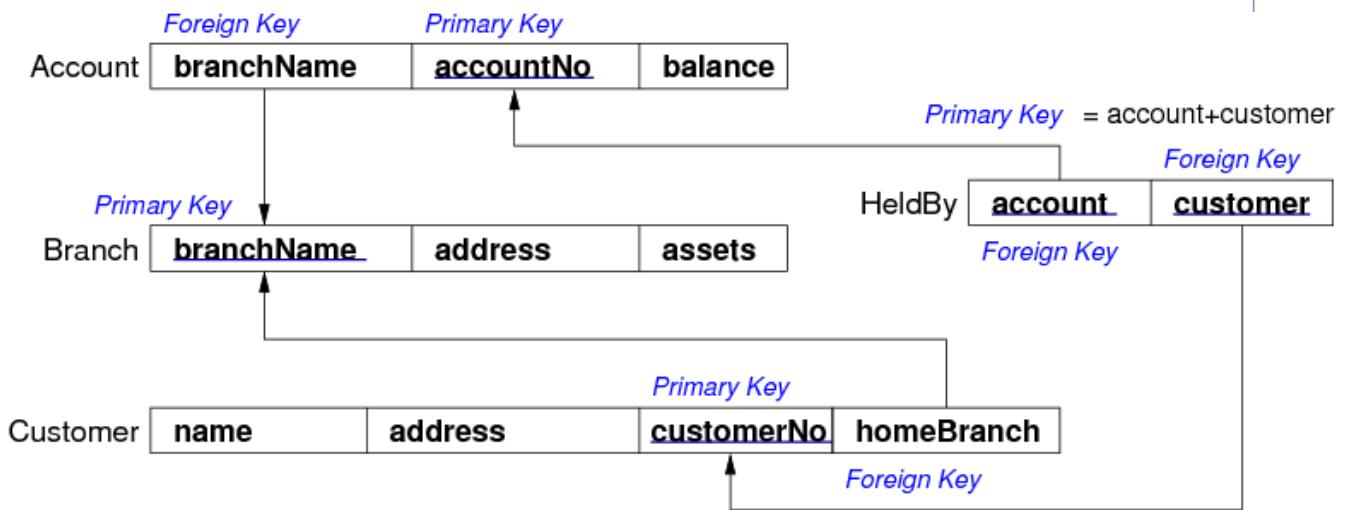
- **Class(...,Mon,2pm,Lyre,...)** is well-defined
- **Class(...,NULL,2pm,Lyre,...)** is not well-defined

# ❖ Referential Integrity

## Referential integrity constraints

- describe references between relations (tables)
- are related to notion of a **foreign key** (FK)

Example:



## ❖ Referential Integrity (cont)

A set of attributes  $F$  in relation  $R_1$  is a **foreign key** for  $R_2$  if:

- the attributes in  $F$  correspond to the primary key of  $R_2$
- the value for  $F$  in each tuple of  $R_1$ 
  - either occurs as a primary key in  $R_2$
  - or is entirely **NULL**

Foreign keys are critical in relational DBs; they provide ...

- the "glue" that links individual relations (tables)
- the way to assemble query answers from multiple tables
- the relational representation of ER relationships

## ❖ Relational Databases

A relational database schema is

- a set of relation schemas  $\{R_1, R_2, \dots, R_n\}$ , and
- a set of integrity constraints

What's difference?  
How these gonna used?

A relational database instance is

- a set of relation instances  $\{r_1(R_1), r_2(R_2), \dots, r_n(R_n)\}$
- where all of the integrity constraints are satisfied

One of the important functions of a relational DBMS:

- ensure that all data in the database satisfies constraints

Changes to the data fail if they would cause constraint violation

## ❖ Describing Relational Schemas

We need a language to express relational schemas  
(which is more detailed than boxes-and-arrows diagrams used above)

SQL provides a **Data Definition Language (DDL)** for this.

```
CREATE TABLE TableName (
    attrName1 domain1 constraints1 ,
    attrName2 domain2 constraints2 ,
    ...
    PRIMARY KEY (attri, attrj, ...),
    FOREIGN KEY (attrx, attry, ...)
        REFERENCES
        OtherTable (attrm, attrn, ...),
    ...
);
```

To be continued ...

---

Produced: 14 Sep 2020

# ER→Relational Mapping

---

- ER to Relational Mapping
- Relational Model vs ER Model
- Mapping Strong Entities
- Mapping Weak Entities
- Mapping N:M Relationships
- Mapping 1:N Relationships
- Mapping 1:1 Relationships
- Mapping n-way Relationships
- Mapping Composite Attributes
- Mapping Multi-valued Attributes (MVAs)
- Mapping Subclasses

## ❖ ER to Relational Mapping

Reminder: a useful strategy for database design:

- perform initial data modelling using ER  
*(conceptual-level modelling)*
- transform conceptual design into relational model  
*(implementation-level modelling)*

A formal mapping exists for ER model → Relational model.

This maps "structures"; but additional info is needed, e.g.

- concrete domains for attributes and other constraints

## ❖ Relational Model vs ER Model

Correspondences between relational and ER data models:

- $\text{attribute(ER)} \approx \text{attribute(Rel)}$ ,  $\text{entity(ER)} \approx \text{tuple(Rel)}$
- $\text{entity set(ER)} \approx \text{relation(Rel)}$ ,  $\text{relationship(ER)} \approx \text{relation(Rel)}$

Differences between relational and ER models:

- Rel uses relations to model entities *and* relationships
- Rel has no composite or multi-valued attributes (only atomic)
- Rel has no object-oriented notions (e.g. subclasses, inheritance)

Note that ...

- not all aspects of ER can be represented exactly in a relational schema
- some aspects of relational schemas (e.g. domains) do not appear in ER

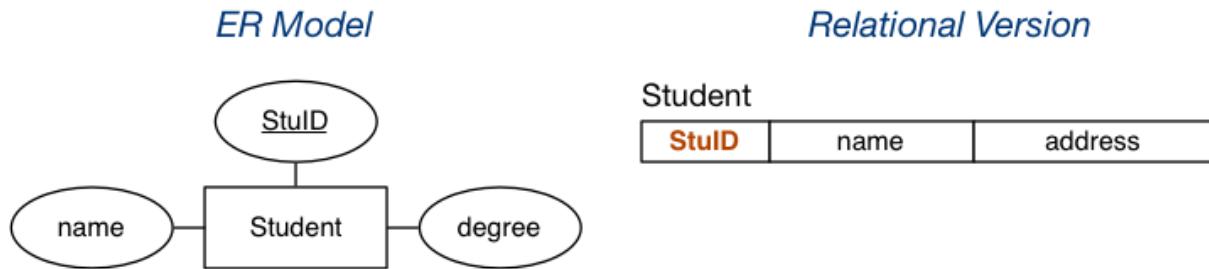
## ❖ Mapping Strong Entities

An entity set  $E$  with atomic attributes  $a_1, a_2, \dots, a_n$

maps to

A relation  $R$  with attributes (columns)  $a_1, a_2, \dots, a_n$

Example:

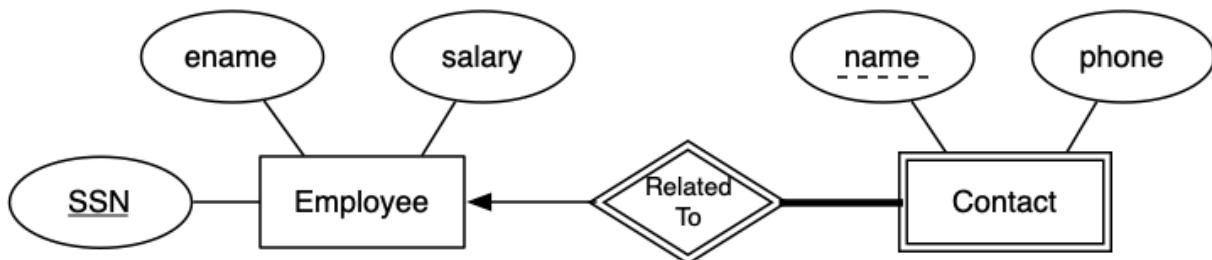


Note: the key is preserved in the mapping.

## ❖ Mapping Weak Entities

Example:

*ER Model*



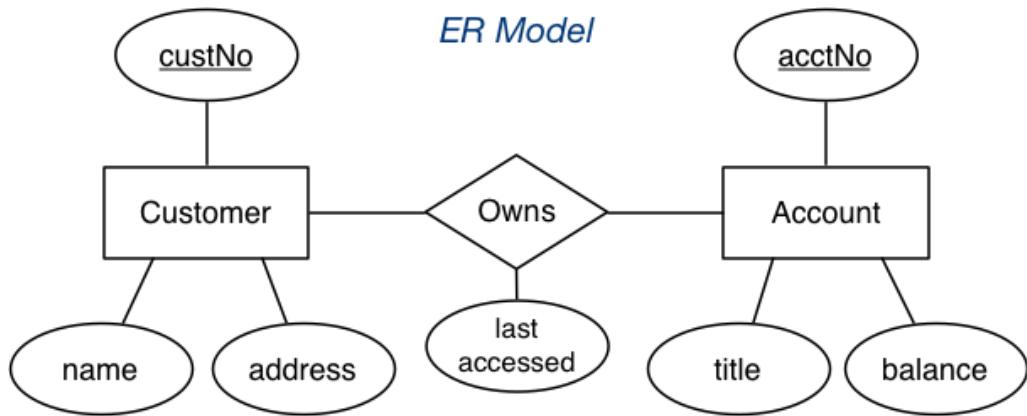
*Relational Version*

Employee	<b>SSN</b>	ename	salary
----------	------------	-------	--------

Contact	<b>SSN</b>	name	phone
---------	------------	------	-------

## ❖ Mapping N:M Relationships

Example:

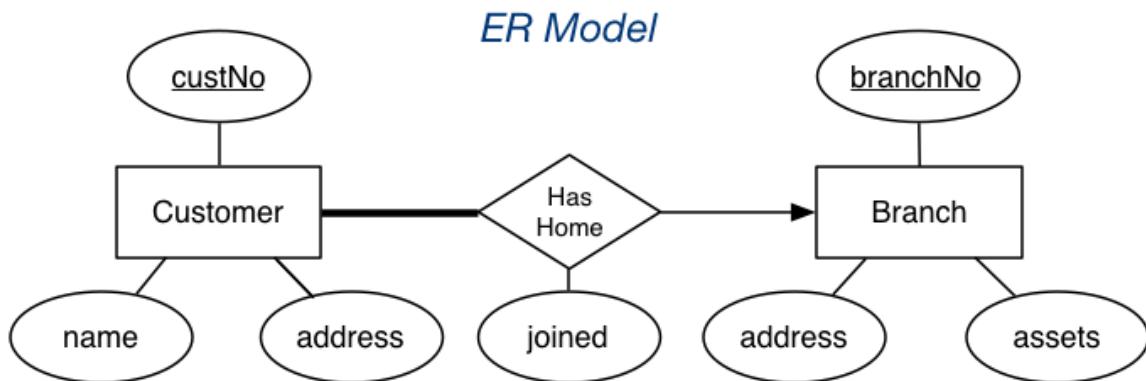


*Relational Version*

Customer	<b>custNo</b>	name	address
Account	<b>acctNo</b>	title	balance
Owns	<b>acctNo</b>	<b>custNo</b>	lastAccessed

## ❖ Mapping 1:N Relationships

Example:



*Relational Version*

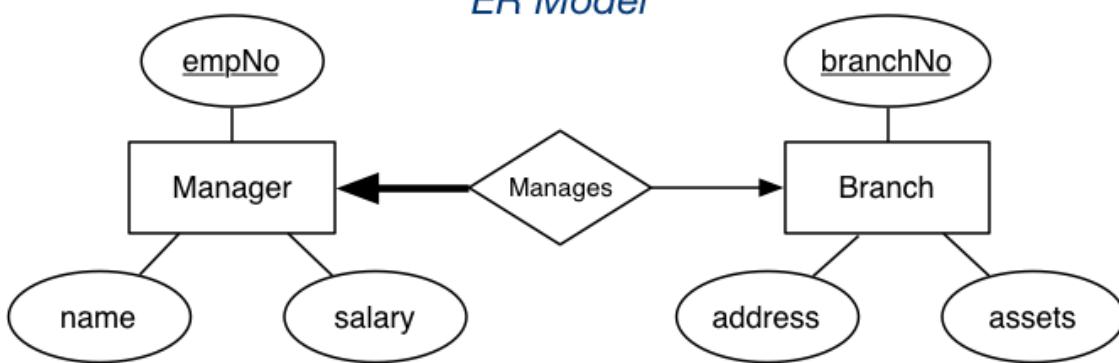
Customer	<b>custNo</b>	name	address	branchNo	joined
----------	---------------	------	---------	----------	--------

Branch	<b>branchNo</b>	address	assets
--------	-----------------	---------	--------

## ❖ Mapping 1:1 Relationships

Example:

*ER Model*

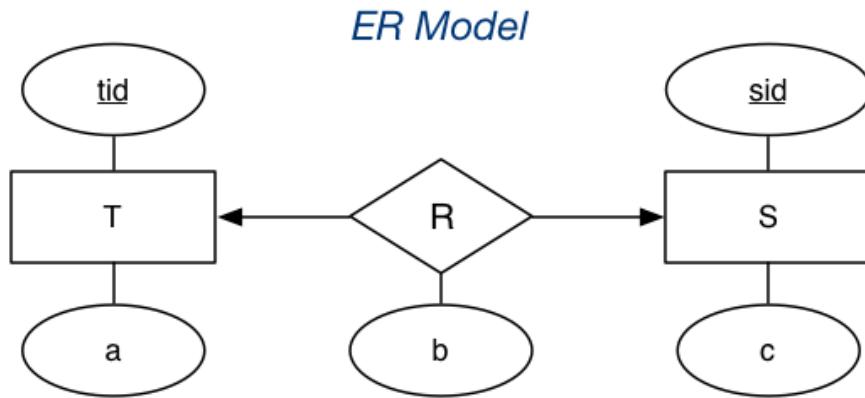


*Relational Version*

Manager	empNo	name	salary	branchNo
Branch	branchNo	address	assets	

## ❖ Mapping 1:1 Relationships (cont)

If there is no reason to favour one side of the relationship ...



*Relational Version #1*

T	<b>tid</b>	a	<b>sid</b>	b
---	------------	---	------------	---

S	<b>sid</b>	c
---	------------	---

*Relational Version #2*

T	<b>tid</b>	a
---	------------	---

S	<b>sid</b>	c	<b>tid</b>	b
---	------------	---	------------	---

## ❖ Mapping n-way Relationships

Relationship mappings above assume binary relationship.

If multiple entities are involved:

- $n:m$  generalises naturally to  $n:m:p:q$ 
  - include foreign key for each participating entity
  - include any other attributes of the relationship
- other multiplicities (e.g.  $1:n:m$ ) ...
  - need to be mapped the same as  $n:m:p:q$
  - so not quite an accurate mapping of the ER

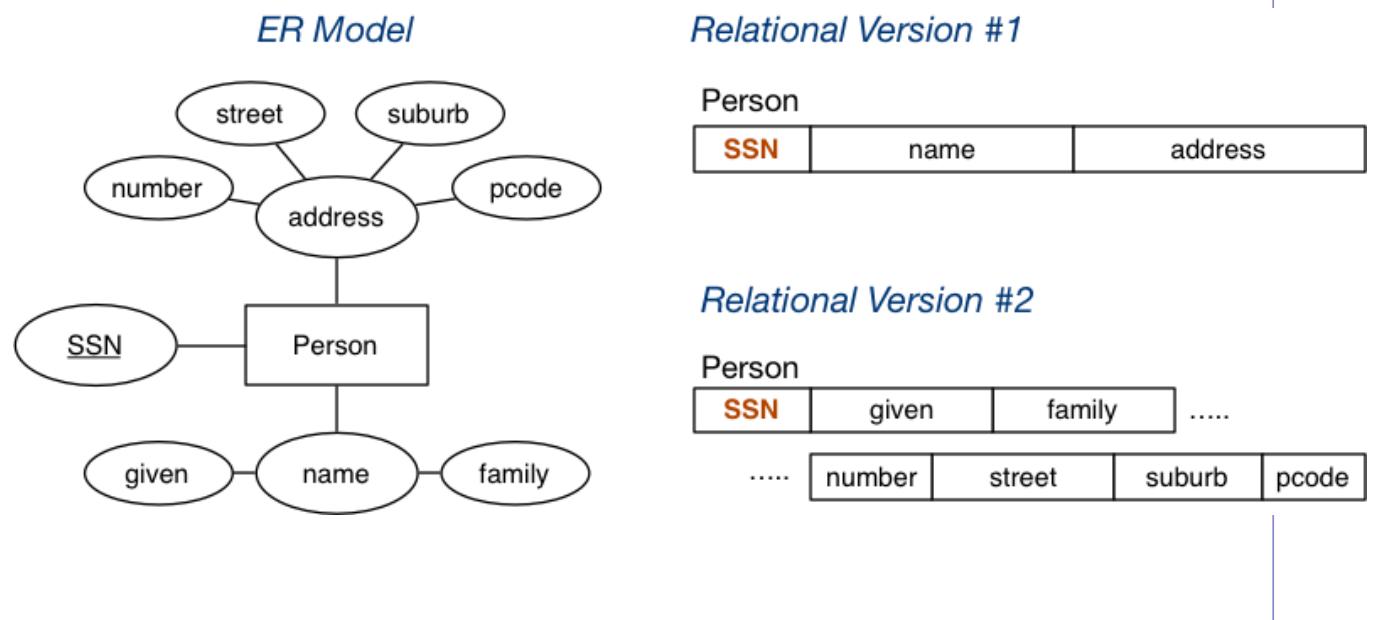
Some people advocate converting n-way relationships into:

- a new entity, and a set of  $n$  binary relationships

## ❖ Mapping Composite Attributes

Composite attributes are mapped by concatenation or flattening.

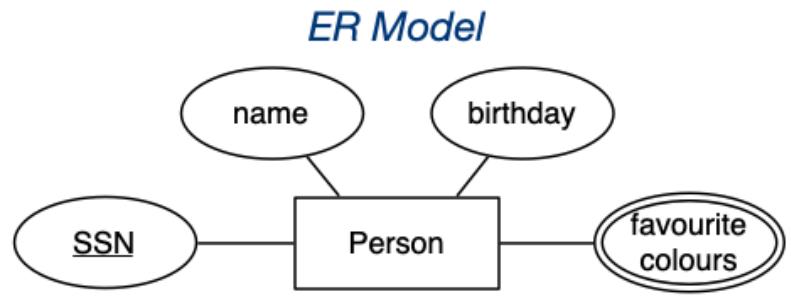
Example:



## ❖ Mapping Multi-valued Attributes (MVAs)

MVAs are mapped by a new table linking values to their entity.

Example:

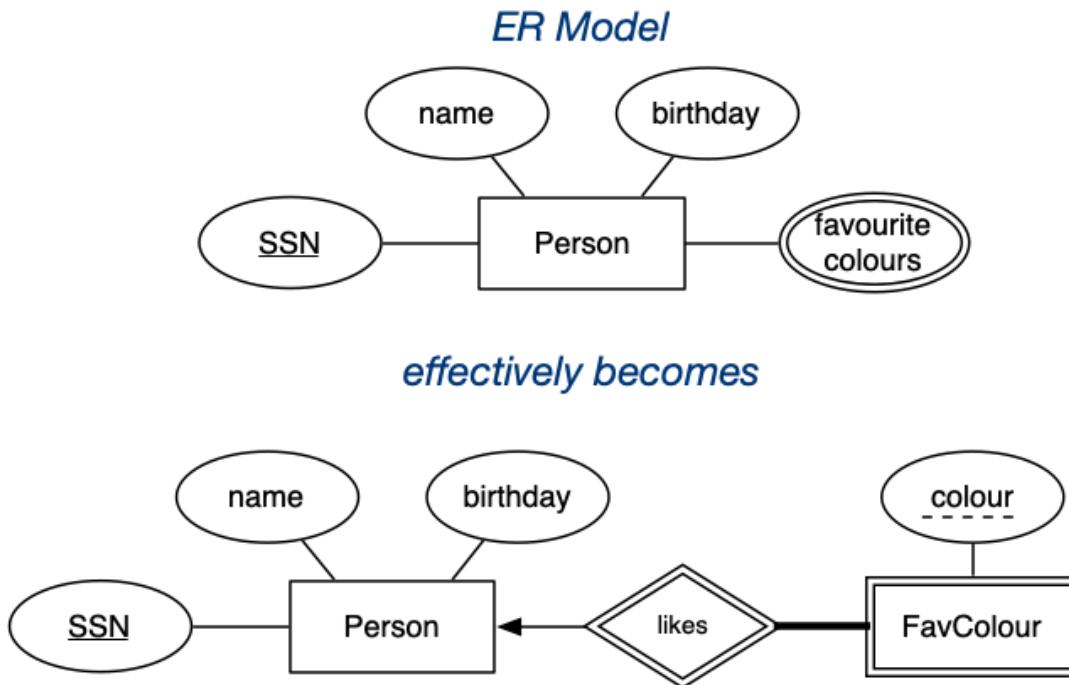


*Relational Version*

Person	SSN	name	birthday
FavColour	SSN	colour	

## ❖ Mapping Multi-valued Attributes (MVAs) (cont)

Analogy:



## ❖ Mapping Multi-valued Attributes (MVAs) (cont)

Example: the two entities

```
Person(12345, John, 12-feb-1990, [red,green,blue])
Person(54321, Jane, 25-dec-1990, [green,purple])
```

would be represented as

```
Person(12345, John, 12-feb-1990)
Person(54321, Jane, 25-dec-1990)
FavColour(12345, red)
FavColour(12345, green)
FavColour(12345, blue)
FavColour(54321, green)
FavColour(54321, purple)
```

## ❖ Mapping Subclasses

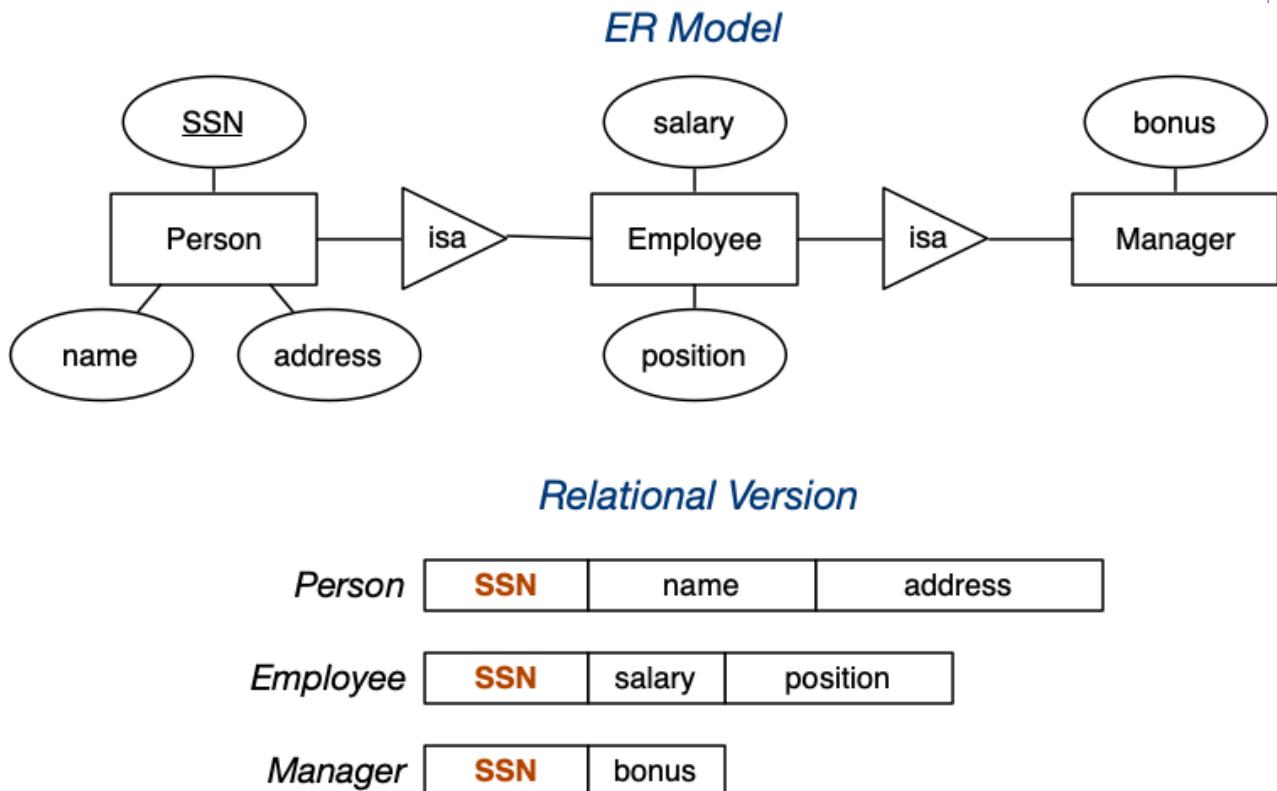
Three different approaches to mapping subclasses to tables:

- ER style
  - each entity becomes a separate table,
  - containing attributes of subclass + FK to superclass table
- object-oriented
  - each entity becomes a separate table,
  - inheriting all attributes from all superclasses
- single table with nulls
  - whole class hierarchy becomes one table,
  - containing all attributes of all subclasses (null, if unused)

Which mapping is best depends on how data is to be used.

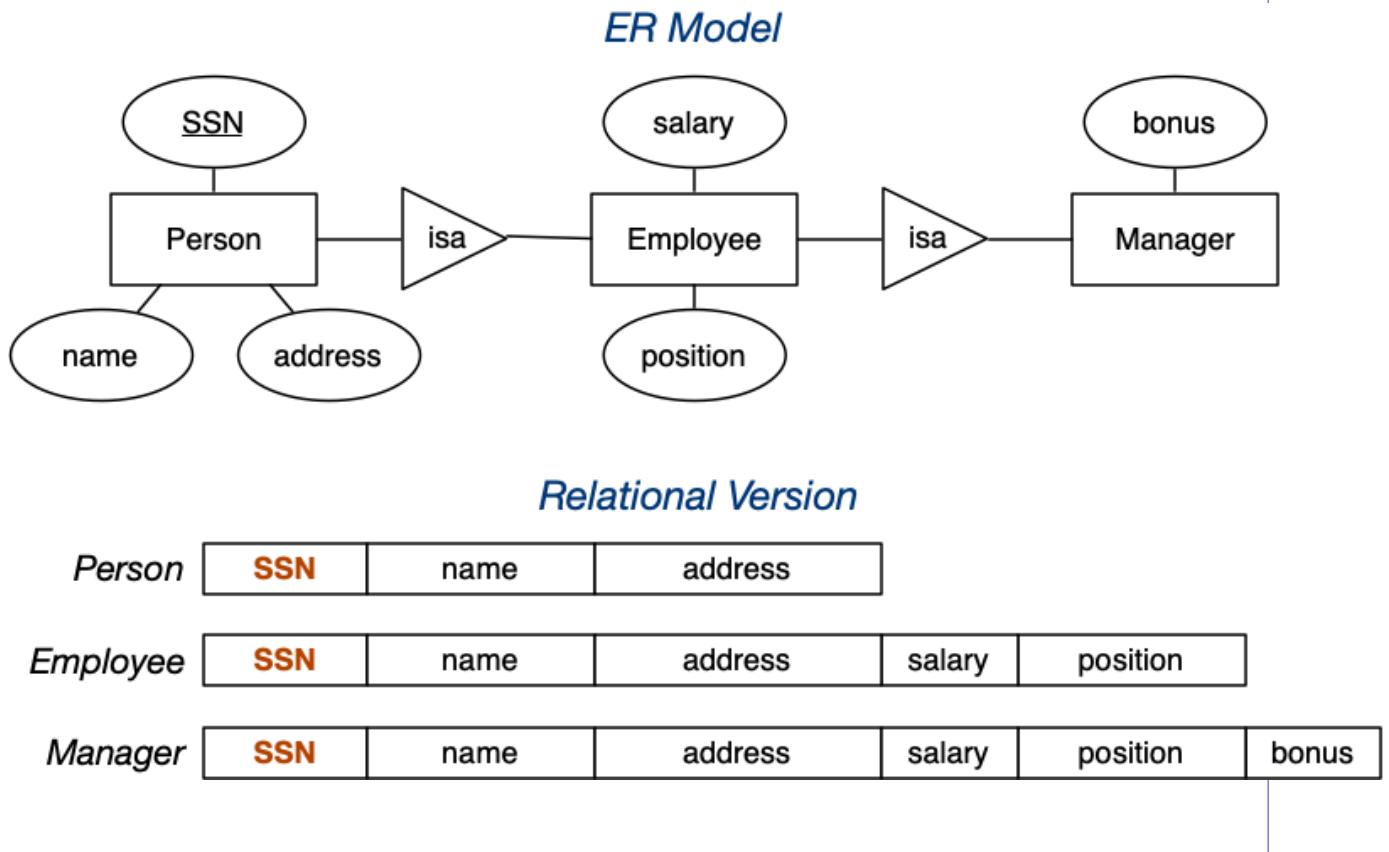
## ❖ Mapping Subclasses (cont)

Example of ER-style mapping:



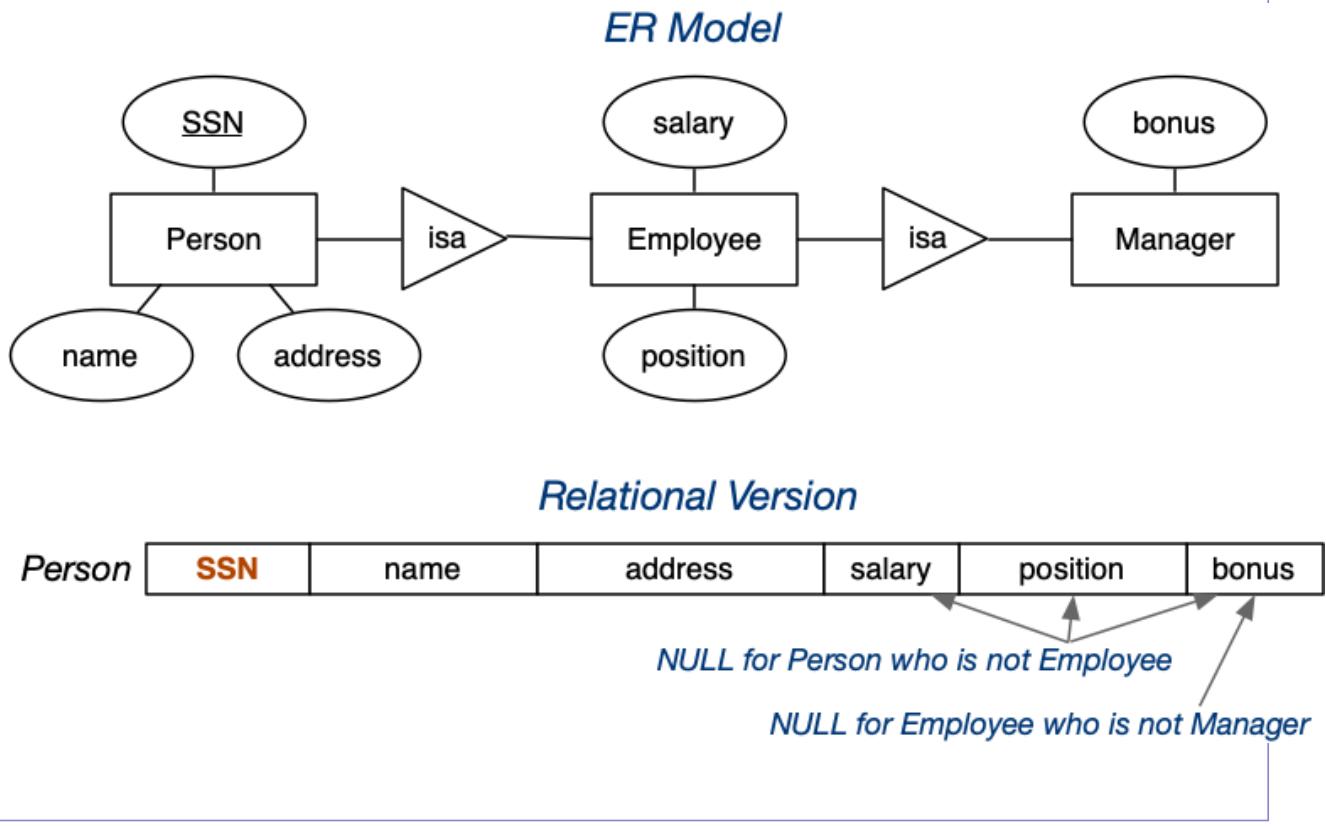
## ❖ Mapping Subclasses (cont)

Example of object-oriented mapping:



## ❖ Mapping Subclasses (cont)

Example of single-table-with-nulls mapping:



---

Produced: 15 Sep 2020

# SQL Introduction

---

- SQL vs Relational Model
- SQL History
- SQL Intro
- SQL Syntax in a Nutshell
- Names in SQL
- Types/Constants in SQL
- Examples of Defining Domains/Types
- Tuple and Set Literals

## ❖ SQL vs Relational Model

The **relational model** is a formal system for

- describing data (relations, tuples, attributes, domains, constraints)
- manipulating data (relational algebra ... covered elsewhere)

**SQL** is a "programming" language for

- describing data (tables, rows, fields, types, constraints)
- manipulating data (query language)

SQL extends the relational model in some ways (e.g bags vs sets of tuples)

SQL omits some aspects of the relational model (e.g. general constraints)

## ❖ SQL History

Developed at IBM in the mid-1970's (System-R)

Standardised in 1986, and then in 1989, 1992, 1999, 2003, ... 2019

Many database management systems (DBMSs) have been built around SQL

- System-R, Oracle, Ingres, DB2, PostgreSQL, MySQL, SQL-server, SQLite, ...

DBMSs vs the standard

- all DBMSs implement a subset of the 1999 standard (aka SQL3)
- all DBMSs implement proprietary extensions to the standard

Conforming to standard should ensure portability of database applications

## ❖ SQL Intro

SQL has several sub-languages ...

- meta-data definition language (e.g. `create table`, etc.)
- meta-data update language (e.g. `alter table`, `drop table`)
- data update language (e.g. `insert`, `update`, `delete`)
- query language (e.g. `select ... from ... where`, etc.)

Meta-data languages manage the database **schema**

Data update language manages sets of **tuples**

## ❖ SQL Intro (cont)

Syntax-wise, SQL is similar to other programming languages

- has keywords, identifiers, constants, operators
- but strings are different to most PLs
  - '...' are constant strings, e.g. 'a', 'abc123', 'John' 's bag'
  - "..." allow non-alpha chars in identifiers and make id's case-sensitive

*Normal identifier is not case sensitive, but " identifier is case-sensitive.*

In the standard, all non-quoted identifiers map to all upper-case

- e.g. **BankBranches** = **bankbranches** are treated as  
**BANKBRANCHES**

In PostgreSQL, all non-quoted identifiers map to all lower-case

- e.g. **BankBranches** = **BANKBRANCHES** are treated as  
**bankbranches**

In all standards-adhering DBMSs, different quoted identifiers are different

- "BankBranches" ≠ "bankbranches" ≠ "BANKBRANCHES"

## ❖ SQL Syntax in a Nutshell

SQL definitions, queries and statements are composed of:

- comments ... -- comments to end of line
- identifiers ... similar to regular programming languages
- keywords ... a large set (e.g. **CREATE**, **DROP**, **TABLE**)
- data types ... small set of basic types (e.g. **integer**, **date**)
- operators ... similar to regular programming languages
- constants ... similar to regular programming languages

*Similar* means "often the same, but not always" ...

## ❖ SQL Syntax in a Nutshell (cont)

Comments: everything after `--` *is a comment*

Identifiers: alphanumeric (a la C), but also "**An Identifier**"

Reserved words: many e.g. **CREATE, SELECT, TABLE, ...**

Reserved words cannot be used identifiers unless quoted e.g  
`"table"`

Strings: e.g. `'a string'`, `'don't ask'`, but no `'\n'` (use  
`\n`)

Numbers: like C, e.g. `1, -5, 3.14159, ...`

Types: **integer, float, char(n), varchar(n), date, ...**

Operators: `=, <>, <, <=, >, >=, AND, OR, NOT, ...`

## ❖ Names in SQL

Identifiers denote:

- database objects such as tables, attributes, views, ...
- meta-objects such as types, functions, constraints, ...

Naming conventions that I (try to) use in this course:

- relation names: e.g. **Branches**, **Students**, ... (use plurals)
- attribute names: e.g. **name**, **code**, **firstName**, ...
- foreign keys: named after either or both of
  - table being referenced e.g. **staff** or **staff\_id**, ...
  - relationship being modelled e.g. **teaches**, ...

We initially write SQL keywords in all upper-case in slides.

## ❖ Types/Constants in SQL

Numeric types: **INTEGER**, **REAL**, **NUMERIC(*w,d*)**

```
10      -1      3.14159      2e-5      6.022e23
```

String types: **CHAR(*n*)**, **VARCHAR(*n*)**, **TEXT**

```
'John'      'some text'      '!%#%!$'      'O''Brien'  
'''        '[A-Z]{4}\d{4}'      'a VeRy! LoNg String'
```

PostgreSQL provides extended strings containing \ escapes, e.g.

```
E'\n'      E'O\'Brien'      E'[A-Z]{4}\d{4}'      E'John'
```

Type-casting via *Expr :: Type* (e.g. '10'::integer)

## ❖ Types/Constants in SQL (cont)

Logical type: **BOOLEAN**, **TRUE** and **FALSE** (or **true** and **false**)

PostgreSQL also allows '**t**', '**true**', '**yes**', '**f**', '**false**', '**no**'

Time-related types: **DATE**, **TIME**, **TIMESTAMP**, **INTERVAL**

```
'2008-04-13'  '13:30:15'  '2004-10-19 10:23:54'  
'Wed Dec 17 07:37:16 1997 PST'  
'10 minutes'  '5 days, 6 hours, 15 seconds'
```

Subtraction of timestamps yields an interval, e.g.

```
now()::TIMESTAMP - birthdate::TIMESTAMP
```

PostgreSQL also has a range of non-standard types, e.g.

- geometric (point/line/...), currency, IP addresses, JSON, XML, objectIDs, ...
- non-standard types typically use string literals (' . . . ') which need to be interpreted

## ❖ Types/Constants in SQL (cont)

Users can define their own types in several ways:

```
-- domains: constrained version of existing type  
  
CREATE DOMAIN Name AS Type CHECK ( Constraint )  
  
-- tuple types: defined for each table  
  
CREATE TYPE Name AS ( AttrName AttrType, ... )  
  
-- enumerated type: specify elements and ordering  
  
CREATE TYPE Name AS ENUM ( 'Label', ... )
```

## ❖ Examples of Defining Domains/Types

```
-- positive integers
CREATE DOMAIN PosInt AS integer CHECK (value > 0);

-- a UNSW course code
CREATE DOMAIN CourseCode AS char(8)
    CHECK (value ~ '[A-Z]{4}[0-9]{4}');

-- a UNSW student/staff ID
CREATE DOMAIN ZID AS integer
    CHECK (value between 1000000 and 9999999);

-- standard UNSW grades (FL,PS,CR,DN,HD)
CREATE DOMAIN Grade AS char(2)
    CHECK (value in ('FL', 'PS', 'CR', 'DN', 'HD'));
-- or
CREATE TYPE Grade AS ENUM ('FL', 'PS', 'CR', 'DN', 'HD');
```

## ❖ Tuple and Set Literals

Tuple and set constants are both written as:

```
( val1, val2, val3, ... )
```

The correct interpretation is worked out from the context.

Examples:

```
INSERT INTO Student(studeID, name, degree)
    VALUES (2177364, 'Jack Smith', 'BSc')
        -- tuple literal

CONSTRAINT CHECK gender IN ('male','female','unspecified')
        -- set literal
```



# SQL Expressions

---

- Expressions in SQL
- SQL Operators
- The **NULL** Value
- Conditional Expressions

## ❖ Expressions in SQL

Expressions in SQL involve: objects, constants, operators

- objects are typically names of attributes (or PLpgSQL variables)
- operators may be symbols (e.g. `+`, `=`) or keywords (e.g. **between**)

SQL constants are similar to typical programming language constants

- integers: `123, -5`; floats: `3.14, 1.0e-3`; boolean: **true, false**

But strings are substantially different

- '`...`' rather than "`...`", no `\n`-like "escape" chars
- escape mechanisms: `'O' 'Brien'` or `E'O\'Brien'` (non-standard)
- dollar quoting: `$$O'Brien$$` or `$tag$O'Brien$tag$`

## ❖ SQL Operators

Comparison operators are defined on all types:

```
<    >    <=    >=    =    <>
```

In PostgreSQL, `!=` is a synonym for `<>` (but there's no `==`)

Boolean operators **AND**, **OR**, **NOT** are also available

Note **AND**,**OR** are not "short-circuit" in the same way as C's `&&`, `||`

Most data types also have type-specific operations available

String comparison (e.g.  $str_1 < str_2$ ) uses dictionary order

See PostgreSQL Documentation Chapter 8/9 for data types and operators

## ❖ SQL Operators (cont)

SQL provides pattern matching for strings via **LIKE** and **NOT LIKE**

- `%` matches anything (cf. regexp `.*`)
- `_` matches any single char (cf. regexp `.`)

Examples:

<code>name LIKE 'Ja%</code>	<code>name</code> begins with 'Ja'
<code>name LIKE '_i%</code>	<code>name</code> has 'i' as 2nd letter
<code>name LIKE '%o%o%</code>	<code>name</code> contains two 'o's
<code>name LIKE '%ith'</code>	<code>name</code> ends with 'ith'
<code>name LIKE 'John'</code>	<code>name</code> equals 'John'

PostgreSQL also supports case-insensitive matching: **ILIKE**

## ❖ SQL Operators (cont)

PostgreSQL provides **regexp**-based pattern matching via `~` and `!~`

Examples (using POSIX regular expressions):

<b>name</b> ~ '^Ja'	<b>name</b> begins with 'Ja'
<b>name</b> ~ '^.{i}'	<b>name</b> has 'i' as 2nd letter
<b>name</b> ~ '.*o.*o.*'	<b>name</b> contains two 'o's
<b>name</b> ~ 'ith\$'	<b>name</b> ends with 'ith'
<b>name</b> ~ 'John'	<b>name</b> contains 'John'

Also provides case-insensitive matching via `~*` and `!~*`

## ❖ SQL Operators (cont)

Other operators/functions for string manipulation:

- $str_1 \ | \ | \ str_2$  ... return concatenation of  $str_1$  and  $str_2$
- **lower**( $str$ ) ... return lower-case version of  $str$
- **substring**( $str,start,count$ ) ... extract substring from  $str$

Etc. etc. ... consult your local SQL Manual (e.g. PostgreSQL Sec 9.4)

Note that above operations are null-preserving (strict):

- if any operand is **NULL**, result is **NULL**
- beware of  $(a \ | \ | \ ' \ | \ | \ b)$  ... **NULL** if either of **a** or **b** is **NULL**

## ❖ SQL Operators (cont)

Arithmetic operations:

+ - \* / abs ceil floor power sqrt sin etc.

Aggregations "summarize" a column of numbers in a relation:

- **count**(*attr*) ... number of rows in *attr* column
- **sum**(*attr*) ... sum of values for *attr*
- **avg**(*attr*) ... mean of values for *attr*
- **min/max**(*attr*) ... min/max of values for *attr*

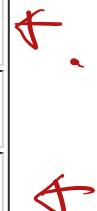
Note: **count** applies to columns of non-numbers as well.

## ❖ The **NULL** Value

Expressions containing **NULL** generally yield **NULL**.

However, boolean expressions use three-valued logic:

<i>a</i>	<i>b</i>	<i>a AND b</i>	<i>a OR b</i>
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL



## ❖ The **NULL** Value (cont)

Important consequence of **NULL** behaviour ...

These expressions do not work as (might be) expected:

```
x = NULL      x <> NULL
```

Both return **NULL** regardless of the value of  $x$

Can only test for **NULL** using:

```
x IS NULL      x IS NOT NULL
```

## ❖ Conditional Expressions

Other ways that SQL provides for dealing with **NULL**:

**coalesce( $val_1, val_2, \dots val_n$ )**

- returns first non-null value  $val_i$
- useful for providing a "displayable" value for nulls

E.g. **select coalesce(mark, '??') from Marks ...**

**nullif( $val_1, val_2$ )**

- returns **NULL** if  $val_1$  is equal to  $val_2$
- can be used to implement an "inverse" to **coalesce**

E.g. **nullif(mark, '??')**

## ❖ Conditional Expressions (cont)

SQL also provides a generalised conditional expression:

```
CASE
    WHEN test1 THEN result1
    WHEN test2 THEN result2
    ...
    ELSE resultn
END
```

E.g. **case when mark>=85 then 'HD' ... else '??'** **end**

Tests that yield **NULL** are treated as **FALSE**

If no **ELSE**, and all tests fail, **CASE** yields **NULL**

---

Produced: 22 Sep 2020

# SQL Data Definition Language

---

- Relational Data Definition
- Example Relational Schema
- SQL Data Definition Language
- Defining a Database Schema
- Data Integrity
- Another Example Schema
- Default Values
- Defining Keys
- Attribute Value Constraints
- Named Constraints

## ❖ Relational Data Definition

In order to give a relational data model, we need to:

- describe tables
- describe attributes that comprise tables
- describe any constraints on the data

A **relation schema** defines an individual table

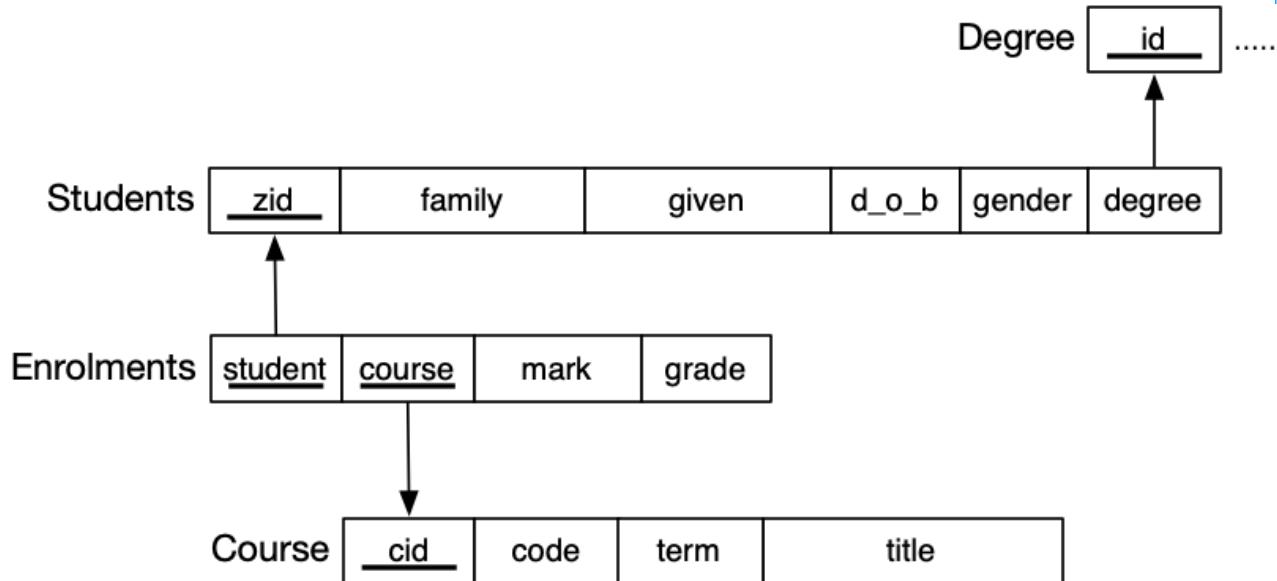
- table name, attribute names, attribute domains, keys, etc.

A **database schema** is a collection of relation schemas that

- defines the structure the whole database
- additional constraints on the whole database

## ❖ Example Relational Schema

So far, we have given relational schemas informally, e.g.



## ❖ SQL Data Definition Language

In the example schema above, we provided only

- relation names, attribute names, primary keys, foreign keys

A usable database needs to provide much more detail

SQL has a rich data definition language (DDL) that can describe

- names of tables
- names and domains for attributes
- various types of constraints (e.g. primary/foreign keys)

It also provides mechanisms for performance tuning (see later).

## ❖ Defining a Database Schema

Tables (relations) are described using:

```
CREATE TABLE TableName (
    attribute1   domain1   constraints1,
    attribute2   domain2   constraints2,
    ...
    table-level constraints, ...
)
```

This SQL statement ..

- defines the table schema (adds it to database meta-data)
- creates an empty instance of the table (zero tuples)

Tables are removed via **DROP TABLE *TableName*;**

## ❖ Defining a Database Schema (cont)

Example: defining the **Students** table ...

```
CREATE TABLE Students (
    zid      serial,
    family   varchar(40),
    given    varchar(40) NOT NULL,
    d_o_b    date NOT NULL,
    gender   char(1) CHECK (gender in ('M', 'F')),
    degree   integer,
    PRIMARY KEY (zid),
    FOREIGN KEY (degree) REFERENCES Degrees(did)
);
```

Note that there is much more info here than in the relational schema diagram.

A primary key attribute is implicitly defined to be **UNIQUE** and **NOT NULL**

## ❖ Defining a Database Schema (cont)

Example: alternative definition of the **Students** table ...

```
CREATE DOMAIN GenderType AS
    char(1) CHECK (value in ('M', 'F'));

CREATE TABLE Students (
    zid      serial PRIMARY KEY,
    -- only works if primary key is one attr
    family   text, -- no need to worry about max length
    given    text NOT NULL,
    d_o_b    date NOT NULL,
    gender   GenderType,
    degree   integer REFERENCES Degrees(did)
);
```

At this stage, prefer to use the long-form declaration of primary and foreign keys

## ❖ Defining a Database Schema (cont)

Example: defining the **Courses** table ...

```
CREATE TABLE Courses (
    cid      serial,
    code     char(8) NOT NULL uhs
              CHECK (code ~ '[A-Z]{4}[0-9]{4}' ),
    term     char(4) NOT NULL
              CHECK (term ~ '[0-9]{2}T[0-3]' ),
    title   text UNIQUE NOT NULL,
    PRIMARY KEY (cid)
);
```

Uses non-standard regular expression checking on **code** and **term**

No two **Courses** can have the same title; but not used as primary key

## ❖ Defining a Database Schema (cont)

Example: defining the **Enrolments** relationship ...

```
CREATE TABLE Enrolments (
    student integer,
    course integer,
    mark    integer CHECK (mark BETWEEN 0 AND 100),
    grade   GradeType,
    PRIMARY KEY (student, course),
    FOREIGN KEY (student)
        REFERENCES Students(zid)
    FOREIGN KEY (course)
        REFERENCES Courses(cid)
);
```

Could not enforce total participation constraint if e.g. all courses must have > 0 students

Possible alternative names for foreign keys **student\_id** and **course\_id**

## ❖ Data Integrity

Defining tables as above affects behaviour of DBMS when changing data

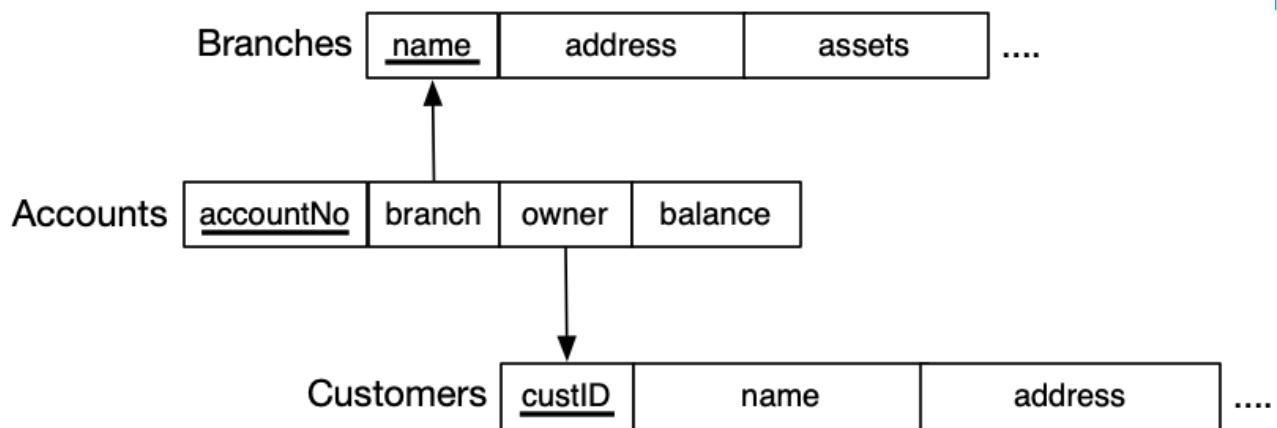
Constraints and types ensure that integrity of data is preserved

- no duplicate keys
- no "dangling references"
- all attributes have valid values
- etc. etc. etc.

Preserving data integrity is a *critical* function of a DBMS.

## ❖ Another Example Schema

Variation on banking schema used elsewhere



## ❖ Default Values

Can specify a **DEFAULT** value for an attribute

- will be assigned to attribute if no value is supplied during insert

### Example:

```
CREATE TABLE Accounts (
    acctNo  char(5) PRIMARY KEY,
    branch  varchar(30) REFERENCES Branches(name)
                  DEFAULT 'Central',
    owner    integer REFERENCES Customers(custID),
    balance  float DEFAULT 0.0
);

INSERT INTO Accounts(acctNo,owner) VALUES ('A-456',645342)
-- produces the tuple
Accounts('A-456', 'Central', 645342, 0.0)
```

## ❖ Defining Keys

Primary keys:

- if PK is one attribute, can define as attribute constraint
- if PK is multiple attributes, must define in table constraints
- PK implies **NOT NULL UNIQUE** for all attributes in key

Foreign keys:

- if FK is one attribute, can define as attribute constraint
- can omit **FOREIGN KEY** keywords in attribute constraint
- if FK has multiple attributes, must define as a single table constraint
- should always specify corresponding PK attribute in FK constraint, e.g

```
customer integer  
    FOREIGN KEY REFERENCES Customers(customerNo)
```

## ❖ Defining Keys (cont)

Defining primary keys assures entity integrity

- must give values for all attributes in the primary key

For example this insertion would fail ...

```
INSERT INTO Enrolments(student, course, mark, grade)
VALUES (5123456, NULL, NULL, NULL);
```

because no **course** was specified; but **mark** and **grade** can be **NULL**

Defining primary keys assures uniqueness

- cannot insert a tuple which contains an existing PK value

## ❖ Defining Keys (cont)

Defining foreign keys assures **referential integrity**.

On insertion, cannot add a tuple where FK value does not exist as a PK

For example, this insert would fail ...

```
INSERT INTO Accounts(acctNo, owner, branch, balance)
VALUES ('A-123', 765432, 'Nowhere', 5000);
```

if there is no customer with id **765432** or no branch **Nowhere**

## ❖ Defining Keys (cont)

On deletion, interesting issues arise, e.g.

**Accounts .branch** refers to primary key **Branches .name**

If we want to delete a tuple from **Branches**, and there are tuples in **Accounts** that refer to it, we could ...

- **reject** the deletion (PostgreSQL/Oracle default behaviour)
- **set-NULL** the foreign key attributes in **Account** records
- **cascade** the deletion and remove **Account** records

SQL allows us to choose a strategy appropriate for the application

## ❖ Attribute Value Constraints

**NOT NULL** and **UNIQUE** are special constraints on attributes.

SQL has a general mechanism for specifying attribute constraints

```
attrName  type  CHECK ( Condition )
```

*Condition* is a boolean expression and can involve other attributes, relations and **SELECT** queries.

```
CREATE TABLE Example
(
    gender char(1)    CHECK (gender IN ('M','F')) ,
    Xvalue integer    NOT NULL,
    Yvalue integer    CHECK (Yvalue > Xvalue),
    Zvalue float      CHECK (Zvalue >
                                (SELECT MAX(price)
                                 FROM   Sells)
                            )
);
```

(but many RDBMSs (e.g. Oracle and PostgreSQL) don't allow **SELECT** in **CHECK**)

## ❖ Named Constraints

A constraint in an SQL table definition can (optionally) be named via

```
CONSTRAINT constraintName constraint
```

Example:

```
CREATE TABLE Example
(
    gender char(1) CONSTRAINT GenderCheck
        CHECK (gender IN ('M', 'F')) ,
    Xvalue integer NOT NULL,
    Yvalue integer CONSTRAINT XYOrder
        CHECK (Yvalue > Xvalue),
);
```



# Relational Database Management Systems

- What is an RDBMS?
- RDBMSs in COMP3311
- PostgreSQL Architecture
- SQLite Architecture
- Using PostgreSQL in CSE
- Managing Databases
- Managing Tables
- Managing Tuples
- Table Definition Example
- Exercise: Creating/Populating Databases
- Managing Other DB Objects

## ❖ What is an RDBMS?

A **relational database management system** (RDBMS) is

- software designed to support large-scale data-intensive applications
- allowing high-level description of data (tables, constraints)
- with high-level access to the data (relational model, SQL)
- providing efficient storage and retrieval (disk/memory management)
- supporting multiple simultaneous users (privilege, protection)
- doing multiple simultaneous operations (transactions, concurrency)
- maintaining reliable access to the stored data (backup, recovery)

Note: databases provide **persistent** storage of information

## ❖ RDBMSs in COMP3311

### PostgreSQL

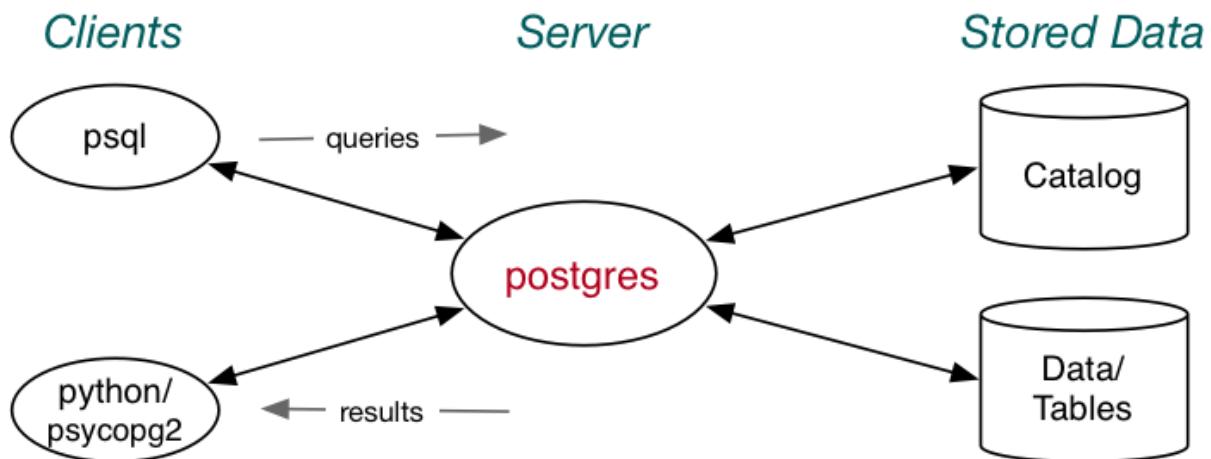
- full-featured, client-server DBMS, resource intensive
- applications communicate via server to DB
- can run distributed and replicated
- follows SQL standard closely, but not totally
- extra data types (e.g. JSON), multiple procedural languages

### SQLite

- full-featured, serverless DBMS, light user of resources
- intended to be embedded in applications
- follows SQL standard closely, but not totally
- no stored procedures, add functions by embedding in apps

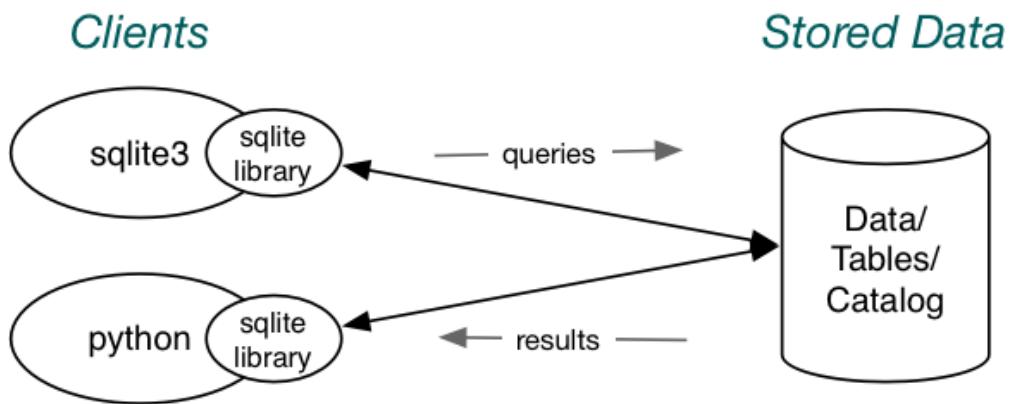
## ❖ PostgreSQL Architecture

PostgreSQL's client-server architecture:



## ❖ SQLite Architecture

SQLite's serverless architecture:



## ❖ Using PostgreSQL in CSE

Using your PostgreSQL server in CSE (once installed):

- login to grieg, set up environment, start server
- use psql, etc. to manipulate databases
- stop server, log off grieg

```
wagner$ ssh YOU@grieg
grieg$ source /srvr/YOU/env
grieg$ pg start
grieg$ psql myDatabase
... do stuff with your database ...
grieg$ pg stop
grieg$ exit
```

Need to run the command **priv srvr** once before the above will work

## ❖ Using PostgreSQL in CSE (cont)

PostgreSQL files (helps to understand state of server)

- PostgreSQL environment settings ... **/srvr/you/env**
- PostgreSQL home directory ... **/srvr/you/sql/**
- under the home directory ...
  - **postgresql.conf** ... main configuration file
  - **base/** ... subdirectories containing database files
  - **postmaster.pid** ... process ID of server process
  - **.s.PGSQL.5432** ... socket for clients to connect to server
  - **.s.PGSQL.5432.lock** ... lock file for socket
  - **Log** ... log file to monitor server errors, etc.

## ❖ Managing Databases

Shell commands to create/remove databases:

- **createdb** *dbname* ... create a new totally empty database
- **dropdb** *dbname* ... remove *all* data associated with a DB

(If no *dbname* supplied, assumes a database called *YOU*)

Shell commands to dump/restore database contents:

- **pg\_dump** *dbname* > *dumpfile*
- **psql** *dbname -f* *dumpfile*

(Database *dbname* is typically created just before restore)

Main SQL statements in *dumpfile*: **CREATE TABLE**, **ALTER TABLE**, **COPY**

## ❖ Managing Tables

SQL statements:

- **CREATE TABLE** *table* (*Attributes+Constraints*)
- **ALTER TABLE** *table* *TableSchemaChanges*
- **DROP TABLE** *table(s)* [ **CASCADE** ]
- **TRUNCATE TABLE** *table(s)* [ **CASCADE** ]

(All conform to SQL standard, but all also have extensions)

**DROP . . CASCADE** also drops objects which depend on the table

- objects could be tuples or views, but *not* whole tables

**TRUNCATE . . CASCADE** truncates tables which refer to the table

## ❖ Managing Tuples

SQL statements:

- **INSERT INTO** *table* ( *Attrs* ) **VALUES** *Tuple(s)*
- **DELETE FROM** *table* **WHERE** *condition*
- **UPDATE** *table* **SET** *AttrValueChanges* **WHERE** *condition*

*Attrs* = ( *attr*<sub>1</sub>, ... *attr*<sub>*n*</sub> )      *Tuple* = ( *val*<sub>1</sub>, ... *val*<sub>*n*</sub> )

*AttrValueChanges* is a comma-separated list of:

- *attrname* = *expression*

Each list element assigns a new value to a given attribute.

## ❖ Table Definition Example

Make a table to hold student data:

```
CREATE TABLE Student (
    zid      serial,
    family   varchar(40),
    given    varchar(40) NOT null,
    d_o_b    date NOT NULL,
    gender   char(1) check (gender in ('M','F')),
    degree   integer,
    PRIMARY KEY (zid),
    FOREIGN KEY (degree) REFERENCES Degrees(did)
);
```

**serial** is a special type which automaticall generates unique integer values

## ❖ Exercise: Creating/Populating Databases

Do the following:

- create a database called **ex1**
- create a table **T** with two integer fields **x** and **y**
- examine the catalog definition of table **T**
- use **insert** statements to load some tuples
- use **pg\_dump** to make a copy of the database contents
- remove the **ex1** database, then restore it from the dump

## ❖ Managing Other DB Objects

Databases contain objects other than tables and tuples:

- views, functions, sequences, types, indexes, roles, ...

Most have SQL statements for:

- **CREATE** *ObjectType name* ...
- **DROP** *ObjectType name* ...

Views and functions also have available:

- **CREATE OR REPLACE** *ObjectType name* ...

See PostgreSQL documentation Section VI, Chapter I for SQL statement details.

---

Produced: 20 Sep 2020

# Mapping ER to SQL

- Mapping ER to SQL
- Reminder: SQL/Relational Model vs ER Model
- Mapping ER to SQL
- Mapping Strong Entities
- Mapping Weak Entities
- Mapping N:M Relationships
- Mapping 1:N Relationships
- Mapping 1:1 Relationships
- Mapping n-way Relationships
- Mapping Composite Attributes
- Mapping Multi-valued Attributes (MVAs)
- Mapping Subclasses

## ❖ Mapping ER to SQL

We have explored mapping ER designs to relational schemas

SQL schemas are essentially more detailed versions of relational schemas

The mapping is much the same, except that

- you need to provide more details on allowed values
- you can map some ideas from ER that are not in relational schemas

There are also some ideas from ER than do not map to an SQL schema

## ❖ Reminder: SQL/Relational Model vs ER Model

Correspondences between SQL/relational and ER data models:

- $\text{attribute(ER)} \approx \text{attribute(Rel)}$ ,  $\text{entity(ER)} \approx \text{row/tuple(Rel)}$
- $\text{entity set(ER)} \approx \text{table/relation(Rel)}$ ,  $\text{relationship(ER)} \approx \text{table/relation(Rel)}$

Differences between SQL and ER models:

- SQL uses tables to model entities *and* relationships
- SQL has no composite or multi-valued attributes (only atomic)
- SQL has no object-oriented notions (e.g. subclasses, inheritance)

Note that ...

- not all aspects of ER can be represented exactly in an SQL schema
- some aspects of SQL schemas (e.g. domains) do not appear in ER

## ❖ Mapping ER to SQL

Some conventions that we use in mapping ER to SQL

- stop using upper-case for SQL keywords (use `table` vs `TABLE`)
- all tables based on entities are given plural names
- attributes in entities are given the same name in ER and SQL
- attributes in relationships are given the same name in ER and SQL
- ER key attributes are defined using **primary key**
- text-based attributes are defined with type **text**,  
unless there is a size which is obvious from the context
- attribute domains can be PostgreSQL-specific types where useful
- foreign keys within entity tables are named after the relationship
- foreign keys in relationship tables are named **table\_id**

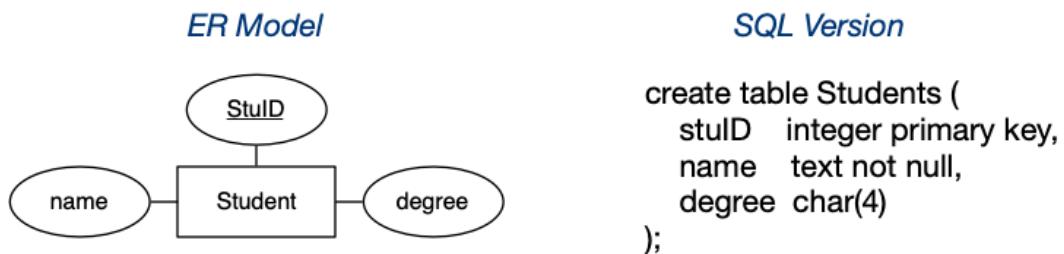
## ❖ Mapping Strong Entities

An entity set  $E$  with atomic attributes  $a_1, a_2, \dots, a_n$

maps to

A table  $R$  with attributes (columns)  $a_1, a_2, \dots, a_n$

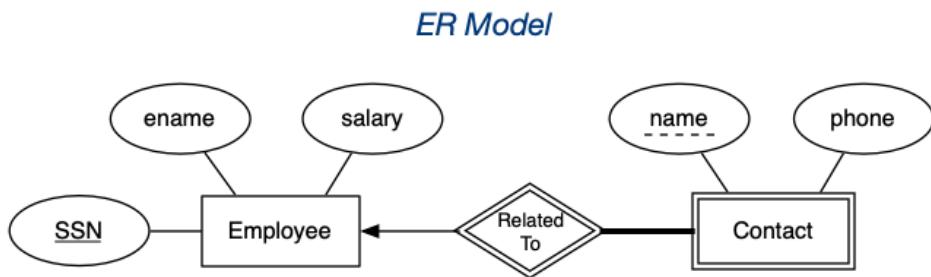
Example:



Note: the key is preserved in the mapping.

## ❖ Mapping Weak Entities

Example:



*SQL Version*

```

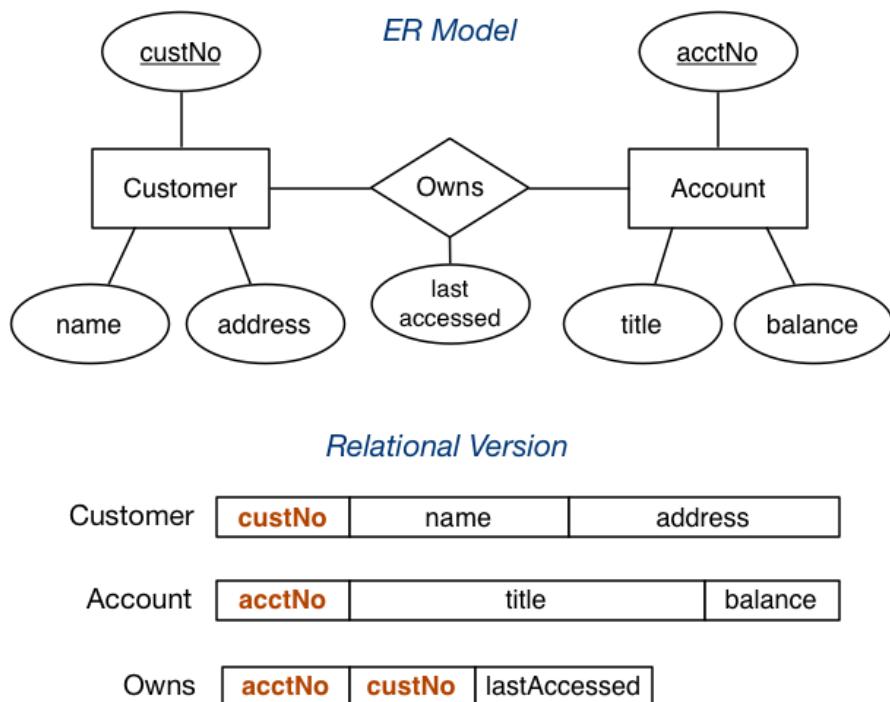
create table Employees (
    SSN    text primary key,
    ename   text,
    salary   currency
);
  
```

```

create table Contacts (
    relatedTo text not null, -- total participation
    name      text,          -- not null implied by PK
    phone     text not null,
    primary key (relatedTo, name),
    foreign key (relatedTo) references Employees (ssn)
);
  
```

## ❖ Mapping N:M Relationships

Example:

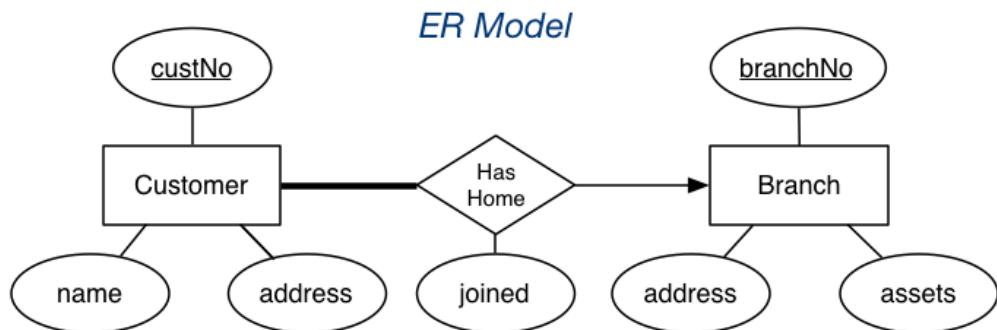


## ❖ Mapping N:M Relationships (cont)

```
create table Customers (
    custNo serial primary key,
    name    text not null,
    address text -- don't need to know customer's address
);
create table Accounts (
    acctNo  char(5) check (acctNo ~ '[A-Z]-[0-9]{3}'),
    title   text not null, -- acctNos are like 'A-123'
    balance float default 0.0,
    primary key (acctNo)
);
create table Owns (
    customer_id integer references Customers(custNo),
    account_id  char(5) references Accounts(acctNo),
    last_accessed timestamp,
    primary key (customer_id, account_id)
);
```

## ❖ Mapping 1:N Relationships

Example:



*Relational Version*

Customer	<b>custNo</b>	name	address	<b>branchNo</b>	joined
----------	---------------	------	---------	-----------------	--------

Branch	<b>branchNo</b>	address	assets
--------	-----------------	---------	--------

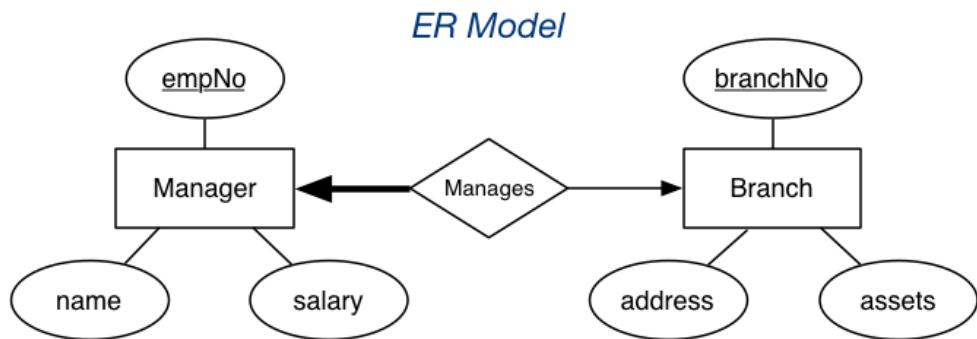
## ❖ Mapping 1:N Relationships (cont)

```
create table Branches (
    branchNo serial primary key,
    address  text not null,
    assets   currency
);
create table Customers (
    custNo  serial primary key,
    name    text not null,
    address text,
    hasHome integer not null, -- total participation
    joined  date not null,
    foreign key (hasHome) references Branches(branchNo)
);
```

**hasHome** implements the 1:n relationship; **not null** implements total participation

## ❖ Mapping 1:1 Relationships

Example:



*Relational Version*

Manager	<b>empNo</b>	name	salary	<b>branchNo</b>
---------	--------------	------	--------	-----------------

Branch	<b>branchNo</b>	address	assets
--------	-----------------	---------	--------

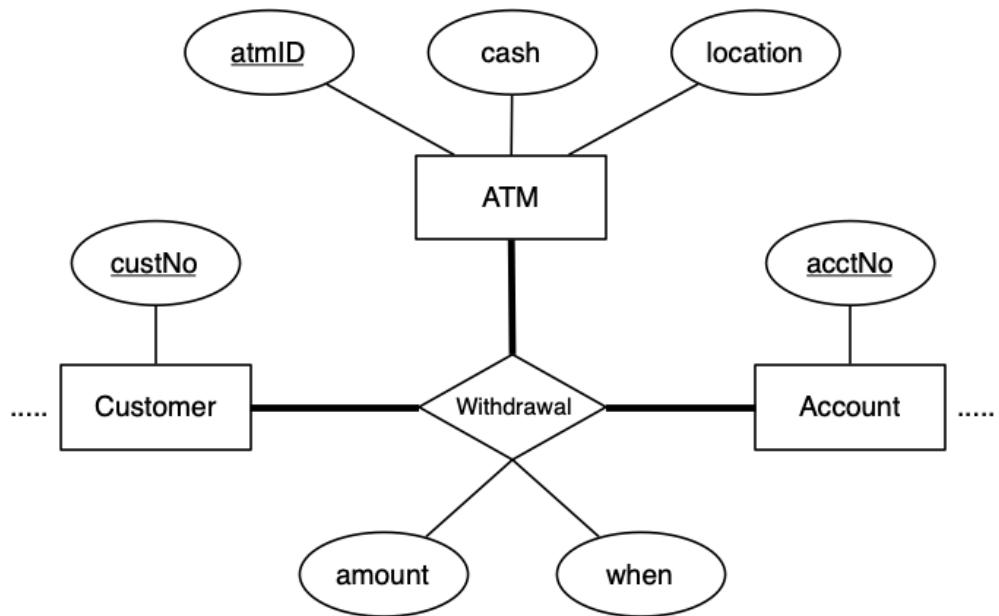
## ❖ Mapping 1:1 Relationships (cont)

```
create table Branches (
    branchNo serial primary key,
    address  text not null,
    assets   currency          -- a new branch
);                                -- may have no accounts
create table Managers (
    empNo    serial primary key,
    name     text not null,
    salary   currency not null, -- when first employed,
                           -- must have a salary
    manages  integer not null, -- total participation
    foreign key (manages) references Branches(branchNo)
);
```

If both entities have total participation, cannot express this in SQL  
except by putting a (redundant) **not null** foreign key in one table

## ❖ Mapping n-way Relationships

Example:



A customer accesses one of their accounts at a specific ATM

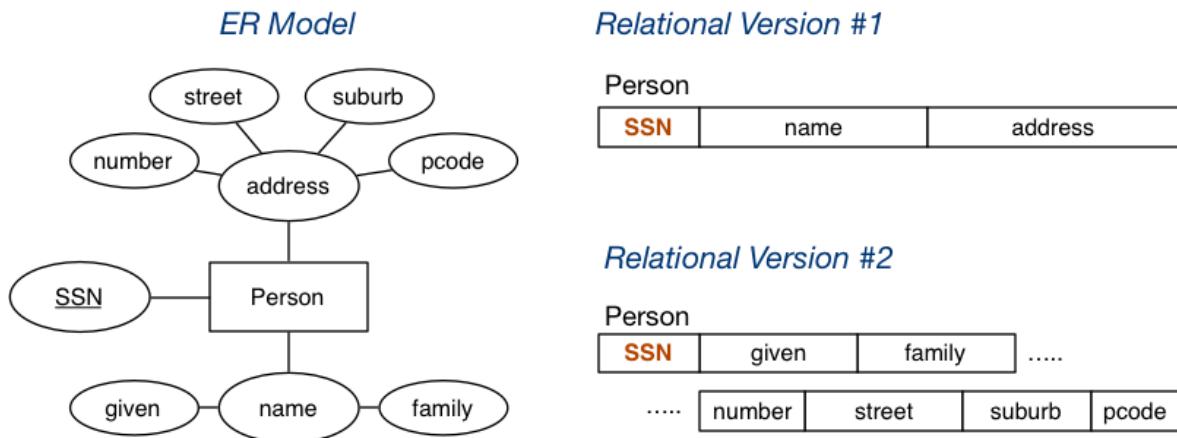
## ❖ Mapping n-way Relationships (cont)

```
create table Customers (
    custNo    serial primary key, ...
);
create table Accounts (
    acctNo    char(5) ... primary key, ...
);
create table ATMs (
    atmID     serial primary key,
    cash      currency check (cash >= 0),
    location  text not null
);
create table Withdrawal (
    customer_id  integer references Customers(custNo),
    account_id   char(5) references Accounts(acctNo),
    atm_id       integer references ATMs(atmID),
    amount       currency not null,
    when         timestamp default now(),
    primary key  (customer_id,account_id,atm_id)
);
```

## ❖ Mapping Composite Attributes

Composite attributes are mapped by concatenation or flattening.

Example:



## ❖ Mapping Composite Attributes (cont)

```
-- Version 1: concatenated
create table People (
    ssn      integer primary key,
    name     text not null,
    address  text not null
);
-- Version 2: flattened
create table People (
    ssn      integer primary key,
    given    text not null,
    family   text,
    number   integer not null,
    street   text not null,
    suburb   text not null,
    pcode    char(4) not null check (pcode ~ '[0-9]{4}')
);
address = (number::text||' '||street||', '||suburb||' '||pcode)
```

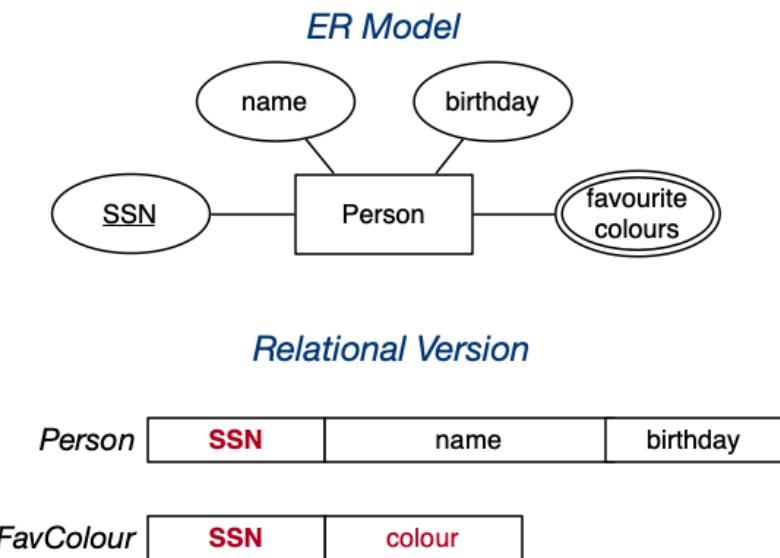
Searching: **suburb** = 'Coogee' vs **address** like '%Coogee%'

Sorting: **order by family** vs can't be done (easily)

## ❖ Mapping Multi-valued Attributes (MVAs)

MVAs are mapped by a new table linking values to their entity.

Example:



## ❖ Mapping Multi-valued Attributes (MVAs) (cont)

```
create table People (
    ssn      integer primary key,
    name     text not null,
    birthday date
);
create table FavColour (
    person_id integer references People(ssn),
    colour    text,
    primary key (person_id,colour)
);
```

Note that **colour** is implicitly **not null** because it is part of the primary key

## ❖ Mapping Subclasses

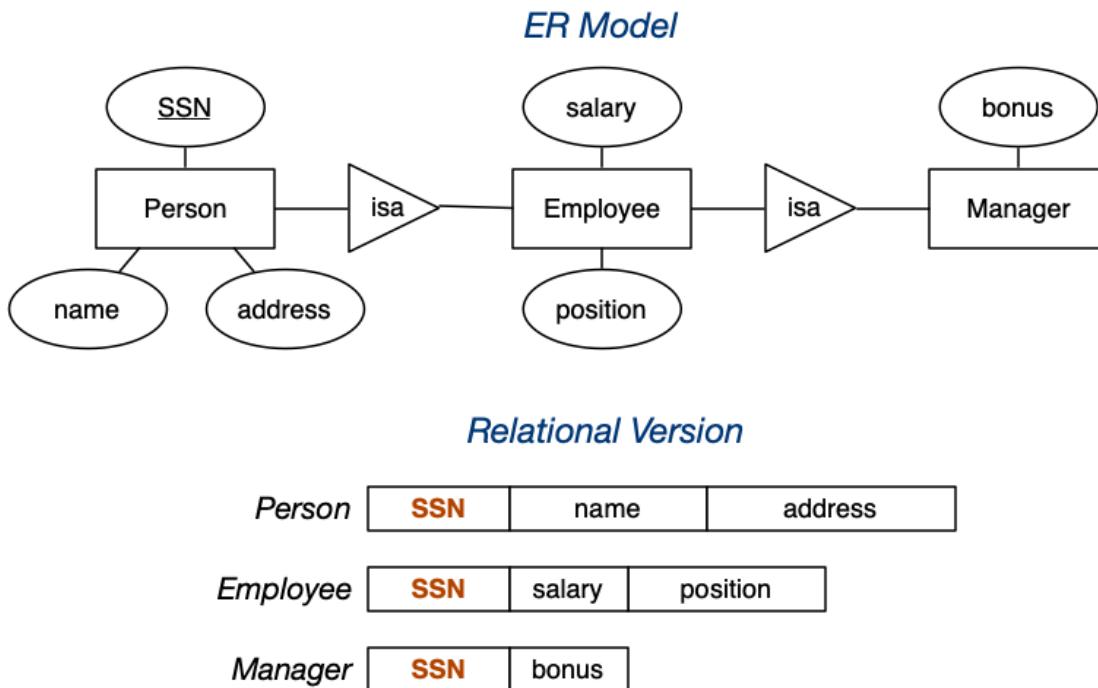
Three different approaches to mapping subclasses to tables:

- ER style
  - each entity becomes a separate table,
  - containing attributes of subclass + FK to superclass table
- object-oriented
  - each entity becomes a separate table,
  - inheriting all attributes from all superclasses
- single table with nulls
  - whole class hierarchy becomes one table,
  - containing all attributes of all subclasses (null, if unused)

Which mapping is best depends on how data is to be used.

## ❖ Mapping Subclasses (cont)

Example of ER-style mapping:

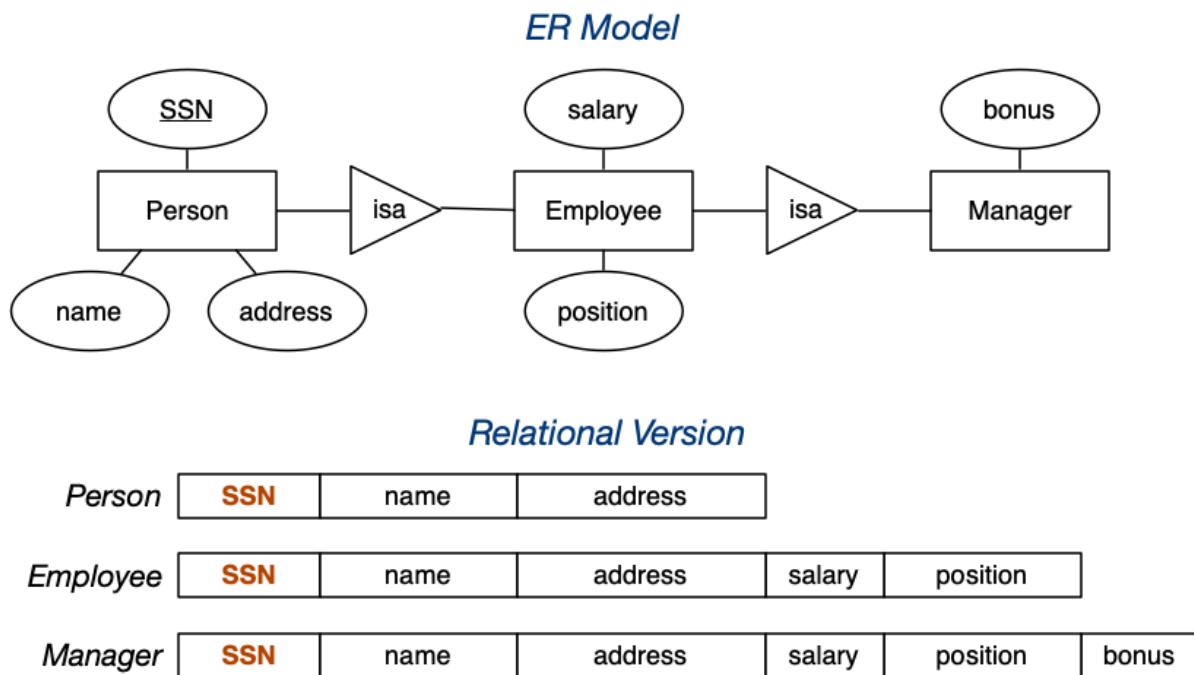


## ❖ Mapping Subclasses (cont)

```
create table People (
    ssn      integer primary key,
    name     text not null,
    address  text
);
create table Employees (
    person_id integer primary key,
    salary    currency not null,
    position  text not null,
    foreign key (person_id) references People(ssn)
);
create table Managers (
    employee_id integer primary key,
    bonus      currency,
    foreign key (employee_id)
        references Employees(person_id)
);
```

## ❖ Mapping Subclasses (cont)

Example of object-oriented mapping:

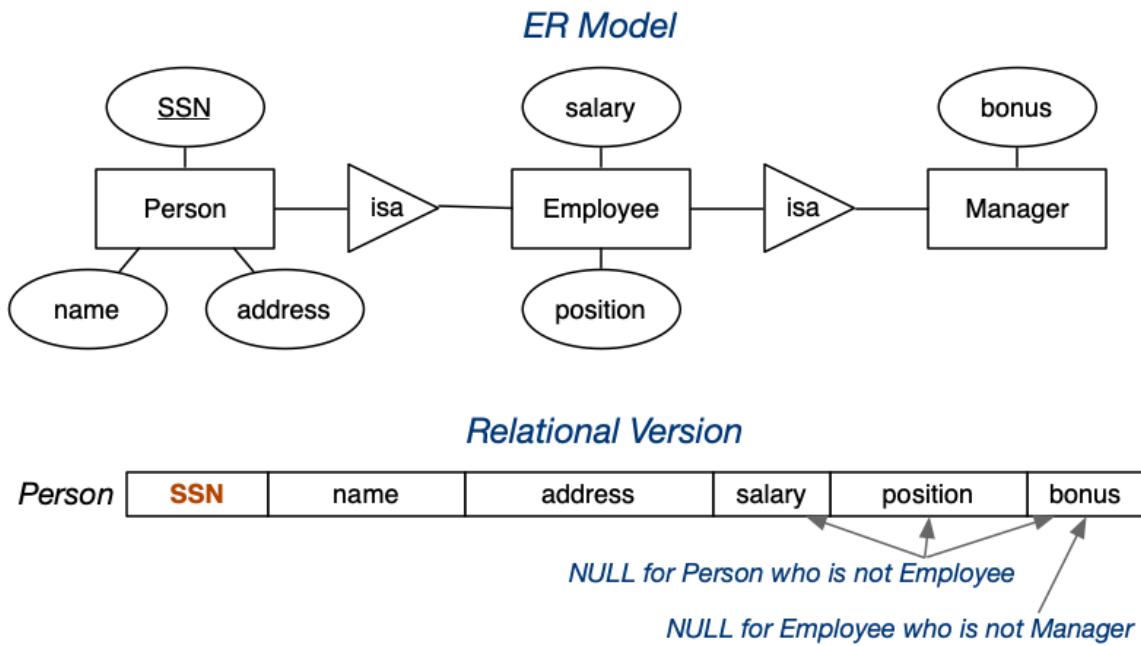


## ❖ Mapping Subclasses (cont)

```
create table People (
    ssn      integer primary key,
    name     text not null,
    address  text
);
create table Employees (
    ssn      integer primary key,
    name     text not null,
    address  text
    salary   currency not null,
    position text not null,
    foreign key (ssn) references People(ssn)
);
create table Managers (
    ssn      integer primary key,
    name     text not null,
    address  text
    salary   currency not null,
    position text not null,
    bonus    currency,
    foreign key (ssn) references People(ssn)
);
```

## ❖ Mapping Subclasses (cont)

Example of single-table-with-nulls mapping:

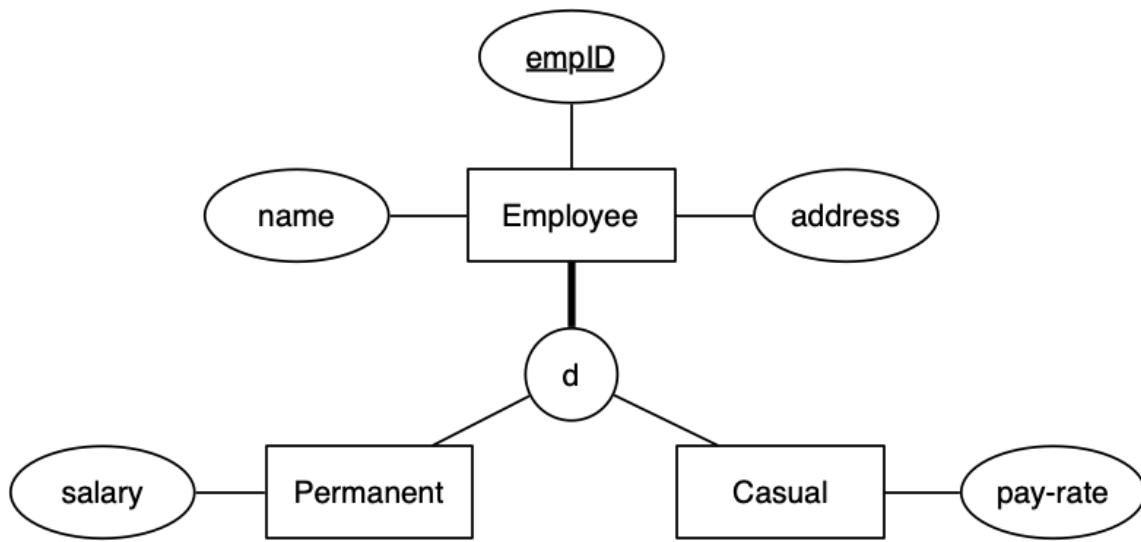


## ❖ Mapping Subclasses (cont)

```
create table People (
    ssn      integer primary key,
    ptype    char(1) not null
              check (ptype in ('P','E','M')),
    name     text not null,
    address  text
    salary   currency,
    position text,
    bonus    currency,
    constraint subclasses check
        ((ptype = 'P' and salary is null
          and position is null and bonus is null)
        or
        (ptype = 'E' and salary is not null
          and position is not null and bonus is null)
        or
        (ptype = 'M' and salary is not null
          and position is not null and bonus is not null))
);
```

## ❖ Mapping Subclasses (cont)

Example:



Every employee is either permanent or casual, but not both.

## ❖ Mapping Subclasses (cont)

ER-style mapping to SQL schema:

```
create table Employees (
    empID    serial primary key,
    name     text not null,
    address  text not null
);
create table Permanents (
    employee_id integer primary key,
    salary      currency not null,
    foreign key (employee_id) references Employees(empID)
);
create table Casuals (
    employee_id integer primary key,
    pay_rate    currency not null,
    foreign key (employee_id) references Employees(empID)
);
```

Does *not* capture either participation or disjoint-ness constraints!

Would need to program a solution to this e.g web-form that requires user to enter both Employee and subclass info



# SQL: Sample Database

---

- SQL Sample Database

## ❖ SQL Sample Database

---

It is easier to discuss SQL via concrete examples

We use a database about beer, people who drink it, ...

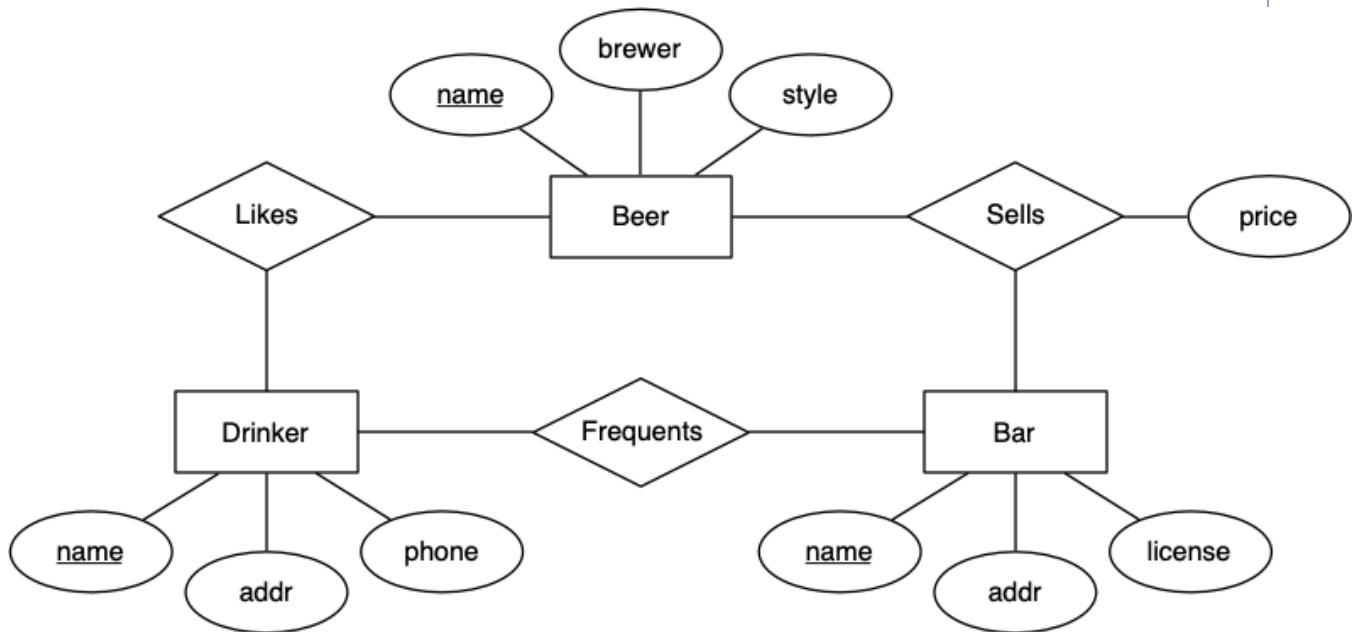
- beers have a name and style, and are made by a brewery
- beer drinkers go to bars/hotels where they can drink beer
- bars/hotels sell a variety of beers, at varying prices
- drinkers have favourite beers and other beers they like

Many other aspects, e.g. ABV, ratings, notes, etc. could have been added

This database is not autobiographical ...

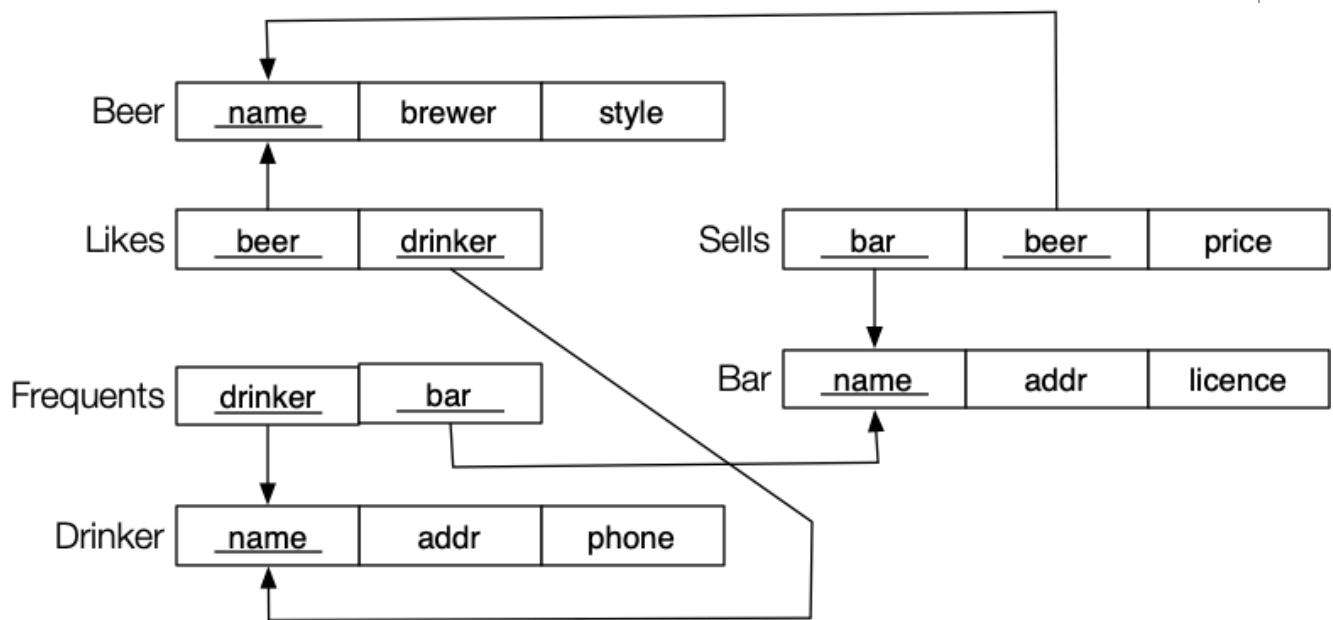
## ❖ SQL Sample Database (cont)

ER design for beer database ...



## ❖ SQL Sample Database (cont)

Relational model for beer database ...



## ❖ SQL Sample Database (cont)

SQL schema for beer database ...

```
-- Some useful data "types"

create domain BeerName varchar(50);
create domain BarName varchar(30);
create domain DrinkerName varchar(20);

-- Tables based on entities

create table Beers (
    name      BeerName,
    brewer    varchar(40) not null,
    style     varchar(40),
    primary key (name)
);

create table Bars (
    name      BarName,
    addr     varchar(20),
    license  integer not null,
    primary key (name)
);

create table Drinkers (
    name      DrinkerName,
    addr     varchar(30) not null,
    phone    char(10) not null,
    primary key (name)
);

-- Tables based on relationships

create table Sells (
    bar      BarName,
    beer    BeerName,
    price   float,
    primary key (bar,beer),
    foreign key (bar) references Bars(name),
    foreign key (beer) references Beers(name)
);
```

```
create table Likes (
    drinker DrinkerName,
    beer     BeerName,
    primary key (drinker,beer),
    foreign key (drinker) references Drinkers(name),
    foreign key (beer) references Beers(name)
);

create table Frequents (
    drinker DrinkerName,
    bar      BarName,
    primary key (drinker,bar),
    foreign key (drinker) references Drinkers(name),
    foreign key (bar) references Bars(name)
);
```

## ❖ SQL Sample Database (cont)

Sample beer data ...

beer=# select * from Beers order by name limit 15;		
name	brewer	style
1750 Export Porter	Kees	Imperial Red Rye IPA
4D	Dainton	Scotch Ale
80/-	Caledonian	NEIPA
Age of Aquarius	Garage Project	Flanders Red Ale
Alexander	Rodenbach	Amber Ale
Amber Ale	James Squire	Imperial Stout
Apollo After Dark	Hawkers	Red Biere de Garde
Astrolabe	Frenchies	Russian Imperial Stout
BBARIS	Mismatch	Pastry Stout
Banana Pastry Stout	Hop Nation	Pale Ale
Barley Griffin	Bentspoke	Amber IPA
Berserker	Ekim	Double Red Ale
Betelgeuse	Kaiju	Dark IPA
Big Nut	Bentspoke	Barleywine
Bigfoot	Sierra Nevada	
(15 rows)		

## ❖ SQL Sample Database (cont)

Sample data about drinkers ..

```
beer=# select * from Drinkers;
  name    |      addr      |      phone
-----+-----+-----+
 Adam   | Randwick     | 9385-4444
 Gernot  | Newtown      | 9415-3378
 John    | Alexandria   | 9665-1234
 Andrew   | Clovelly     | 9123-1234
 Justin   | Mosman       | 9845-4321
 Helen    | Coogee        | 9876-5432
(6 rows)
```

## ❖ SQL Sample Database (cont)

Sample data about drinkers ..

```
beer=# select * from Bars;
      name          |      addr        | license
-----+-----+-----+
Australia Hotel | The Rocks    | 123456
Coogee Bay Hotel | Coogee       | 966500
Lord Nelson     | The Rocks    | 123888
Marble Bar       | Sydney       | 122123
Regent Hotel     | Kingsford   | 987654
Royal Hotel      | Randwick    | 938500
Local Taphouse   | Darlinghurst | 884488
(7 rows)
```

---

Produced: 27 Sep 2020

# SQL: Updating the Data

---

- Data Modification in SQL
- Insertion
- Bulk Insertion of Data
- Deletion
- Semantics of Deletion
- Updates

## ❖ Data Modification in SQL

We have seen statements to modify table meta-data (in DB catalog):

- **CREATE TABLE** ... add new, initially empty, table to DB
- **DROP TABLE** ... remove table data (all tuples) and meta-data
- **ALTER TABLE** ... change meta-data of table (e.g add constraints)

SQL also provides statements for modifying data in tables:

- **INSERT** ... add a new tuple(s) into a table
- **DELETE** ... remove tuples from a table (via condition)
- **UPDATE** ... modify values in existing tuples (via condition)

Constraint checking is applied automatically on any change.

Operation fails (no change to DB) if any constraint check fails

## ❖ Insertion

Add new tuples via the **INSERT** operation:

```
INSERT INTO RelationName
VALUES (val1, val2, val3, ...)
```

```
INSERT INTO RelationName(Attr1, Attr2, ...)
VALUES (valForAttr1, valForAttr2, ...)
```

```
INSERT INTO RelationName
VALUES Tuple1, Tuple2, Tuple3, ...
```

The first two add a single new tuple into *RelationName*.

The last form adds multiple tuples into *RelationName*.

## ❖ Insertion (cont)

**INSERT INTO  $R$  VALUES ( $v_1, v_2, \dots$ )**

- values must be supplied for all attributes of  $R$
- in same order as appear in **CREATE TABLE** statement
- special value **DEFAULT** forces default value or **NULL**

**INSERT INTO  $R(A_1, A_2, \dots)$  VALUES ( $v_1, v_2, \dots$ )**

- can specify any subset of attributes of  $R$
- values must match attribute specification order
- unspecified attributes are assigned default or null

## ❖ Insertion (cont)

**Example:** Add the fact that Justin likes 'Old'.

```
INSERT INTO Likes VALUES ('Justin', 'Old');  
-- or --  
INSERT INTO Likes(drinker,beer) VALUES('Justin', 'Old');  
-- or --  
INSERT INTO Likes(beer,drinker) VALUES('Old', 'Justin');
```

**Example:** Add a new beer with unknown style.

```
INSERT INTO Beers(name,brewer)  
    VALUES('Mysterio', 'Hop Nation');  
-- which inserts the tuple ...  
( 'Mysterio', 'Hop Nation', null)
```

## ❖ Insertion (cont)

**Example:** insertion with default values

```
ALTER TABLE Likes
    ALTER COLUMN beer SET DEFAULT 'New';
ALTER TABLE Likes
    ALTER COLUMN drinker SET DEFAULT 'Joe';

INSERT INTO Likes(drinker) VALUES('Fred');
INSERT INTO Likes(beer) VALUES('Sparkling Ale');

-- inserts the two new tuples ...
('Fred', 'New')
('Joe', 'Sparkling Ale')
```

## ❖ Insertion (cont)

**Example:** insertion with insufficient values.

E.g. specify that drinkers' phone numbers cannot be **NULL**.

```
ALTER TABLE Drinkers  
    ALTER COLUMN phone SET NOT NULL;
```

Then try to insert a drinker whose phone number we don't know:

```
INSERT INTO Drinkers(name,addr) VALUES ('Zoe','Manly');  
ERROR: null value in column "phone" violates  
          not-null constraint  
DETAIL: Failing row contains (Zoe, Manly, null).
```

## ❖ Bulk Insertion of Data

Tuples may be inserted individually:

```
insert into Stuff(x,y,s) values (2,4,'green');
insert into Stuff(x,y,s) values (4,8,null);
insert into Stuff(x,y,s) values (8,null,'red');
...
```

but this is tedious if 1000's of tuples are involved.

It is also inefficient

- all relevant constraints are checked on insertion of each tuple

So, most DBMSs provide non-SQL methods for bulk insertion

## ❖ Bulk Insertion of Data (cont)

Bulk insertion methods typically ...

- use a compact representation for each tuple
- "load" all tuples without constraint checking
- do all constraint checks at the end
- if any tuples fail checks, none are inserted

Example: PostgreSQL's **copy** statement:

```
COPY Stuff(x,y,s) FROM stdin;
2      4      green
4      8      \N
8      \N     red
\.
```

Can also copy from a named file (but must be readable by PostgreSQL server)

## ❖ Deletion

Removing tuples is accomplished via **DELETE** statement:

```
DELETE FROM Relation
WHERE Condition
```

Removes all tuples from *Relation* that satisfy *Condition*.

**Example:** Justin no longer likes Sparkling Ale.

```
DELETE FROM Likes
WHERE drinker = 'Justin'
      AND beer = 'Sparkling Ale';
```

**Special case:** Make relation *R* empty.

```
DELETE FROM R;      or      DELETE FROM R WHERE true;
```

## ❖ Deletion (cont)

**Example:** remove all expensive beers from sale.

```
DELETE FROM Sells WHERE price >= 5.00;
```

**Example:** remove all beers with unknown style

```
DELETE FROM Beers WHERE style IS NULL;
```

This fails\* if such Beers are referenced from other tables

E.g. such Beers are liked by someone or sold in some bar

\* no beers are removed, even if some are not referenced

## ❖ Semantics of Deletion

Method A for **DELETE FROM R WHERE Cond:**

```
FOR EACH tuple T in R DO
    IF T satisfies Cond THEN
        remove T from relation R
    END
END
```

Method B for **DELETE FROM R WHERE Cond:**

```
FOR EACH tuple T in R DO
    IF T satisfies Cond THEN
        make a note of this T
    END
END
FOR EACH noted tuple T DO
    remove T from relation R
END
```

## ❖ Semantics of Deletion (cont)

Does it matter which method the DBMS uses?

For most cases, the same tuples would be deleted

But if *Cond* invokes a query on the table *R*

- the result of *Cond* might change as the deletion progresses
- so Method A might delete less tuples than Method B

E.g.

```
DELETE FROM Beers  
WHERE (SELECT count(*) FROM Beers) > 10;
```

Method A deletes beers until there are only 10 left

Method B deletes all beers if there were more than 10 to start with

## ❖ Updates

The **UPDATE** statement allows you to ...

- modify values of specified attributes in specified tuples of a relation

```
UPDATE R
SET    List of assignments
WHERE   Condition
```

Each tuple in relation  $R$  that satisfies  $Condition$  is affected

Assignments may:

- assign constant values to attributes,  
e.g. **SET price = 2.00**
- use existing values in the tuple to compute new values,  
e.g. **SET price = price \* 0.5**

## ❖ Updates (cont)

**Example:** Adam changes his phone number.

```
UPDATE Drinkers
SET    phone = '9385-2222'
WHERE   name = 'Adam';
```

**Example:** John moves to Coogee.

```
UPDATE Drinkers
SET    addr = 'Coogee',
      phone = '9665-4321'
WHERE   name = 'John';
```

## ❖ Updates (cont)

Examples that modify many tuples ...

**Example:** Make \$6 the maximum price for beer.

```
UPDATE Sells  
SET    price = 6.00  
WHERE  price > 6.00;
```

**Example:** Increase beer prices by 10%.

```
UPDATE Sells  
SET    price = price * 1.10;
```

Updates all tuples (as if **WHERE true**)

---

Produced: 28 Sep 2020

# SQL: Queries on One Table

- Queries
- SQL Query Language
- Problem-solving in SQL
- Views
- Exercise: Queries on Beer Database

## ❖ Queries

A **query** is a **declarative program** that retrieves data from a database.

declarative = say what we want, not method to get it

Queries are used in two ways in RDBMSs:

- interactively (e.g. in **psql**)
  - the entire result is displayed in tabular format on the output
- by a program (e.g. in a PLpgSQL function)
  - the result tuples are consumed one-at-a-time by the program

SQL is based on the **relational algebra**, which we discuss elsewhere

## ❖ SQL Query Language

An SQL **query** consists of a sequence of clauses:

```
SELECT      projectionList
FROM        relations/joins
WHERE       condition
GROUP BY   groupingAttributes
HAVING     groupCondition
```

**FROM, WHERE, GROUP BY, HAVING** clauses are optional.

Result of query: a relation, typically displayed as a table.

Result could be just one tuple with one attribute (i.e. one value) or even empty

## ❖ SQL Query Language (cont)

Functionality provided by SQL ...

**Filtering:** extract attributes from tuples, extract tuples from tables

*condition for each single tuple.*

```
SELECT b,c FROM R(a,b,c,d) WHERE a > 5
```

**Combining:** merging related tuples from different tables

```
... FROM R(x,y,z) JOIN S(a,b,c) ON R.y = S.a
```

**Summarising:** aggregating values in a single column

```
SELECT avg(mark) FROM ...
```

**Set operations:** union, intersection, difference

## ❖ SQL Query Language (cont)

More functionality provided by SQL ...

**Grouping:** forming subsets of tuples sharing some property

```
... GROUP BY R.a
```

(forms groups of tuples from **R** sharing the same value of **a**)

**Group Filtering:** selecting only groups satisfying a condition

```
... GROUP BY R.a HAVING max(R.a) < 75
```

**Renaming:** assign a name to a component of a query

```
SELECT a as name
FROM Employee(a,b,c) e WHERE e.b > 50000
```

## ❖ SQL Query Language (cont)

Schema:

- *Students(id, name, ...)*
- *Enrolments(student, course, mark, grade)*

Example SQL query on this schema:

```
SELECT      s.id, s.name, avg(e.mark) as avgMark
FROM        Students s
            JOIN Enrolments e on (s.id = e.student)
GROUP BY    s.id, s.name
-- or --
SELECT      s.id, s.name, avg(e.mark) as avgMark
FROM        Students s, Enrolments e
WHERE       s.id = e.student
GROUP BY    s.id, s.name
```

## ❖ SQL Query Language (cont)

How the example query is computed:

- produce all pairs of *Students*,*Enrolments* tuples which satisfy condition (*Students.id* = *Enrolments.student*)
- each tuple has (*id*,*name*,...,*student*,*course*,*mark*,*grade*)
- form groups of tuples with same (*id*,*name*) values
- for each group, compute average mark
- form result tuples (*id*,*name*,*avgMark*)

## ❖ Problem-solving in SQL

Starts with an information request:

- (informal) description of the information required from the database

Ends with:

- a list of tuples that meet the requirements in the request

Pre-req: [know your schema](#)

Look for keywords in request to identify required data :

- tell me the **names** of all **students**...
- **how many students failed** ...
- what is the **highest mark** in ...
- which **courses** are ... (course codes?)

## ❖ Problem-solving in SQL (cont)

Developing SQL queries ...

- relate required data to **attributes** in schema
- identify which **tables** contain these attributes
- combine data from relevant tables (**FROM, JOIN**)
- specify conditions to select relevant data (**WHERE**)
- [optional] define grouping attributes (**GROUP BY**)
- develop expressions to compute output values (**SELECT**)

## ❖ Problem-solving in SQL (cont)

**Example:** just the beers that John likes

- which table contains info about beers that are liked?
- **Likes (drinker, beers)**
- only want tuples where drinker is John (**WHERE**)
- only want beer names (**SELECT beer**)

... giving ...

```
select beer from Likes where drinker='John' ;
```

## ❖ Views

A **view** associates a name with a query:

- **CREATE VIEW** *viewName* [ ( *attributes* ) ] **AS** *Query*

Each time the view is invoked (in a **FROM** clause):

- the *Query* is evaluated, yielding a set of tuples
- the set of tuples is used as the value of the view

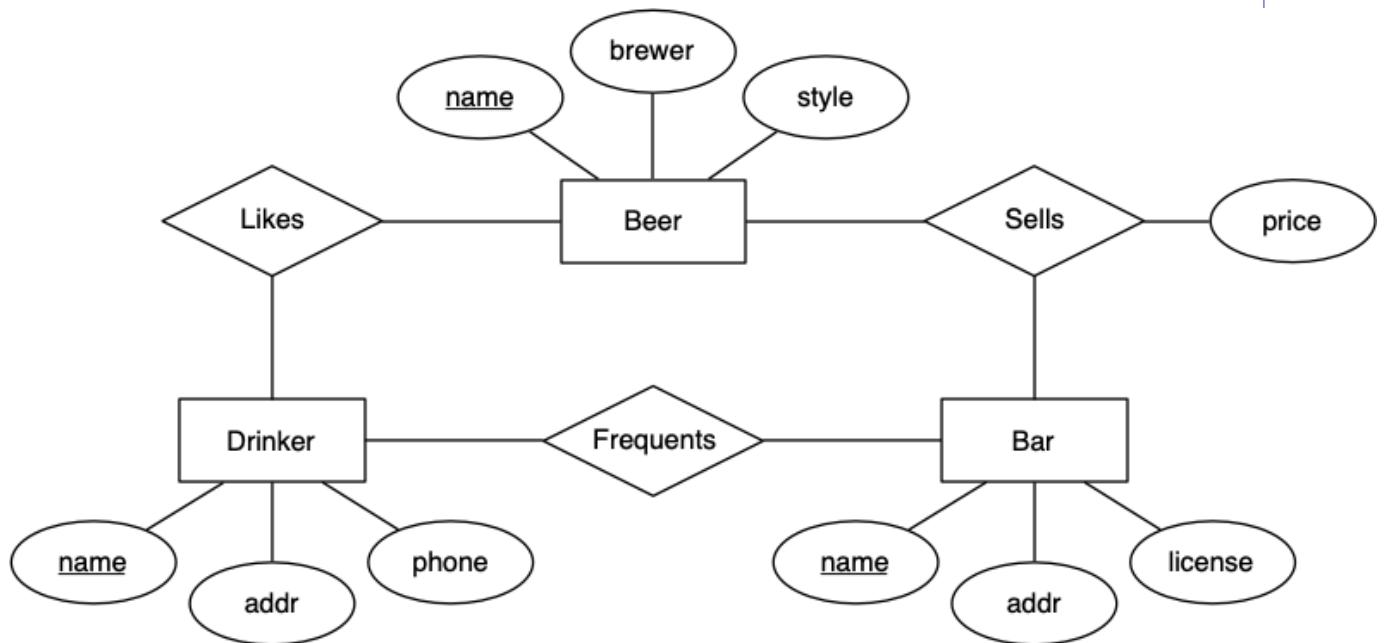
A view can be treated as a "virtual table".

Views are useful for "packaging" a complex query to use in other queries.

cf. writing functions to package computations in programs

## ❖ Exercise: Queries on Beer Database

ER design for Beer database:



## ❖ Exercise: Queries on Beer Database

(cont)

Answer these queries on the Beer database:

1. What beers are made by Toohey's?
2. Show beers with headings "Beer", "Brewer".
3. How many different beers are there?
4. How many different brewers are there?
5. Which beers does John like?
6. Find pairs of beers by the same manufacturer.
7. How many beers does each brewer make?
8. Which brewers make only one beer?
9. Which brewer makes the most beers?

\d ← See detail.

COMP3311 20T3 ◇ SQL: Queries on One Table ◇ [12/12]

1. select name from Beers where brewer = 'Toohey's';

for reading purpose.  
↓

2. select name as "Beer", brewer as "Brewer" from Beers order by name;

1st col                    2nd col.

3. select count(\*)  
      name  
      brewer from Beers;

4. select count(distinct brewer) from Beers;

↑  
keyword

5. select beer from Likes where drinker = 'John';

6. select b1.name, b2.name, b2.brewer  
from Beers b1, Beers b2  
where b1.brewer = b2.brewer and  
b1.name < b2.name.

"\e" editor.

7. select brewer, count(\*) from beers group by brewer.

8. select brewer, count(\*)

```
from beers  
group by brewer  
having count(*) = 1  
order by brewer;
```

9. create view beers2 (brewer, nbeers)

as

```
select brewer, count(*)
```

```
from beers  
group by brewer;
```

---

```
select * from beers2;
```

---

```
select max(nbeers) from beers2;
```

---

```
select brewer from beers2
```

```
where nbeers = (select max(nbeers) from beers2);
```

# SQL: Views

---

- Views
- Renaming View Attributes
- Using Views
- Updating Views
- Evaluating Views
- Materialized Views

## ❖ Views

A **view** is like a "virtual relation" defined via a query.

View definition and removal:

**CREATE VIEW** *ViewName AS Query*

**CREATE VIEW** *ViewName (AttributeNames) AS Query*

**DROP VIEW** *ViewName*

*changing the table name only.*

*Query* may be any SQL query, involving: stored tables, other views

**CREATE OR REPLACE** replaces the *Query* associated with a view

## ❖ Views (cont)

The stored tables used by a view are referred to as **base tables**.

Views are defined only after their base tables are defined.

A view is valid only as long as its underlying query is valid.

Dropping a view has no effect on the base tables.

Views are a convenient abstraction mechanism

- allow you to package and name complex queries
- give you the "table that you wanted" to solve a more complex query

## ❖ Views (cont)

**Example:** defining/naming a complex query using a view:

```
CREATE VIEW
    CourseMarksAndAverages(course,term,student,mark,avg)
AS
SELECT s.code, termName(t.id), e.student, e.mark,
       avg(mark) OVER (PARTITION BY course)
FROM   CourseEnrolments e
       JOIN Courses c on c.id = e.course
       JOIN Subjects s on s.id = c.subject
       JOIN Terms t on t.id = c.term
;
```

which would make the following query easy to solve

```
SELECT course, term, student, mark
FROM   CourseMarksAndAverages
WHERE  mark < avg;
```

## ❖ Views (cont)

**Example:** An avid Carlton drinker might not be interested in other kinds of beer.

```
CREATE VIEW MyBeers AS  
    SELECT * FROM Beers WHERE brewer = 'Carlton';
```

which is used as

```
SELECT * FROM MyBeers;
```

name	brewer	style
Crown Lager	Carlton	Lager
Fosters Lager	Carlton	Lager
Invalid Stout	Carlton	Stout
Melbourne Bitter	Carlton	Lager
Victoria Bitter	Carlton	Lager

## ❖ Views (cont)

A view might not use all attributes of the base relations.

**Example:** We don't really need the address of inner-city hotels.

```
CREATE VIEW InnerCityHotels AS
    SELECT name, license
    FROM Bars
    WHERE addr in ('The Rocks', 'Sydney');
```

```
SELECT * FROM InnerCityHotels;
```

name	license
Australia Hotel	123456
Lord Nelson	123888
Marble Bar	122123

## ❖ Views (cont)

A view might use computed attribute values.

**Example:** Number of beers produced by each brewer.

```
CREATE VIEW BeersBrewed AS
    SELECT brewer, count(*) as nbeers
    FROM   beers GROUP BY brewer;

SELECT * FROM BeersBrewed;

+-----+-----+
| brewer | nbeers |
+-----+-----+
| 3 Ravens | 1 |
| Akasha | 1 |
| Alesmith | 1 |
| ... |

```

## ❖ Renaming View Attributes

This can be achieved in two different ways:

```
CREATE VIEW InnerCityHotels AS
    SELECT name AS bar, license AS lic
    FROM Bars
    WHERE addr IN ('The Rocks', 'Sydney');

CREATE VIEW InnerCityHotels(bar, lic) AS
    SELECT name, license
    FROM Bars
    WHERE addr IN ('The Rocks', 'Sydney');
```

Both of the above produce the same view.

## ❖ Using Views

Views can be used in queries as if they were stored relations.

However, they differ from stored relations in two important respects:

- their "value" can change without being explicitly modified (i.e. the result of a view may change whenever one of its base tables is updated)
- they may not be able to be explicitly modified (updated) (only a certain simple kinds of views can be explicitly updated)

"Modifying a view" means changing the base tables via the view, e.g.

```
insert into MyBeers values ('Zero', 'Carlton', 'No-alcohol');
```

would update the **Beers** table

## ❖ Using Views (cont)

**Example:** of view changing when base table changes.

```
SELECT * FROM InnerCityHotels;
      name      | license
-----+-----
Australia Hotel | 123456
Lord Nelson    | 123888
Marble Bar     | 122123

-- then the Lord Nelson goes broke
DELETE FROM Bars WHERE name = 'Lord Nelson';

-- no explicit update has been made to InnerCityHotels
SELECT * FROM InnerCityHotels;
      name      | license
-----+-----
Australia Hotel | 123456
Marble Bar     | 122123
```

## ❖ Updating Views

Explicit updates are allowed on views satisfying the following:

- the view involves a single relation **R**
- the **WHERE** clause does not involve **R** in a subquery
- the **WHERE** clause only uses attributes from the **SELECT**

Attributes not in the view's **SELECT** will be set to **NULL** in the base relation after an insert into the view.

## ❖ Updating Views (cont)

**Example:** Our **InnerCityHotel** view is not updatable.

```
INSERT INTO InnerCityHotels  
VALUES ('Jackson''s on George', '9876543');
```

creates a new tuple in the **Bars** relation:

```
(Jackson's on George, NULL, 9876543)
```

but this new tuple does not satisfy the view condition:

```
addr IN ('The Rocks', 'Sydney')
```

so it does not appear if we select from the view.

## ❖ Evaluating Views

Two alternative ways of implementing views:

- re-writing rules (or macros)
  - when a view is used in a query, the query is re-written
  - after rewriting, becomes a query only on base relations
- explicit stored relations (called **materialized views**)
  - the view is stored as a real table in the database
  - updated appropriately when base tables are modified

The difference: underlying query evaluated either at query time or at update time.

## ❖ Evaluating Views (cont)

**Example:** Using the **InnerCityHotels** view.

```
CREATE VIEW InnerCityHotels AS
    SELECT name, license
    FROM Bars
    WHERE addr IN ('The Rocks', 'Sydney');

SELECT name
FROM InnerCityHotels
WHERE license = '123456';

--is rewritten into the following form before execution

SELECT name
FROM Bars
WHERE addr IN ('The Rocks', 'Sydney')
    AND license = '123456';
```

## ❖ Materialized Views



Materialized views are implemented as stored tables

On each update to base tables, need to also update the view table.

Clearly this costs space and makes updates more expensive.

However, in a situation where

- updates are infrequent compared to queries on the view
- the cost of "computing" the view is expensive

this approach provides substantial benefits.

Materialized views are used extensively in data warehouses.

---

Produced: 29 Sep 2020

# SQL: Queries on Multiple Tables

---

- Queries on Multiple Tables
- Join
- Name Clashes in Conditions
- Explicit Tuple Variables
- Outer Join
- Subqueries

## ❖ Queries on Multiple Tables

Queries involving a single table are useful.

Exploiting all data in the DB requires

- combining data from multiple tables
- typically involving primary/foreign key matching

**Example:** Which brewers makes beers that John likes?

```
select b.brewer
from Beers b join Likes L on (b.name = L.beer)
where L.drinker = 'John';
```

Info on brewers is in **Beers**; info on who likes what in **Likes**.

Need to combine info from both tables using "common" attributes

## ❖ Queries on Multiple Tables (cont)

Example **Beers** and **Likes** tuples:

Beers(80/-, Caledonian, Scotch Ale)	Likes(John, Sculpin)
Beers(New, Toohey's, Lager)	Likes(John, Red Nut)
Beers(Red Nut, Bentspoke, Red IPA)	Likes(Adam, New)
Beers(Sculpin, Ballast Point, IPA)	Likes(John, 80/-)

"Merged" tuples resulting from

- **Beers b join Likes L on (b.name = L.beer)**

```
Joined(80/-, Caledonian, Scotch Ale, John, 80/-)
Joined(New, Toohey's, Lager, Adam, New)
Joined(Red Nut, Bentspoke, Red IPA, John, Red Nut)
Joined(Sculpin, Ballast Point, IPA, John, Sculpin)
```

In the query, the **where** clause removes all tuples not related to John

## ❖ Join

Join is the SQL operator that combines tuples from tables.

Such an important operation that several variations exist

- **natural join** matches tuples via equality on common attributes
- **equijoin** matches tuples via equality on specified attributes
- **theta-join** matches tuples via a boolean expression
- **outer join** like theta-join, but includes non-matching tuples

We focus on theta-join and outer join in this course

## ❖ Join (cont)

Join fits into **SELECT** queries as follows:

```
SELECT Attributes
FROM   R1
        JOIN R2 ON (JoinCondition1)
        JOIN R3 ON (JoinCondition2)
        ...
WHERE  Condition
```

Can include an arbitrary number of joins.

**WHERE** clause typically filters out some of the joined tuples.

## ❖ Join (cont)

Alternative syntax for joins:

```
SELECT brewer
FROM Likes L, Beers b
WHERE L.beer = b.name
      AND L.drinker = 'John';
```

Join condition(s) are specified in the **WHERE** clause

We prefer the explicit **JOIN** syntax, but this is sometimes more compact

Note: duplicates could be eliminated by using **distinct**

## ❖ Join (cont)

Operational semantics of **R1 JOIN R2 ON (Condition)**:

```

FOR EACH tuple t1 in R1 DO
    FOR EACH tuple t2 in R2 DO
        check Condition for current
        t1, t2 attribute values
        IF Condition holds THEN
            add (t1,t2) to result
        END
    END
END

```

Easy to generalise: add more relations, include **WHERE** condition

Requires one tuple variable for each relation, and nested loops over relations.  
But this is **not** how it's actually computed!

## ❖ Name Clashes in Conditions

If a **SELECT** statement

- refers to multiple tables
- some tables have attributes with the same name

use the table name to disambiguate.

**Example:** Which hotels have the same name as a beer?

```
SELECT Bars.name
FROM   Bars, Beers
WHERE  Bars.name = Beers.name;
-- or, using table aliases ...
SELECT r.name
FROM   Bars r, Beers b
WHERE  r.name = b.name
```

## ❖ Explicit Tuple Variables

Table-dot-attribute doesn't help if we use same table twice in **SELECT**.

To handle this, define new names for each "instance" of the table

```
SELECT r1.a, r2.b FROM R r1, R r2 WHERE r1.a = r2.a
```

**Example:** Find pairs of beers by the same manufacturer.

```
SELECT b1.name, b2.name
FROM Beers b1 JOIN Beers b2 ON (b1.brewer = b2.brewer)
WHERE b1.name < b2.name;
```

The **WHERE** condition is used to avoid:

- pairing a beer with itself e.g. (**New, New**)
- same pairs with different order e.g. (**New, Old**) (**Old, New**)

## ❖ Outer Join

Join only produces a result tuple from  $t_R$  and  $t_S$  where ...

- there are appropriate values in both tuples
- so that the join condition is satisfied

```
SELECT * FROM R JOIN S WHERE (Condition)
```

Sometimes, we want a result for *every* **R** tuple

- even if some **R** tuples have no matching **S** tuple

These kinds of requests often include "for each" or "for every"

## ❖ Outer Join (cont)

**Example:** for each suburb with a bar, find out who drinks there.

Theta-join only gives results for suburbs where people drink.

```
SELECT B.addr, F.drinker
FROM   Bars B
       JOIN Frequent F ON (F.bar = B.name)
ORDER  BY addr;
```

addr	drinker
Coogee	Adam
Coogee	John
Kingsford	Justin
Sydney	Justin
The Rocks	John

But what if we want all suburbs, even if some have no drinkers?

This is from an older and simpler instance of the beers database.

## ❖ Outer Join (cont)

An **outer join** solves this problem.

For  $R \text{ OUTER JOIN } S \text{ ON } (\text{Condition})$

- all "tuples" in  $R$  have an entry in the result
- if a tuple from  $R$  matches tuples in  $S$ , we get the normal join result tuples
- if a tuple from  $R$  has no matches in  $S$ , the attributes supplied by  $S$  are **NULL**

This outer join variant is called **LEFT OUTER JOIN**.

## ❖ Outer Join (cont)

Solving the example query with an outer join:

```
SELECT B.addr, F.drinker
FROM Bars B
    LEFT OUTER JOIN Frequent F on (F.bar = B.name)
ORDER BY B.addr;
```

addr	drinker
Coogee	Adam
Coogee	John
Kingsford	Justin
Randwick	
Sydney	Justin
The Rocks	John

Note that Randwick is now mentioned (because of the Royal Hotel).

## ❖ Outer Join (cont)

Operational semantics of **R1 LEFT OUTER JOIN R2 ON (Cond)**:

```
FOR EACH tuple t1 in R1 DO
    nmatches = 0
    FOR EACH tuple t2 in R2 DO
        check Cond for current
        t1, t2 attribute values
        IF Cond holds THEN
            nmatches++
            add (t1,t2) to result
        END
    END
    IF nmatches == 0 THEN
        t2 = (null,null,null,...)
        add (t1,t2) to result
    END
END
```

## ❖ Outer Join (cont)

Many RDBMSs provide three variants of outer join:

- $R \text{ LEFT OUTER JOIN } S$ 
  - behaves as described above
- $R \text{ RIGHT OUTER JOIN } S$ 
  - includes all tuples from  $S$  in the result
  - **NULL**-fills any  $S$  tuples with no matches in  $R$
- $R \text{ FULL OUTER JOIN } S$ 
  - includes all tuples from  $R$  and  $S$  in the result
  - those without matches in other relation are **NULL**-filled

## ❖ Subqueries

The result of a query can be used in the **WHERE** clause of another query.

**Case 1:** Subquery returns a single, unary tuple

```
SELECT * FROM R WHERE R.a = (SELECT S.x FROM S WHERE Cond1)
```

**Case 2:** Subquery returns multiple values

```
SELECT * FROM R WHERE R.a IN (SELECT S.x FROM S WHERE Cond2)
```

This approach is often used in the initial discussion of SQL in some textbooks.

These kinds of queries can generally be solved *more efficiently* using a join

```
SELECT * FROM R JOIN S ON (R.a = S.x) WHERE Cond
```



## SQL Queries (iii)

- Sets in SQL
- Bags in SQL
- The **IN** Operator
- The **EXISTS** Operator
- Quantifiers
- Union, Intersection, Difference
- Division
- Selection with Aggregation

## ❖ Sets in SQL

The relational model is set-based

Set literals are written as `( expr1, expr2, ... )` (each *expr<sub>i</sub>* yields an atomic value)

SQL query results are (more or less) sets of tuples or atomic values

Examples:

```
-- set literals
(1,2,3)      ('a','b','c','d')
-- set of atomic values
(select salary from Employees)
-- set of tuple values
(select id, name from Employees)
```

SQL provides a variety of set-based operators:  $\in$ ,  $\cup$ ,  $\cap$ ,  $\neg$ ,  $/$ ,  $\exists$ ,  $\forall$ , ...

## ❖ Bags in SQL

SQL query results are actually **bags** (multisets), allowing duplicates, e.g.

```
select age from Students;
-- yields (18,18,18,...19,19,19,19,...20,20,20,...)
```

Can convert bag to set (eliminate duplicates) using **DISTINCT**, e.g

```
select distinct age from Students;
```

SQL set operations **UNION**, **INTERSECT**, **EXCEPT** ...

- yield sets by default (i.e. eliminate duplicates)
- can produce bags with keyword **ALL** (e.g. **UNION ALL**)

```
(1,2,3) UNION (2,3,4) yields (1,2,3,4)
(1,2,3) UNION ALL (2,3,4) yields (1,2,3,2,3,4)
```

Bars where either "John" or "Gernot" visit.

Bars where both "John" or "Gernot" visit.

```
(select bar from frequents where drinker = 'John')
intersect
(select bar from frequents where drinker = 'Gernot');
```

Similar for except.

## ❖ The IN Operator

Tests whether a specified tuple is contained in a relation  
(i.e.  $t \in R$ )

*tuple IN relation* is true iff the tuple is contained in the relation.

Conversely for *tuple NOT IN relation*.

Syntax:

```
SELECT *
FROM   R
WHERE  R.a IN (SELECT x FROM S WHERE Cond)
        -- assume multiple results
```

## ❖ The IN Operator (cont)

**Example:** Find the name and brewer of beers that John likes.

```
SELECT name, brewer
FROM Beers
WHERE name IN
    (SELECT beer
     FROM Likes
     WHERE drinker = 'John');
```

name	brewer
80/-	Caledonian
Bigfoot Barley Wine	Sierra Nevada
Pale Ale	Sierra Nevada
Three Sheets	Lord Nelson

Subquery = "What are the names of the beers that John likes?"

(This and subsequent beer queries use an older smaller version of the Beer database)

## ❖ The EXISTS Operator

**EXISTS** (*relation*) is true iff the relation is non-empty.

**Example:** Find the beers that are the unique beer by their manufacturer.

```
SELECT name, brewer
FROM Beers b1
WHERE NOT EXISTS
  (SELECT *
   FROM Beers b2
   WHERE b2.brewer = b1.brewer
     AND b2.name <> b1.name);
```

A subquery that refers to values from a surrounding query is called a **correlated subquery**.

## ❖ Quantifiers

**ANY** and **ALL** behave as existential and universal quantifiers respectively.

**Example:** Find the beers sold for the highest price.

```
SELECT beer
FROM   Sells
WHERE  price >=
        ALL(SELECT price FROM sells);
```

Beware: in common use, "any" and "all" are often synonyms.

E.g. "I'm better than any of you" vs. "I'm better than all of you".

## ❖ Union, Intersection, Difference

SQL implements the standard set operations

$R1 \text{ UNION } R2$  set of tuples in either  $R1$  or  $R2$

$R1 \text{ INTERSECT } R2$  set of tuples in both  $R1$  and  $R2$

$R1 \text{ EXCEPT } R2$  set of tuples in  $R1$  but not  $R2$

**R1** and **R2** must be union-compatible (i.e. same schema)  
number to number.

Union and intersection semantics are straightforward.

## ❖ Union, Intersection, Difference (cont)

**Example:** Find the drinkers and beers such that the drinker likes the beer and frequents a bar that sells it.

```
(SELECT drinker, beer FROM Likes)
INTERSECT
(SELECT drinker,beer
 FROM Sells natural join Frequents);
```

drinker	beer
Adam	New
John	Three Sheets
Justin	Victoria Bitter

## ❖ Union, Intersection, Difference (cont)

Set difference is implemented by **EXCEPT**

**R**

A	B
1	'a'
2	'b'
3	'a'

**S**

A	B
1	'a'
1	'b'
2	'a'

**R except S**

A	B
2	'b'
3	'a'

**S except R**

A	B
1	'b'
2	'a'

Semantics of set difference:  $R \text{ except } S = \{ x \in R, \text{ where } x \notin S \}$

## ❖ Division

Division aims to find values in one table that occur in conjunction with all values in another table:

**R**

<b>A</b>	<b>B</b>
1	'a'
2	'b'
3	'a'
1	'b'
2	'a'

**S**

<b>B</b>
'a'
'b'

**R / S**

<b>A</b>
1
2

Arises in queries like "Find Xs that are related to all Ys / every Y"

## ❖ Division (cont)

Not all SQL implementations provide a division operator

But can be achieved by combination of existing operations

**Example:** Find bars that each sell all of the beers Justin likes.

```
SELECT DISTINCT a.bar
FROM   Sells a
WHERE  NOT EXISTS (
    (SELECT beer FROM Likes
     WHERE drinker = 'Justin')
    EXCEPT
    (SELECT beer FROM Sells b
     WHERE bar = a.bar)
) ;
```

## ❖ Selection with Aggregation

Selection clauses can contain aggregation operations.

**Example:** What is the average price of New?

```
SELECT AVG(price)
FROM   Sells
WHERE  beer = 'New';
```

avg

-----  
2.38749998807907

- the bag semantics of SQL gives the correct result here
- the price for New in all hotels will be included, even if two hotels sell it at the same price
- if we used set semantics, we'd get the average of all the **different** prices for New.

## ❖ Selection with Aggregation (cont)

If we want set semantics, can force using **DISTINCT**.

**Example:** How many different bars sell beer?

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

count
-----
6
```

Without **DISTINCT**, counts number of entries in the **Sells** table.

Aggregation operators on numbers: **SUM**, **AVG**, **MIN**, **MAX**,

---

Produced: 4 Oct 2020

## SQL Queries (iv): Grouping

- Grouping
- Restrictions on **SELECT** Lists
- Filtering Groups
- Partitions

## ❖ Grouping

**SELECT-FROM-WHERE** can be followed by **GROUP BY** to:

- partition result relation into groups (according to values of specified attribute)
- summarise (aggregate) some aspects of each group
- output one tuple per group, with grouping attribute and aggregates

**R**

A	B
1	'a'
2	'b'
3	'a'
1	'b'
2	'a'
1	'c'

**R group by A**

A	B
1	'a'
1	'b'
1	'c'
2	'b'
2	'a'
3	'a'

**A, count(\*), max(B)**

A	count	max
1	3	'c'
2	2	'b'
3	1	'a'

## ❖ Grouping (cont)

**Example:** How many different beers does each brewer make?

```
SELECT    brewer, COUNT(name) as nbeers
FROM      Beers
GROUP BY  brewer;
```

brewer	nbeers
West City	1
James Squire	5
Yullis	1
Hop Nation	4
Anderson Valley	1
Beatnik	1
BoatrockeR	3
Kizakura	1
...	

## ❖ Grouping (cont)

**GROUP BY** is used as follows:

```
SELECT      attributes/aggregations
FROM        relations
WHERE       condition
GROUP BY   attributes
```

Semantics:

1. apply product and selection as for **SELECT-FROM-WHERE**
2. partition result into groups based on values of *attributes*
3. apply any aggregation separately to each group

Grouping is typically used in queries involving the phrase "for each".

## ❖ Restrictions on **SELECT** Lists

When using grouping, every attribute in the **SELECT** list must:

- have an aggregation operator applied to it OR
- appear in the **GROUP-BY** clause

**Incorrect Example:** Find the styles associated with each brewer

```
SELECT    brewer, style
FROM      Beers
GROUP BY brewer;
```

PostgreSQL's response to this query:

```
ERROR: column beers.style must appear in the GROUP BY
      clause or be used in an aggregate function
```

## ❖ Filtering Groups

In some queries, you can use the **WHERE** condition to eliminate groups.

**Example:** Average beer price by suburb excluding hotels in The Rocks.

```
SELECT      b.addr, AVG(s.price)
FROM        Sells s join Bars b on (s.bar=b.name)
WHERE       b.addr <> 'The Rocks'
GROUP BY    b.addr;
```

For conditions on whole groups, use the **HAVING** clause.

## ❖ Filtering Groups (cont)

**HAVING** is used to qualify a **GROUP-BY** clause:

```
SELECT      attributes/aggregations  
FROM        relations  
WHERE       condition1    (on tuples)  
GROUP BY   attributes  
HAVING     condition2;  (on group)
```

Semantics of **HAVING**:

*where → Tuples.  
having → group.*

1. generate the groups as for **GROUP-BY**
2. discard groups **not** satisfying **HAVING** condition
3. apply aggregations to remaining groups

## ❖ Filtering Groups (cont)

**Example:** Number of styles from brewers who make at least 5 beers?

```
SELECT    brewer, count(name) as nbeers,
          count(distinct style) as nstyles
FROM      Beers
GROUP BY brewer
HAVING   count(name) > 4
ORDER BY brewer;
```

brewer	nbeers	nstyles
Bentspoke	9	7
Carlton	5	2
Frenchies	5	5
Hawkers	5	5
James Squire	5	4
One Drop	9	7
Sierra Nevada	5	5
Tallboy and Moose	5	5

**distinct** required, otherwise **nbeers=nstyles** for all brewers

## ❖ Filtering Groups (cont)

Alternative formulation of division using **GROUP-BY** and **HAVING**

**Example:** Find bars that each sell all of the beers Justin likes.

```
SELECT DISTINCT S.bar
FROM   Sells S, Likes L on (S.beer = L.beer)
WHERE  L.drinker = 'Justin'
GROUP  BY S.bar
HAVING count(S.beer) =
       (SELECT count(beer) FROM Likes
        WHERE drinker = 'Justin');
```

## ❖ Partitions

Sometimes it is useful to

- partition a table into groups
- compute results that apply to each group
- use these results with individual tuples in the group

Comparison with **GROUP-BY**

- **GROUP-BY** produces one tuple for each group
- **PARTITION** augments each tuple with group-based value(s)
- can use other functions than aggregates (e.g. ranking)
- can use attributes other than the partitioning ones

## ❖ Partitions (cont)

Syntax for **PARTITION**:

```
SELECT attr1, attr2, ...,
       aggregate1 OVER (PARTITION BY attri),
       aggregate2 OVER (PARTITION BY attrj), ...
  FROM Table
 WHERE condition on attributes
```

Note: the *condition* cannot include the *aggregate* value(s)

## ❖ Partitions (cont)

**Example:** show each city with daily temperature and temperature range

Schema: *Weather(city,date,temperature)*

```
SELECT city, date, temperature  
      min(temperature) OVER (PARTITION BY city) as lowest,  
      max(temperature) OVER (PARTITION BY city) as highest  
FROM   Weather;
```

Output: *Result(city, date, temperature, lowest, highest)*

## ❖ Partitions (cont)

Example showing **GROUP BY** and **PARTITION** difference:

```
SELECT city, min(temperature) max(temperature)
FROM Weather GROUP BY city
```

Result: one tuple for each city *Result(city,min,max)*

```
SELECT city, date, temperature as temp,
       min(temperature) OVER (PARTITION BY city),
       max(temperature) OVER (PARTITION BY city)
FROM Weather;
```

Result: one tuple for each temperature measurement.



## SQL Queries (v): Abstraction

---

- Complex Queries
- Using Views for Abstraction
- **FROM**-clause Subqueries for Abstraction
- **WITH**-clause Subqueries for Abstraction
- Recursive Queries

## ❖ Complex Queries

For complex queries, it is often useful to

- break the query into a collection of smaller queries
- define the top-level query in terms of these

This can be accomplished in several ways in SQL:

- **views** (discussed in detail below)
- subqueries in the **WHERE** clause
- subqueries in the **FROM** clause
- subqueries in a **WITH** clause

**VIEWS** and **WHERE** clause subqueries haveen discussed elsewhere.

**WHERE** clause subqueries can be **correlated** with the top-level query.

## ❖ Complex Queries (cont)

**Example:** get a list of low-scoring students in each course  
(low-scoring = mark is less than average mark for class)

Schema: *Enrolment(course,student,mark)*

Approach:

- generate tuples containing *(course,student,mark,classAvg)*
- select just those tuples satisfying *(mark < classAvg)*

Implementation of first step via window function

```
SELECT course, student, mark,  
       avg(mark) OVER (PARTITION BY course)  
FROM   Enrolments;
```

We now look at several ways to complete this data request ...

## ❖ Using Views for Abstraction

Defining complex queries using views:

```
CREATE VIEW
    CourseMarksWithAvg(course,student,mark,avg)
AS
SELECT course, student, mark,
       avg(mark) OVER (PARTITION BY course)
FROM   Enrolments;

SELECT course, student, mark
FROM   CourseMarksWithAvg
WHERE  mark < avg;
```

## ❖ Using Views for Abstraction (cont)

In the general case:

```
CREATE VIEW View1(a,b,c,d) AS Query1;
CREATE VIEW View2(e,f,g) AS Query2;
...
SELECT attributes
FROM View1, View2
WHERE conditions on attributes of View1 and View2
```

Notes:

- look like tables ("virtual" tables)
- exist as objects in the database (stored queries)
- useful if specific query is required frequently

## ❖ FROM-clause Subqueries for Abstraction

Defining complex queries using **FROM** subqueries:

```
SELECT course, student, mark
FROM   (SELECT course, student, mark,
               avg(mark) OVER (PARTITION BY course)
            FROM   Enrolments) AS CourseMarksWithAvg
WHERE  mark < avg;
```

Avoids the need to define views.

## ❖ FROM-clause Subqueries for Abstraction

(cont)

In the general case:

```
SELECT attributes
FROM   (Query1) AS Name1,
       (Query2) AS Name2
       ...
WHERE  conditions on attributes of Name1 and Name2
```

Notes:

- must provide name for each subquery, even if never used
- subquery table inherits attribute names from query  
(e.g. in the above, we assume that *Query*<sub>1</sub> returns an attribute called **a**)

## ❖ WITH-clause Subqueries for Abstraction

Defining complex queries using **WITH**:

```
WITH CourseMarksWithAvg AS
  (SELECT course, student, mark,
           avg(mark) OVER (PARTITION BY course)
      FROM Enrolments)
SELECT course, student, mark, avg
  FROM CourseMarksWithAvg
 WHERE mark < avg;
```

Avoids the need to define views.

## ❖ WITH-clause Subqueries for Abstraction

(cont)

In the general case:

```
WITH    Name1(a,b,c) AS (Query1),  
       Name2 AS (Query2), ...  
SELECT attributes  
FROM   Name1, Name2, ...  
WHERE  conditions on attributes of Name1 and Name2
```

Notes:

- $Name_1$ , etc. are like temporary tables
- named tables inherit attribute names from query

## ❖ Recursive Queries

**WITH** also provides the basis for recursive queries.

Recursive queries are structured as:

```
WITH RECURSIVE R(attributes) AS (
    SELECT ... not involving R
    UNION
    SELECT ... FROM R, ...
)
SELECT attributes
FROM   R, ...
WHERE  condition involving R's attributes
```

Useful for scenarios in which we need to traverse multi-level relationships.

## ❖ Recursive Queries (cont)

For a definition like

```
WITH RECURSIVE R AS ( Q1 UNION Q2 )
```

**Q<sub>1</sub>** does not include **R** (base case); **Q<sub>2</sub>** includes **R** (recursive case)

How recursion works:

```
Working = Result = evaluate Q1
while (Working table is not empty) {
    Temp = evaluate Q2, using Working in place of R
    Temp = Temp - Result
    Result = Result UNION Temp
    Working = Temp
}
```

i.e. generate new tuples until we see nothing not already seen.

## ❖ Recursive Queries (cont)

**Example:** count numbers of all sub-parts in a given part.

Schema: *Parts(part, sub\_part, quantity)*

```
WITH RECURSIVE IncludedParts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity
    FROM   Parts WHERE part = GivenPart
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity
    FROM   IncludedParts i, Parts p
    WHERE  p.part = i.sub_part
)
SELECT sub_part, SUM(quantity) as total_quantity
FROM   IncludedParts
GROUP  BY sub_part
```

Includes sub-parts, sub-sub-parts, sub-sub-sub-parts, etc.



# Stored Procedures

---

- Limitations of Basic SQL
- Extending SQL
- SQL as a Programming Language
- Database Programming
- Stored Procedures
- SQL Functions
- Functions vs Views

## ❖ Limitations of Basic SQL

What we have seen of SQL so far:

- data definition language (`create table(...)`)
- constraints (domain, key, referential integrity)
- query language (`select...from...where...`)
- views (give names to SQL queries)

This provides powerful declarative data extraction mechanisms.

This is not sufficient to write complete applications.

More **extensibility** and **programmability** are needed.

## ❖ Extending SQL

Ways in which standard SQL might be extended:

- new data types (incl. constraints, I/O, indexes, ...)
- object-orientation
- more powerful constraint checking
- packaging/parameterizing queries
- more functions/aggregates for use in queries
- event-based triggered actions

All are required to *assist* in application development.

But still do not provide a solution to developing applications.

## ❖ SQL as a Programming Language

At some point in developing complete database applications

- we need to implement user interactions
- we need to control sequences of database operations
- we need to process query results in complex ways
- we need to build a web interface for users to access data

and SQL cannot do any of these.

SQL cannot even do something as simple as factorial!

Ok ... so PostgreSQL added a factorial operator ... but it's non-standard.

## ❖ SQL as a Programming Language (cont)

Consider the problem of withdrawal from a bank account:

*If a bank customer attempts to withdraw more funds than they have in their account, then indicate "Insufficient Funds", otherwise update the account*

An attempt to implement this in SQL:

```
select 'Insufficient Funds'  
from Accounts  
where acctNo = AcctNum and balance < Amount;  
update Accounts  
set balance = balance - Amount  
where acctNo = AcctNum and balance >= Amount;  
select 'New balance: '||balance  
from Accounts  
where acctNo = AcctNum;
```

## ❖ SQL as a Programming Language (cont)

Two possible evaluation scenarios:

- displays "Insufficient Funds", **UPDATE** has no effect, displays unchanged balance
- **UPDATE** occurs as required, displays changed balance

Some problems:

- SQL doesn't allow parameterisation (e.g. *AcctNum*)
- always attempts **UPDATE**, even when it knows it's invalid
- need to evaluate (**balance** < *Amount*) test twice
- always displays balance, even when not changed

To accurately express the "business logic", we need facilities like conditional execution and parameter passing.

## ❖ Database Programming

Database programming requires a combination of

- manipulation of data in DB (via SQL)
- conventional programming (via procedural code)

This combination is realised in a number of ways:

- passing SQL commands via a "call-level" interface  
(prog lang is decoupled from DBMS; most flexible; e.g. Java/JDBC, PHP, Python)
- embedding SQL into augmented programming languages  
(requires pre-processor for language; typically DBMS-specific; e.g. SQL/C)
- special-purpose programming languages in the DBMS  
(closely integrated with DBMS; enable extensibility; e.g. PL/SQL, PLpgSQL)

Here we focus on the last: extending DBMS capabilities via programs stored in the DB

## ❖ Database Programming (cont)

Combining **SQL** and **procedural** code solves the "withdrawal" problem:

```
create function
    withdraw(acctNum text, amount integer) returns text
declare bal integer;
begin
    set bal = (select balance
                from   Accounts
                where  acctNo = acctNum);
    if (bal < amount) then
        return 'Insufficient Funds';
    else
        update Accounts
        set    balance = balance - amount
        where  acctNo = acctNum;
        set bal = (select balance
                    from   Accounts
                    where  acctNo = acctNum);
        return 'New Balance: ' || bal;
    end if
end;
```

(This example is actually a stored procedure, using SQL/PSM syntax)

## ❖ Stored Procedures

Stored procedures are small programs ...

- stored in the database, alongside the stored data
- invoked in SQL queries, or automatically invoked in triggers

SQL/PSM is a standard for stored procedures, developed in 1996.

By then, most DBMSs had their own stored procedure languages.

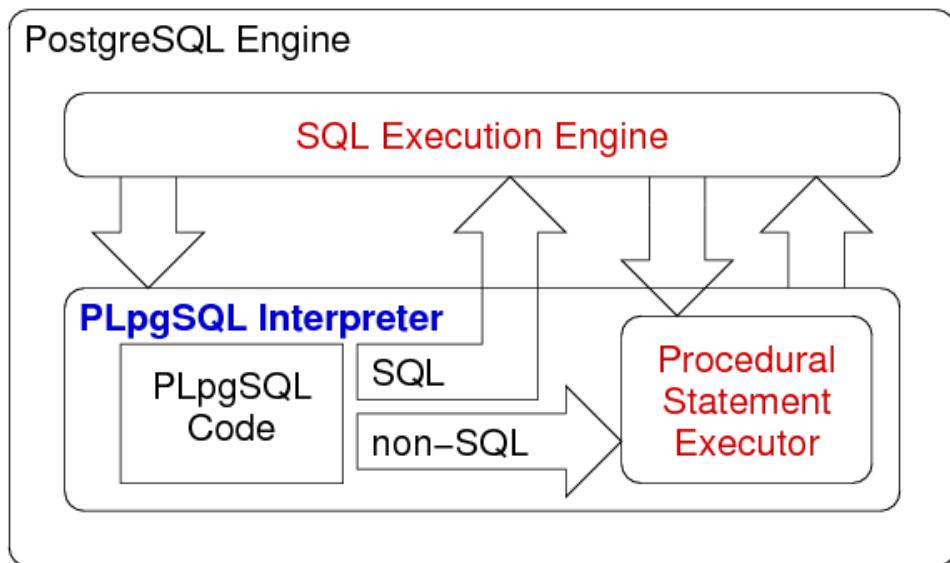
PostgreSQL supports stored procedures in a variety of languages

- PLpgsql ... PostgreSQL-specific procedural language (cf. Oracle's PL/SQL)
- SQL ... functions that resemble parameterised views
- Python, Perl, Tcl, ... etc.

## ❖ Stored Procedures (cont)

The PLpgSQL interpreter

- executes procedural code and manages variables
- calls PostgreSQL engine to evaluate SQL statements



Embedded in DBMS engine, so efficient to execute with queries

## ❖ SQL Functions

PostgreSQL allows functions to be defined in SQL

```
CREATE OR REPLACE
    funcName(arg1type, arg2type, ....)
    RETURNS rettype
AS $$
    SQL statements
$$ LANGUAGE sql;
```

Within the function, arguments are accessed as **\$1, \$2, ...**

Return value: result of the last SQL statement.

*rettype* can be any PostgreSQL data type (incl tuples,tables).

Function returning a table: **returns setof TupleType**

Details: PostgreSQL Documentation, Section 37.5

## ❖ SQL Functions (cont)

**Example:** info about bars from a given suburb

```
create or replace function
    hotelsIn(text) returns setof Bars
as $$
select * from Bars where addr = $1;
$$ language sql;

-- usage examples
select * from hotelsIn('The Rocks');
      name      |     addr      |   license
-----+-----+-----
Australia Hotel | The Rocks | 123456
Lord Nelson    | The Rocks | 123888

select * from hotelsIn('Randwick');
      name      |     addr      |   license
-----+-----+-----
Royal Hotel    | Randwick | 938500
```

## ❖ SQL Functions (cont)

**Example:** Name of cheapest beer at each bar

```
create view Cheapest(bar, price) as
select bar, min(price) from Sells group by bar;

select s.*
from   Sells s
where  s.price =
       (select price from Cheapest where bar = s.bar);
```

Could be implemented by defining an SQL function

**LowestPriceAt(bar)**

```
create or replace
    function LowestPriceAt(text) returns float
as $$
select min(price) from Sells where bar = $1;
$$ language sql;

select * from Sells where price = LowestPriceAt(bar);
```

## ❖ Functions vs Views

A parameterless function behaves similar to a view

E.g.

```
create or replace view EmpList
as
select given||' '||family as name,
       street||', '||town as addr
from   Employees;
```

which is used as

```
mydb=# select * from EmpList;
```

## ❖ Functions vs Views (cont)

Compared to its implementation as a function:

```
create type EmpRecord as (name text, addr text);

create or replace function
    EmpList() returns setof EmpRecord
as $$
select family||' '||given as name,
       street||', '||town as addr
from   Employees
$$ language sql;
```

which is used as

```
mydb=# select * from EmpList();
```



# PLpgsql (i)

- PLpgsql
- Defining PLpgsql Functions
- PLpgsql Examples
- PLpgsql Gotchas
- Data Types
- Syntax/Control Structures
- SELECT...INTO

## ❖ PLpgsql

**PLpgsql** = Procedural Language extensions to PostgreSQL

A PostgreSQL-specific language integrating features of:

- procedural programming and SQL programming

Provides a means for extending DBMS functionality, e.g.

- implementing constraint checking (triggered functions)
- complex query evaluation (e.g. recursive)
- complex computation of column values
- detailed control of displayed results

Details: PostgreSQL Documentation, Chapter 42

## ❖ Defining PLpgsql Functions

PLpgsql functions are created (and inserted into db) via:

```
CREATE OR REPLACE
    funcName(param1, param2, ....)
    RETURNS rettype
AS $$  
DECLARE
    variable declarations
BEGIN
    code for function
END;
$$ LANGUAGE plpgsql;
```

Note: the entire function body is a single SQL string  
(\$\$...\$\$)

## ❖ PLpgSQL Examples

**Example:** function to compute  $x/y$  "safely"

```
create or replace function
    div(x integer, y integer) returns integer
as $$

declare
    result integer;          -- variable
begin
    if (y <> 0) then      -- conditional
        result := x/y;      -- assignment
    else
        result := 0;         -- assignment
    end if;
    return result;
end;
$$ language plpgsql;
```

## ❖ PLpgSQL Examples (cont)

**Example:** function to compute n!

```
create or replace function
    factorial(n integer) returns integer
as $$

declare
    i integer;
    fac integer := 1;
begin
    for i in 1..n loop
        fac := fac * i;
    end loop;
    return fac;
end;
$$ language plpgsql;
```

## ❖ PLpgsql Examples (cont)

**Example:** function to compute n! recursively

```
create function
    factorial(n integer) returns integer
as $$ 
begin
    if n < 2 then
        return 1;
    else
        return n * factorial(n-1);
    end if;
end;
$$ language plpgsql;
```

Usage: **select factorial(5);**

## ❖ PLpgSQL Examples (cont)

**Example:** handle withdraw from account and return status message

```

create function
    withdraw(acctNum text, amount integer) returns text
as $$ 
declare bal integer;
begin
    select balance into bal
    from   Accounts
    where  acctNo = acctNum;
    if bal < amount then
        return 'Insufficient Funds';
    else
        update Accounts
        set    balance = balance - amount
        where  acctNo = acctNum;
        select balance into bal
        from   Accounts
        where  acctNo = acctNum;
        return 'New Balance: ' || bal;
    end if;
end;
$$ language plpgsql;

```

Type cast

16 :: Double

## ❖ PLpgsql Gotchas

Some things to beware of:

- doesn't provide any i/o facilities (except **RAISE NOTICE**)
  - the aim is to build computations on tables that SQL alone can't do
- functions are not syntax-checked when loaded into DB
  - you don't find out about the syntax error until "run-time"
- error messages are sometimes not particularly helpful
- functions are defined as strings
  - change of "lexical scope" can sometimes be confusing
- giving params/variables the same names as attributes
  - can avoid by starting all param/var names with underscore

Summary: debugging PLpgsql can sometimes be tricky.

## ❖ Data Types

PLpgSQL constants and variables can be defined using:

- standard SQL data types (**CHAR**, **DATE**, **NUMBER**, ...)
- user-defined PostgreSQL data types (e.g. **Point**)
- a special structured record type (**RECORD**)
- table-row types (e.g. **Branches%ROWTYPE** or simply **Branches**)
- types of existing variables (e.g. **Branches.location%TYPE**)

There is also a **CURSOR** type for interacting with SQL.

## ❖ Data Types (cont)

Variables can also be defined in terms of:

- the type of an existing variable or table column
- the type of an existing table row (implicit **RECORD** type)

### Examples:

```
quantity      INTEGER;
start_qty     quantity%TYPE;

employee      Employees%ROWTYPE;
-- or
employee      Employees;

name          Employees.name%TYPE;
```

## ❖ Syntax/Control Structures

Typical set of control structures, with extensions:

Assignment    *var := expr*

**SELECT expr INTO var**

Selection

```
IF Cond1 THEN S1
ELSIF Cond2 THEN S2 ...
ELSE S END IF
```

Iteration

```
LOOP S END LOOP
WHILE Cond LOOP S END LOOP
FOR rec_var IN Query LOOP ...
FOR int_var IN lo..hi LOOP ...
```

$S_i$  = list of PLpgsql statements, each terminated by semi-colon

## ❖ SELECT...INTO

Can capture query results via:

```
SELECT Exp1, Exp2, ..., Expn
INTO   Var1, Var2, ..., Varn
FROM    TableList
WHERE   Condition ...
```

The semantics:

- execute the query as usual
- return "projection list" ( $Exp_1, Exp_2, \dots$ ) as usual
- assign each  $Exp_i$  to corresponding  $Var_i$

## ❖ SELECT...INTO (cont)

Assigning a simple value via **SELECT . . . INTO**:

```
-- cost is local var, price is attr
select price into cost
from StockList
where item = 'Cricket Bat';
cost := cost * (1+tax_rate);
total := total + cost;
```

The current PostgreSQL parser also allows this syntax:

```
select into cost price
from StockList
where item = 'Cricket Bat';
```

## ❖ SELECT...INTO (cont)

Assigning whole rows via **SELECT...INTO**:

```
declare
    emp      Employees%ROWTYPE;
    -- alternatively, emp RECORD;
    eName    text;
    pay      real;
begin
    select * into emp
    from Employees where id = 966543;
    eName := emp.name;
    ...
    select name,salary into eName,pay
    from Employees where id = 966543;
end;
```

## ❖ SELECT...INTO (cont)

In the case of a PLpgSQL statement like

```
select a into b from R where ...
```

If the selection returns no tuples

- the variable **b** gets the value **NULL**

If the selection returns multiple tuples

- the variable **b** gets the value from the first tuple

## ❖ SELECT...INTO (cont)

An alternative to check for "no data found"

Use the special variable **FOUND** ...

- local to each function, set false at start of function
- set true if a **SELECT** finds at least one tuple
- set true if **INSERT/DELETE/UPDATE** affects at least one tuple
- otherwise, remains as **FALSE**

Example of use:

```
select a into b from R where ...
if (not found) then
    -- handle case where no matching tuples b
```

---

Produced: 6 Oct 2020

## PLpgsql (ii)

- PLpgsql Functions (recap)
- Debugging Output
- Returning Multiple Values
- INSERT ... RETURNING
- Exceptions

## ❖ PLpgSQL Functions (recap)

Defining PLpgSQL functions:

```
CREATE OR REPLACE
  funcName(param1, param2, ....)
  RETURNS rettype
AS $$
```

DECLARE  
  *variable declarations*

```
BEGIN
  code for function
END;
$$ LANGUAGE plpgsql;
```

Setting *rettype* to **void** means "no return value"

## ❖ Debugging Output

Printing info about intermediate states is critical for debugging

Depending on how PostgreSQL is configured

- **raise notice** allows you to display info from a function
- displayed in **psql** window during the function's execution
- usage: **raise notice**  
**'FormatString' , Value<sub>1</sub>, ... Value<sub>n</sub>;**

Example:

```
-- assuming x==3, y==3.14, z='abc'  
raise notice 'x+1 = %, y = %, z = %', x+1, y, z;  
-- displays "NOTICE: x+1 = 4, y = 3.14, z = abc"
```

## ❖ Debugging Output (cont)

**Example:** a simple function with debugging output

### Function

```
create or replace function
    seq(_n int) returns setof int
as $$$
declare i int;
begin
    for i in 1.._n loop
        raise notice 'i=%',i;
        return next i;
    end loop;
end;
$$ language plpgsql;
```

### Output

```
db=# select * from seq(3);
NOTICE:  i=1
NOTICE:  i=2
NOTICE:  i=3
          seq
-----
           1
           2
           3
(3 rows)
```

Replacing **notice** by **exception** causes function to terminate in first iteration

## ❖ Returning Multiple Values

PLpgSQL functions can return a set of values (**setof** *Type*)

- effectively a function returning a table
- *Type* could be atomic  $\Rightarrow$  like a single column
- *Type* could be tuples  $\Rightarrow$  like a full table

Atomic types, e.g.

`integer, float, numeric, date, text, varchar(n), ...`

Tuple types, e.g.

```
create type Point as (x float, y float);
```

## ❖ Returning Multiple Values (cont)

Example function returning a set of tuples

```
create type MyPoint as (x integer, y integer);

create or replace function
    points(n integer, m integer) returns setof MyPoint
as $$ 
declare
    i integer;    j integer;
    p MyPoint;   -- tuple variable
begin
    for i in 1 .. n loop
        for j in 1 .. m loop
            p.x := i;  p.y := j;
            return next p;
        end loop;
    end loop;
end;
$$ language plpgsql;
```

## ❖ Returning Multiple Values (cont)

Functions returning **setof Type** are used like tables

```
db=# select * from points(2,3);
 x | y
---+---
 1 | 1
 1 | 2
 1 | 3
 2 | 1
 2 | 2
 2 | 3
(6 rows)
```

## ❖ INSERT ... RETURNING

Can capture values from tuples inserted into DB:

```
insert into Table(...) values  
(Val1, Val2, ... Valn)  
returning ProjectionList into VarList
```

Useful for recording id values generated for **serial** PKs:

```
declare newid integer; colour text;  
...  
insert into T(id,a,b,c) values (default,2,3,'red')  
returning id,c into newid,colour;  
-- id contains the primary key value  
-- for the new tuple T(?,2,3,'red')
```

## ❖ Exceptions

PLpgSQL supports exception handling via

```
begin
    Statements...
exception
    when Exceptions1 then
        StatementsForHandler1
    when Exceptions2 then
        StatementsForHandler2
    ...
end;
```

Each *Exceptions*<sub>i</sub> is an **OR** list of exception names, e.g.

division\_by\_zero OR floating\_point\_exception OR ...

A list of exceptions is in Appendix A of the PostgreSQL Manual.

## ❖ Exceptions (cont)

When an exception occurs:

- control is transferred to the relevant exception handling code
- all database changes so far in this transaction are undone
- all function variables retain their current values
- handler executes and then transaction aborts (and function exits)

*Illustrated  
in  
example  
below.*

If no handler in current scope, exception passed to next outer level.

Default exception handlers, at outermost level, exit and log error.

## ❖ Exceptions (cont)

**Example:** exception handling:

```
-- table T contains one tuple ('Tom','Jones')
declare
    x integer := 3;
    y integer;
begin
    update T set firstname = 'Joe'
    where lastname = 'Jones';
    -- table T now contains ('Joe','Jones')
    x := x + 1;
    y := x / 0;
exception
    when division_by_zero then
        -- update on T is rolled back to ('Tom','Jones')
        raise notice 'caught division_by_zero';
        return x; -- value returned is 4
end;
```

## ❖ Exceptions (cont)

The **raise** operator can generate server log entries, e.g.

```
raise debug1 'Simple message';
raise notice 'User = %',user_id;
raise exception 'Fatal: value was %',value; Kill the process
```

There are several levels of severity:

*low → high.*

- **DEBUG1, LOG, INFO, NOTICE, WARNING, and EXCEPTION**
- not all severities generate a message to the client (**psql**)

Your CSE server log is the file **/srvr/you/pgsql/Log**

Server logs can grow *very* large; delete when you shut your server down



## PLpgsql (iii)

- PLpgsql Functions (recap)
- Query results in PLpgsql
- Dynamically Generated Queries
- Functions vs Views

## ❖ PLpgSQL Functions (recap)

Defining PLpgSQL functions:

```
CREATE OR REPLACE
  funcName(param1, param2, ....)
  RETURNS rettype
AS $$
```

DECLARE  
  *variable declarations*

```
BEGIN
  code for function
END;
$$ LANGUAGE plpgsql;
```

## ❖ Query results in PLpgSQL

Can evaluate a query and iterate through its results

- one tuple at a time, using a **for ... loop**

```
declare
    tup Type;
begin
    for tup in Query
        loop
            Statements;
        end loop;
    end;
```

*Must be some type.*

- Type of **tup** variable must match type of *Query* results
- If declared as **record**, will automatically match *Query* results type

## ❖ Query results in PLpgSQL (cont)

**Example:** count the number of Employees earning more than min.salary

```
create or replace function
    well_paid(_minsal integer) returns integer
as $$
declare
    nemps integer := 0;
    tuple record; -- could also be tuple Employees;
begin
    for tuple in
        select * from Employees where salary > _minsal
    loop
        nemps := nemps + 1;
    end loop;
    return nemps;
end;
$$ language plpgsql;
```

## ❖ Query results in PLpgSQL (cont)

Alternative to the above (but less efficient):

```
create or replace function
    well_paid(_minsal integer) returns integer
as $$
declare
    nemps integer := 0;
    tuple record;
begin
    for tuple in
        select name,salary from Employees
    loop
        if (tuple.salary > _minsal) then
            nemps := nemps + 1;
        end if;
    end loop;
    return nemps;
end;
$$ language plpgsql;
```

Try to do it in clever.

unnecessary extra operation in loop.

## ❖ Query results in PLpgSQL (cont)

And the example could be done more simply (and efficiently) as:

```
create or replace function
    well_paid(_minsal integer) returns integer
as $$ 
declare
    nemps integer;
begin
    select count(*) into nemps
    from Employees where salary > _minsal
    return nemps;
end;
$$ language plpgsql;
```

## ❖ Dynamically Generated Queries

**EXECUTE** takes a string and executes it as an SQL query.

Examples:

```
-- constant string
execute 'select * from Employees';
-- concatenation of constant strings
execute 'select * from ' || 'Employees';
-- using a name of e.g. table or attribute
execute 'select * from ' || quote_ident($1);
-- using a value generated in the program
execute 'delete from Accounts' ||
    ' where holder=' || quote_literal($1);
```

Can be used in any context where an SQL query is expected

This mechanism allows us to **construct** queries "on the fly".

## ❖ Dynamically Generated Queries (cont)

Example: a wrapper for updating a single text field

```
function for updating.
create or replace function
    set(_table text, _attr text, _val text) returns void
as $$
declare
    query text;
begin
    query := 'update ' || quote_ident(_table);
    query := query || ' SET ' || quote_ident(_attr);
    query := query || ' = ' || quote_literal(_val);
    execute query;
end; $$ language plpgsql;   = Update branches set assets = 0.00
```

which could be used as e.g.

```
select set('branches','assets','0.00');
```

## ❖ Dynamically Generated Queries (cont)

One limitation of **EXECUTE**:

- cannot use **select into** inside dynamic queries

Needs to be expressed instead as:

```
declare tuple R%rowtype; n int;
execute 'select * from R where id='||n into tuple;
-- or
declare x int; y int; z text;
execute 'select a,b,c from R where id='||n into x,y,z;
```

Notes:

- if query returns multiple tuples, first one is stored
- if query returns zero tuples, all nulls are stored

## ❖ Functions vs Views

A difference between views and functions returning a **SETOF**:

- **CREATE VIEW** produces a "virtual" table definition
- **SETOF** functions require an existing tuple type

In examples above, we used existing **Employees** tuple type.

In general, you need to define the tuple return type via

```
create type TupleType as ( attr1 type1, ... attrn typen );
```

Other major differences between **setof** functions and views ...

- functions have parameters; views don't (although **where** might help)
- functions are "run-time" objects; views are interpolated into queries

## ❖ Functions vs Views (cont)

Another example of function returning **setof** tuples ...

```
create type EmpInfo as (name text, pay integer);

create or replace function
    richEmps(_minsal integer) returns setof EmpInfo
as $$
declare
    emp record;    info EmpInfo;
begin
    for emp in
        select * from Employees where salary > _minsal
    loop
        info.name := emp.name;
        info.pay := emp.salary;
        return next info;
    end loop;
end; $$ language plpgsql;
```

## ❖ Functions vs Views (cont)

Using the function ...

```
select * from richEmps(100000);
```

has input.  
flexible to change the input.

versus a view

```
create or replace view richEmps(name,pay) as  
select name, salary from Employees where salary > 100000;
```

```
select * from richEmps;
```

-- but no scope for different salary  
no input.

versus an SQL function

```
create or replace function  
richEmps(_minsal integer) returns setof EmpInfo  
as $$  
select name, salary from Employees where salary > _minsal;  
$$ language sql;
```



# Aggregates

---

- Aggregates
- User-defined Aggregates

## ❖ Aggregates

Aggregates reduce a collection of values into a single result.

Examples: **count**(*Tuples*), **sum**(*Numbers*),  
**max**(*AnyOrderedType*)

The action of an aggregate function can be viewed as:

```
State = initial state
for each item T {
    # update State to include T
    State = updateState(State, T)
}
return makeFinal(State)
```

## ❖ Aggregates (cont)

Aggregates are commonly used with **GROUP BY**.

In that context, they "summarise" each group.

Example:

R		
a	b	c
1	2	x
1	3	y
2	2	z
2	1	a
2	3	b

```
select a,sum(b),count(*)  
from R group by a
```

a	sum	count
1	5	2
2	6	3

## ❖ User-defined Aggregates

SQL standard does not specify user-defined aggregates.

But PostgreSQL provides a mechanism for defining them.

To define a new aggregate, first need to supply:

- *BaseType* ... type of input values
- *StateType* ... type of intermediate states
- state mapping function:  $sfunc(state,value) \rightarrow newState$
- [optionally] an initial state value (defaults to null)
- [optionally] final function:  $ffunc(state) \rightarrow result$

## ❖ User-defined Aggregates (cont)

New aggregates defined using **CREATE AGGREGATE** statement:

```
CREATE AGGREGATE AggName(BaseType) (
    sfunc      = UpdateStateFunction,
必须.    { stype       = StateType,
    initcond   = InitialValue,
    finalfunc  = MakeFinalFunction,
    sortop     = OrderingOperator
);
```

- **initcond** (type *StateType*) is optional; defaults to **NULL**
- **finalfunc** is optional; defaults to identity function
- **sortop** is optional; needed for min/max-type aggregates

## ❖ User-defined Aggregates (cont)

Example: defining the **count** aggregate (roughly)

```
create aggregate myCount(anyelement) (
    stype      = int,      -- the accumulator type
    initcond   = 0,        -- initial accumulator value
    sfunc      = oneMore -- increment function
);

create function
    oneMore(sum int, x anyelement) returns int
as $$
begin return sum + 1; end;
$$ language plpgsql;
```

## ❖ User-defined Aggregates (cont)

Example: **sum2** sums two columns of integers

```
create type IntPair as (x int, y int);

create function
    addPair(sum int, p IntPair) returns int
as $$$
begin return sum + p.x + p.y; end;
$$ language plpgsql;

create aggregate sum2(IntPair) (
    stype      = int,
    initcond   = 0,
    sfunc      = addPair
);
```

## ❖ User-defined Aggregates (cont)

PostgreSQL has many aggregates (e.g. **sum**, **count**, ...)

But it doesn't have a product aggregate.

# Implement a **prod** aggregate that

- computes the product of values in a column of numeric data

## Usage:

```
select prod(*) from iota(5);  
prod  
-----  
120
```

## ❖ User-defined Aggregates (cont)

**Example:** product aggregate

```
create function
    mult(soFar numeric, next numeric) returns numeric
as $$$
begin return soFar * next; end;
$$ language plpgsql;

create aggregate prod(numeric) (
    stype      = numeric,
    initcond   = 1,
    sfunc      = mult
);
```

## ❖ User-defined Aggregates (cont)

Define a **concat** aggregate that

- takes a column of string values
- returns a comma-separated string of values

Example:

```
select count(*), concat(name) from Employee;  
-- returns e.g.  
count | concat  
-----+-----  
 4   | John,Jane,David,Phil
```

## ❖ User-defined Aggregates (cont)

**Example:** string concatenation aggregate

```
create function
    join(s1 text, s2 text) returns text
as $$

begin
    if (s1 = '') then      +   in python,
        return s2;
    else
        return s1||','||s2;
    end if;
end;
$$ language plpgsql;

create aggregate concat(text) (
    stype    = text,
    initcond = '',
    sfunc    = join
);
```



# Constraints

---

- Constraints
- Assertions

## ❖ Constraints

So far, we have considered several kinds of constraints:

- attribute (column) constraints
- relation (table) constraints
- referential integrity constraints

Examples:

```
create table Employee (
    id      integer primary key,
    name    varchar(40),
    salary  real,
    age     integer check (age > 15),
    worksIn integer
        references Department(id),
    constraint PayOk check (salary > age*1000)
);
```

## ❖ Constraints (cont)

Column and table constraints ensure validity of one table.

Ref. integrity constraints ensure connections between tables are valid.

However, specifying validity of entire database often requires constraints involving multiple tables.

Simple example (from banking domain):

```
for all Branches b  
    b.assets == (select sum(acct.balance) We need to check something like this.  
                  from Accounts acct  
                  where acct.branch = b.location)
```

i.e. assets of a branch is sum of balances of accounts held at that branch

## ❖ Assertions

Assertions are schema-level constraints

- typically involving multiple tables
- expressing a condition that must hold at all times
- need to be checked on each change to relevant tables
- if change would cause check to fail, reject change

SQL syntax for assertions:

`CREATE ASSERTION name CHECK (condition)`

*Ture will not trigger.*

The *condition* is expressed as "there are no violations in the database"

Implementation: ask a query to find all the violations; check for empty result

## ❖ Assertions (cont)

**Example:** #students in any UNSW course must be < 10000

```
create assertion ClassSizeConstraint check (
    not exists (
        select c.id
        from Courses c
            join Enrolments e on (c.id = e.course)
        group by c.id
        having count(e.student) > 9999
    )
);
```

Needs to be checked after *every* change to either **Courses** or **Enrolments**

## ❖ Assertions (cont)

**Example:** assets of branch = sum of its account balances

```
create assertion AssetsCheck check (
    not exists (
        select branchName from Branches b
        where b.assets <>
            (select sum(a.balance) from Accounts a
             where a.branch = b.location)
    )
);
```

Needs to be checked after *every* change to either **Branches** or **Accounts**

## ❖ Assertions (cont)

On each update, it is expensive *heavy cost.*

- to determine which assertions need to be checked
- to run the queries which check the assertions

A database with many assertions would be way too slow.

So, most RDBMSs do *not* implement general assertions.

Typically, **triggers** are provided as

- a lightweight mechanism for dealing with assertions
- a general event-based programming tool for databases

Triggers typically enforce assertions rather than *checking* them

**SQL Standard**



# Triggers

---

- Triggers
- Trigger Semantics
- Triggers in PostgreSQL
- Trigger Example #1
- Trigger Example #2

## ❖ Triggers

Triggers are

- procedures stored in the database
- activated in response to database events (e.g. updates)

Examples of uses for triggers:

- maintaining summary data
- checking schema-level constraints (assertions) on update
- performing multi-table updates (to maintain assertions)

## ❖ Triggers (cont)

Triggers provide event-condition-action (ECA) programming:

- an **event** activates the trigger
- on activation, the trigger checks a **condition**
- if the condition holds, a procedure is executed (the **action**)

Some typical variations within this:

- execute the action **before**, **after** or **instead of** the triggering event
- can refer to both **old** and **new** values of updated tuples
- can limit updates to a particular set of attributes
- perform action: **for each** modified tuple, **once for all** modified tuples      *check once all finished.*

## ❖ Triggers (cont)

SQL "standard" syntax for defining triggers:

```
CREATE TRIGGER TriggerName
{AFTER|BEFORE} Event1 [ OR Event2 ... ]
[ FOR EACH ROW ]
ON TableName
[ WHEN ( Condition ) ]
Block of Procedural/SQL Code ;
```

Possible *Events* are **INSERT**, **DELETE**, **UPDATE**.

**FOR EACH ROW** clause ...

- if present, code is executed on each modified tuple
- if not present, code is executed once after all tuples are modified, just before changes are finally **COMMITTED**

## ❖ Trigger Semantics

Triggers can be activated **BEFORE** or **AFTER** the event.

If activated **BEFORE**, can affect the change that occurs:

- **NEW** contains "proposed" value of changed tuple
- modifying **NEW** causes a different value to be placed in DB

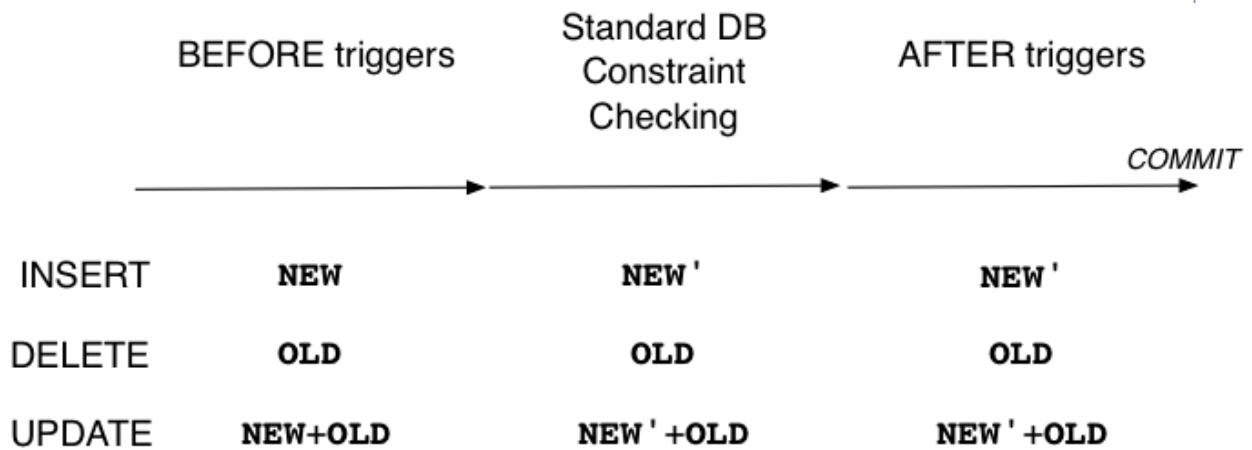
If activated **AFTER**, the effects of the event are visible:

- **NEW** contains the current value of the changed tuple
- **OLD** contains the previous value of the changed tuple
- constraint-checking has been done for **NEW**

Note: **OLD** does not exist for insertion; **NEW** does not exist for deletion.

## ❖ Trigger Semantics (cont)

Sequence of activities during database update:



Reminder: **BEFORE** trigger can modify value of new tuple

## ❖ Trigger Semantics (cont)

Consider two triggers and an INSERT statement

```
create trigger X before insert on T Code1;  
create trigger Y after insert on T Code2;  
insert into T values (a,b,c,...);
```

Sequence of events:

- execute **Code1** for trigger **X**
- code has access to **(a, b, c, ...)** via **NEW**
- code typically checks the values of **a, b, c, ..**
- code can modify values of **a, b, c, ..** in **NEW**
- DBMS does constraint checking as if **NEW** is inserted
- if fails any checking, abort insertion and rollback
- execute **Code2** for trigger **Y**
- code has access to final version of tuple via **NEW**
- code typically does final checking, or modifies other tables in database to ensure assertions are satisfied

Reminder: there is no **OLD** tuple for an **INSERT** trigger.

## ❖ Trigger Semantics (cont)

Consider two triggers and an UPDATE statement

```
create trigger X before update on T Code1;  
create trigger Y after update on T Code2;  
update T set b=j,c=k where a=m;
```

Sequence of events:

- execute **Code1** for trigger **X**
- code has access to current version of tuple via **OLD**
- code has access to updated version of tuple via **NEW**
- code typically checks new values of **b, c, . . .**
- code can modify values of **a, b, c, . . .** in **NEW**
- do constraint checking as if **NEW** has replaced **OLD**
- if fails any checking, abort update and rollback
- execute **Code2** for trigger **Y**
- code has access to final version of tuple via **NEW**
- code typically does final checking, or modifies other tables in database to ensure constraints are satisfied

Reminder: both **OLD** and **NEW** exist in UPDATE triggers.

## ❖ Trigger Semantics (cont)

Consider two triggers and an DELETE statement

```
create trigger X before delete on T Code1;  
create trigger Y after delete on T Code2;  
delete from T where a=m;
```

Sequence of events:

- execute **Code1** for trigger **X**
- code has access to **(a, b, c, ...)** via **OLD**
- code typically checks the values of **a, b, c, ..**
- DBMS does constraint checking as if **OLD** is removed
- if fails any checking, abort deletion (restore **OLD**)
- execute **Code2** for trigger **Y**
- code has access to about-to-be-deleted tuple via **OLD**
- code typically does final checking, or modifies other tables in database to ensure constraints are satisfied

Reminder: tuple **NEW** does not exist in DELETE triggers.

## ❖ Triggers in PostgreSQL

PostgreSQL triggers provide a mechanism for

- **INSERT, DELETE or UPDATE** events
- to automatically activate PLpgSQL functions

Syntax for PostgreSQL trigger definition:

```
CREATE TRIGGER TriggerName
{AFTER | BEFORE} Event1 [OR Event2 ...]
ON TableName
[ WHEN ( Condition ) ]
FOR EACH {ROW | STATEMENT}
EXECUTE PROCEDURE FunctionName(args...);
```

## ❖ Triggers in PostgreSQL (cont)

There is no restriction on what code can go in the function.

However a **BEFORE** function must contain one of:

RETURN old;      or      RETURN new;

depending on which version of the tuple is to be used.

If **BEFORE** trigger returns **OLD**, no change occurs.

If exception is raised in trigger function, no change occurs.

## ❖ Trigger Example #1

Consider a database of people in the USA:

```
create table Person (
    id      integer primary key,
    ssn     varchar(11) unique,
    ... e.g. family, given, street, town ...
    state   char(2), ...
);
create table States (
    id      integer primary key,
    code   char(2) unique,
    ... e.g. name, area, population, flag ...
);
```

Constraint: **Person.state** ∈ (select code from States),  
or  
**exists (select id from States where**  
**code=Person.state)**

## ❖ Trigger Example #1 (cont)

**Example:** ensure that only valid state codes are used:

```

create trigger checkState before insert or update
on Person for each row execute procedure checkState();

create function checkState() returns trigger as $$$
begin
    -- normalise the user-supplied value
    new.state = upper(trim(new.state));
    if (new.state !~ '^[A-Z][A-Z]$') then
        raise exception 'Code must be two alpha chars';
    end if;
    -- implement referential integrity check
    select * from States where code=new.state;
    if (not found) then
        raise exception 'Invalid code %',new.state;
    end if;
    return new;
end;
$$ language plpgsql;

```

## ❖ Trigger Example #1 (cont)

Examples of how this trigger would behave:

```
insert into Person
    values('John',..., 'Calif.',...);
-- fails with 'Statecode must be two alpha chars'

insert into Person
    values('Jane',..., 'NY',...);
-- insert succeeds; Jane lives in New York

update Person
    set town='Sunnyvale', state='CA'
        where name='Dave';
-- update succeeds; Dave moves to California

update Person
    set state='OZ' where name='Pete';
-- fails with 'Invalid state code OZ'
```

## ❖ Trigger Example #2

**Example:** department salary totals

Scenario:

```
Employee(id, name, address, dept, salary, ...)  
Department(id, name, manager, totSal, ...)
```

An assertion that we wish to maintain:

```
create assertion TotalSalary check (  
    not exists (  
        select * from Department d  
        where d.totSal <> (select sum(e.salary)  
                            from Employee e  
                            where e.dept = d.id)  
    )  
)
```

## ❖ Trigger Example #2 (cont)

Events that might affect the validity of the database

- a new employee starts work in some department
- an employee gets a rise in salary
- an employee changes from one department to another
- an employee leaves the company

A single assertion could check for this after each change.

With triggers, we have to program each case separately.

Each program implements updates to *ensure* assertion holds.

## ❖ Trigger Example #2 (cont)

Implement the Employee update triggers from above in PostgreSQL:

Case 1: new employees arrive

```
create trigger TotalSalary1
after insert on Employees
for each row execute procedure totalSalary1();

create function totalSalary1() returns trigger
as $$
begin
    if (new.dept is not null) then
        update Department
        set totSal = totSal + new.salary
        where Department.id = new.dept;
    end if;
    return new;
end;
$$ language plpgsql;
```

## ❖ Trigger Example #2 (cont)

Case 2: employees change departments/salaries

```
create trigger TotalSalary2
after update on Employee
for each row execute procedure totalSalary2();

create function totalSalary2() returns trigger
as $$
begin
    update Department
        set totSal = totSal + new.salary
    where Department.id = new.dept;
    update Department
        set totSal = totSal - old.salary
    where Department.id = old.dept;
    return new;
end;
$$ language plpgsql;
```

## ❖ Trigger Example #2 (cont)

Case 3: employees leave

```
create trigger TotalSalary3
after delete on Employee
for each row execute procedure totalSalary3();

create function totalSalary3() returns trigger
as $$
begin
    if (old.dept is not null) then
        update Department
        set totSal = totSal - old.salary
        where Department.id = old.dept;
    end if;
    return old;
end;
$$ language plpgsql;
```



# Programming with Databases

---

- Programming with Databases
- PL/DB Interface
- PL/DB Mismatch

## ❖ Programming with Databases

So far, we have seen ...

- accessing data via SQL queries
- packaging SQL queries as views/functions
- building functions to return tables
- implementing assertions via triggers

All of the above programming

- is very close to the data
- takes place inside the DBMS

## ❖ Programming with Databases (cont)

While SQL (+ PLpgSQL) gives a powerful data access mechanism

- it is *not* an application programming language

Complete applications require code to

- handle the user interface (GUI or Web)
- interact with other systems (e.g. other DBs)
- perform compute-intensive work (vs. data-intensive)

"Conventional" programming languages (PLs) provide these.

We need PL + DBMS connectivity.

## ❖ Programming with Databases (cont)

Requirements of an interface between PL and RDBMS:

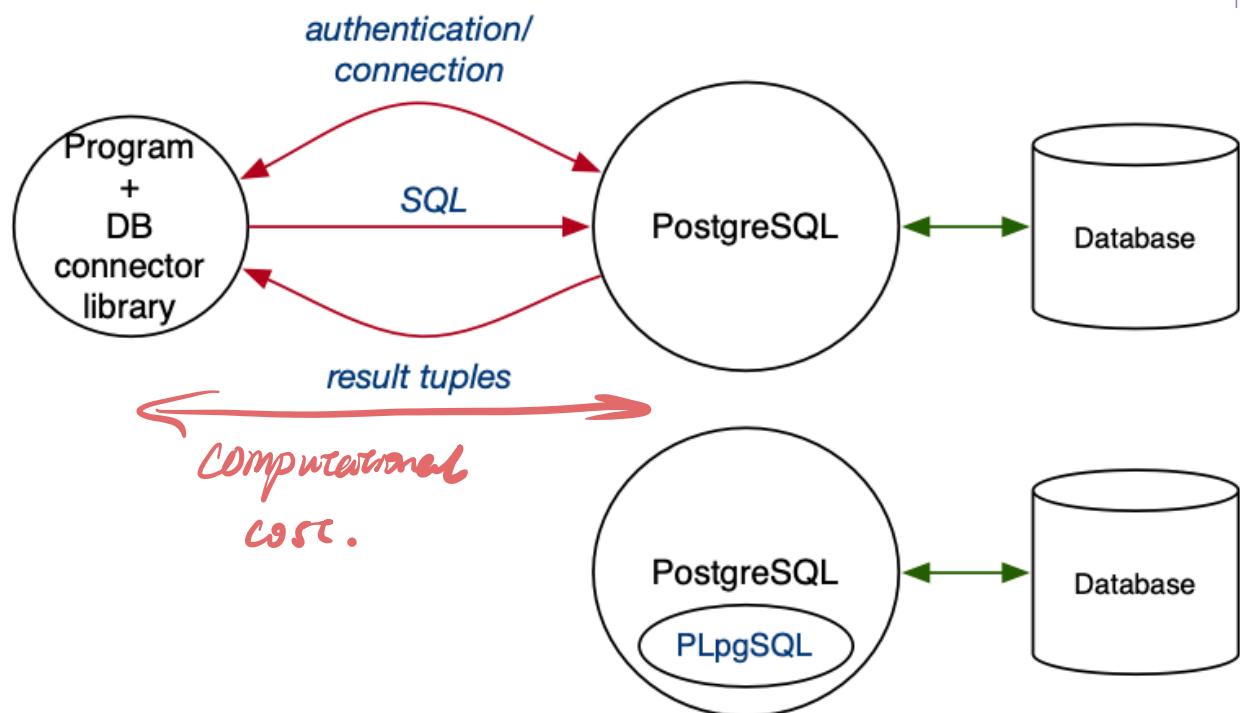
- mechanism for connecting to the DBMS (authentication)
- mechanism for mapping PL "requests" to DB queries
- mechanism for iterating over query results
- mapping between tuples and PL objects

Distance between PL and DBMS is variable, e.g.

- **libpq** allows C programs to use PG structs
- JDBC transmits SQL strings, retrieves tuples-as-objects

## ❖ Programming with Databases (cont)

Programming Language / DBMS architecture:



## ❖ PL/DB Interface

Common DB access API used in programming languages

```
db = connect_to_dbms(DBname, User/Password);  
  
query = build_SQL("SqlStatementTemplate", values);  
  
results = execute_query(db, query);  
  
while (more_tuples_in(results))  
{  
    tuple = fetch_row_from(results);  
    // do something with values in tuple ...  
}
```

This pattern is used in many different libraries:

- Java/JDBC, PHP/PDO, Perl/DBI, Python/Psycopg2, Tcl, ...

## ❖ PL/DB Interface (cont)

DB access libraries have similar overall structure.

But differ in the details:

- whether object-oriented or procedural flavour
- function/method names and parameters
- how to get data from program into SQL statements
- how to get data from tuples into program variables

Object-relational mappers (ORMs) ...

- aim to hide the details of the database schema and queries
- allow programmers to manipulate objects, not tuples
- potentially use the PLDB connection inefficiently

## ❖ PL/DB Mismatch

There is a tension between PLs and DBMSs

- DBMSs deal very efficiently with large sets of tuples
- PLs encourage dealing with single tuples/objects

If not handled carefully, can lead to inefficient use of DB.

Note: relative costs of DB access operations from PL:

- establishing a DBMS connection ... very high
- initiating an SQL query ... high
- accessing individual tuple ... small

## ❖ PL/DB Mismatch (cont)

Consider this (imaginary) PL/DBMS access method:

```
-- establish connection to DBMS
db = dbAccess("DB");
query = "select a,b from R,S where ... ";
-- invoke query and get handle to result set
results = dbQuery(db, query);
-- for each tuple in result set
while (tuple = dbNext(results)) {
    -- process next tuple
    process(tuple['a'], tuple['b']);
}
```

Estimated costs: **dbAccess** = 500ms, **dbQuery** = 200ms, **dbNext** = 10ms

In later cost estimates, ignore **dbAccess** ... same base cost for all examples

## ❖ PL/DB Mismatch (cont)

Example: find mature-age students (e.g. 10000 students, 500 over 40)

```
query = "select * from Student";
results = dbQuery(db, query);
while (tuple = dbNext(results)) {
    if (tuple['age'] >= 40) {
        -- process mature-age student
    }
}
```

We transfer 10000 tuples from DB, 9500 are irrelevant

Cost =  $1 \cdot 200 + 10000 \cdot 10 = 100200\text{ms} = 100\text{s}$

*that's why  
we often see  
our condition  
in query string.*

## ❖ PL/DB Mismatch (cont)

E.g. should be implemented as:



```
query = "select * from Student where age >= 40";  
results = dbQuery(db, query);  
while (tuple = dbNext(results)) {  
    -- process mature-age student  
}
```

Transfers only the 500 tuples that are needed.

Cost =  $1 \cdot 200 + 500 \cdot 10 = 5200\text{ms} = 5\text{s}$

## ❖ PL/DB Mismatch (cont)

Example: find info about all marks for all students

```

query1 = "select id,name from Student";
res1 = dbQuery(db, query1);
while (tuple1 = dbNext(res1)) {
    query2 = "select course,mark from Marks"
              + " where student = " + tuple1['id'];
    res2 = dbQuery(db,query2);
    while (tuple2 = dbNext(res2)) {
        -- process student/course/mark info
    }
}

```

*Nested loop  
bad sign.*

E.g. 10000 students, each with 8 marks,  $\Rightarrow$  run 10001 queries

Cost =  $10001 \times 200 + 80000 \times 10 = 2800s = 46min$

## ❖ PL/DB Mismatch (cont)

E.g. should be implemented as:

```
query = "select id,name,course,mark"  
       + " from Student s join Marks m "  
       + " on (s.id=m.student)"  
results = dbQuery(db, query);  
while (tuple = dbNext(results)) {  
    -- process student/course/mark info  
}
```

We invoke 1 query, and transfer same number of tuples.

Cost =  $1 * 200\text{ms} + 80000 * 10\text{ms} = 800\text{s} = 13\text{min}$



# Python (i)

---

- Python
- Python Basics
- More on Python

## ❖ Python

Python is a very popular programming language

- easy to learn/use
- with a wide range of useful libraries

We assume that you know enough Python to manipulate DBs

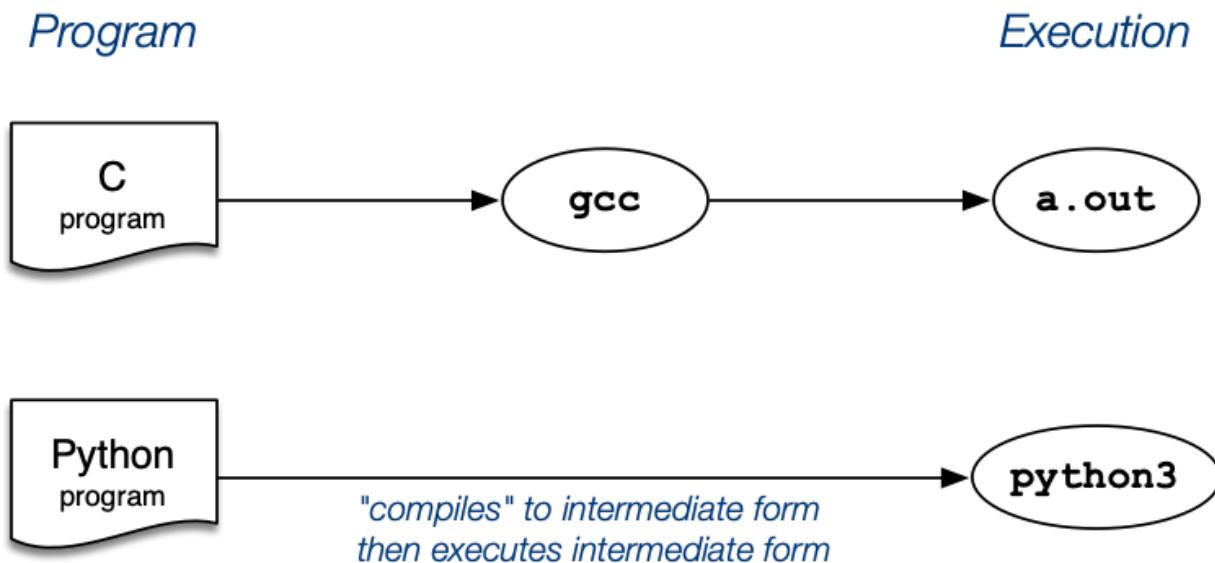
- the primary goal is Database, not Python programming

If you're not overly familiar with Python ...

- there will be many examples of Python code in this course
- there are many excellent tutorials online
- some of this content was "borrowed" from COMP1531 lectures

## ❖ Python (cont)

Unlike C, Python is an interpreted language



Such languages are often called "scripting languages"

## ❖ Python (cont)

Python has an interactive interface (like **psql**)

```
$ python3
Python 3.7.3 (default, Jul 25 2020, 13:03:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" ...
>> print("Hello, world")
Hello, world
>> quit()
$
```

Or you can run programs that are stored in files

```
$ echo 'print("Hello, world")' > hello.py
$ python3 hello.py
Hello, world
$
```

## ❖ Python Basics

Like C, Python programs consist of

- expressions, statements, control structures, function definitions, imports,  
...

Unlike C, Python uses indentation to indicate code nesting, e.g.

### Python

```
if Condition:  
    Statements1  
else:  
    Statements2  
Next Statement
```

### C

```
if (Condition) {  
    Statements1  
} else {  
    Statements2  
}  
Next Statement
```

## ❖ Python Basics (cont)

Comments are introduced by `#`, to end of line

Data types and constants:

- booleans, e.g. `True`, `False`
- numbers, e.g. `1`, `42`, `3.14`, `-5`
- strings, e.g. `"a string"`, `"string2"`, `'it's fun'`
- lists, e.g. `[1,4,9,16,25]`, `['a','b','c']`
- tuples, e.g. `(3,5)`, `(1,'a',3.0)`
- dictionaries, e.g. `{'a': 5, 'b': 98, 'c': 99}`

Assignment is via `=`, "the usual" operators are available

## ❖ Python Basics (cont)

Example operators and expressions

```
name = "Giraffe"
age = 18
height = 2048.11 # mm

print(name + ", " + str(age) + ', ' + str(height))
print(f"{name}, {age}, {height}")
print(type(name))
print(type(age))

n = 16 // 3
print(f"3 ** 3 == {3 ** 3}")
print(f"16 / 3 == {16 / 3}")
print(f"16 // 3 == {n}")
```

## ❖ Python Basics (cont)

### Defining functions

```
# recursive factorial

def fac(n):
    if n <= 1:
        return 1
    else:
        return n * fac(n-1)

print('5! =',fac(5))
```

## ❖ Python Basics (cont)

### Defining functions (cont)

```
# iterative factorial

def faci(n):
    f = 1
    for i in range(1,n):
        f = f * i
    return f

print('6! =',faci(6))
```

A collection of related functions can be packaged into a [module](#)

## ❖ Python Basics (cont)

C programs can import library definitions, e.g.

```
#include <stdlib.h>
#include <stdio.h>
```

Python programs can import external modules (module = collection of definitions)

```
import sys
import psycopg2
import sound.effects.echo
from sound.effects import echo
```

Packages (e.g. **sound**) are collections of sub-packages and modules

## ❖ Python Basics (cont)

Example: **echo** in Python

```
import sys
                    # extracts a slice from argv
for arg in sys.argv[1:] :
    print(arg, end=" ")    # don't put '\n' after print
print("")
```

which is used as (if placed in file **echo.py**)

```
$ python3 echo.py arg1 "arg 2" arg3
arg1 arg 2 arg3
$
```

## ❖ Python Basics (cont)

Example: sequence generator ... 1 2 3 4 5 ...

```
#!/usr/bin/python3
import sys
if len(sys.argv) < 3:
    print("Usage: seqq lo hi")
    exit(1)
hi = int(sys.argv[2])
i = lo = int(sys.argv[1])
while i <= hi:
    print(i)
    i += 1    # no ++ operator
```

which can be used as which is is used as (if placed in executable file  
**seqq**)

```
$ ./seqq 2 4
2
3
4
$
```

## ❖ More on Python

Lots of info available on **python.org**

- including [docs.python.org/3/tutorial/introduction.html](https://docs.python.org/3/tutorial/introduction.html)

And many others, e.g. [www.learnpython.org](http://www.learnpython.org)

Or ask for "free python3 tutorials" on Google

Python has hundreds of modules/libraries on all kinds of topics

We focus on the **psycopg2** database connectivity module



# Psycopg2

---

- Psycopg2
- Database **connections**
- Example: connecting to a database
- Operations on **connections**
- Database **Cursors**
- Operations on **cursors**

## ❖ Psycopg2

Psycopg2 is a Python module that provides

- a method to connect to PostgreSQL databases
- a collection of DB-related exceptions
- a collection of type and object constructors

In order to use Psycopg2 in a Python program

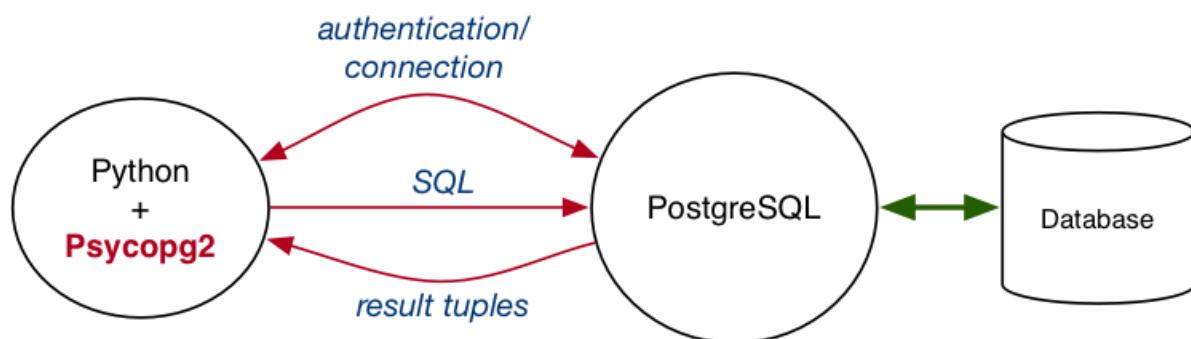
```
import psycopg2
```

Note:

- assumes that the **psycopg2** module is installed on your system
- Psycopg2 is installed on Grieg; installing on a Mac is relatively easy

## ❖ Psycopg2 (cont)

Where **psycopg2** fits in the PL/DB architecture



## ❖ Database connections

```
conn = psycopg2.connect(DB_connection_string)
```

- creates a **connection** object on a named database
- effectively starts a session with the database (cf **psql**)
- returns a **connection** object used to access DB
- if can't connect, raises an exception

DB connection string components

- **dbname** ... name of database
- **user** ... user name (for authentication)
- **password** ... user password (for authentication)
- **host** ... where is the server running (default=localhost)
- **port** ... which port is server listening on (default=5432)

On Grieg, only **dbname** is required.

## ❖ Example: connecting to a database

Simple script that connects and then closes connection

```
import psycopg2

try:
    conn = psycopg2.connect("dbname=mydb")
    print(conn) # state of connection after opening
    conn.close()
    print(conn) # state of connection after closing
except Exception as e:
    print("Unable to connect to the database")
```

which, if **mydb** is accessible, produces output:

```
$ python3 ex1.py
<connection object at 0xf67186ec; dsn: 'dbname=mydb', closed: 0>
<connection object at 0xf67186ec; dsn: 'dbname=mydb', closed: 1>
```

## ❖ Example: connecting to a database (cont)

Example: change the script to get database name from command line

```
import sys
import psycopg2

if len(sys.argv) < 2:
    print("Usage: opendb DBname")
    exit(1)
db = sys.argv[1]
try:
    conn = psycopg2.connect("dbname=" + db)
    print(conn)
    conn.close()
    print(conn)
except Exception as e:
    print(f"Unable to connect to database {db}")
```

## ❖ Operations on connections

`cur = conn.cursor()`

*handler.*

- set up a handle for performing queries/updates on database
- must create a **cursor** before performing any DB operations

`conn.close()`

- close the database connection **conn**

`conn.commit()`

- commit changes made to database since last **commit()**

Plus many others ... see Psycopg2 documentation

## ❖ Database Cursors

Database Object.

Cursors are "pipelines" to the database

**Cursor** objects allow you to ...

- execute queries, perform updates, change meta-data

Cursors are created from a database **connection**

- can create multiple cursors from the same connection
- each cursor handles one DB operation at a time
- but cursors are not isolated (can see each others' changes)

To set up a **cursor** object called **cur** ...

```
cur = conn.cursor()
```

## ❖ Operations on cursors

**cur.execute(*SQL\_statement, Values*)**

- if supplied, insert values into the SQL statement
- then execute the SQL statement
- results are available via the cursor's fetch methods

Examples:

```
# run a fixed query
cur.execute("select * from R where x = 1");
```

*See next page.*

*For multiple input.*

*emphasise it as tuple.*

```
# run a query with values inserted
cur.execute("select * from R where x = %s", (1,))
cur.execute("select * from R where x = %s", [1])
```

```
# run a query stored in a variable
query = "select * from Students where name ilike %s"
pattern = "%mith%"
cur.execute(query, [pattern])
```

## ❖ Operations on cursors (cont)

### `cur.mogrify(SQL_statement, Values)`

- return the SQL statement as a string, with values inserted
- useful for checking whether `execute()` is doing what you want

Examples:

```
query = "select * from R where x = %s"
print(cur.mogrify(query, [1]))
Produces: b'select * from R where x = 1'
```

```
query = "select * from R where x = %s and y = %s"
print(cur.mogrify(query, [1,5]))
Produces: b'select * from R where x = 1 and y = 5'
```

```
query = "select * from Students where name ilike %s"
pattern = "%mith%"
print(cur.mogrify(query, [pattern]))
Produces: b"select * from Students where name ilike '%mith%'"
```

```
query = "select * from Students where family = %s"
fname = "O'Reilly"
print(cur.mogrify(query, [fname]))
Produces: b"select * from Students where family = 'O''Reilly'"
```

## ❖ Operations on cursors (cont)

```
list = cur.fetchall()
```

- gets all answers for a query and stores in a list of tuples
- can iterate through list of results using Python's **for**

Example:

```
# table R contains (1,2), (2,1), ...
cur.execute("select * from R")
for tup in cur.fetchall():
    x,y = tup
    print(x,y)    # or print(tup[0],tup[1])

# prints
1 2
2 1
...
```

## ❖ Operations on cursors (cont)

- tup = cur.fetchone()** *More flexible for fetchone,  
for special needs.*
- gets next result for a query and stores in a tuple
  - can iterate through list of results using Python's **while**

Example:

```
# table R contains (1,2), (2,1), ...
cur.execute("select * from R")
while True:
    t = cur.fetchone()
    if t == None:
        break
    x,y = tup
    print(x,y)

# prints
1 2
2 1
...
```

## ❖ Operations on cursors (cont)

**tup = cur.fetchmany(*nTuples*)**

- gets next *nTuples* results for a query
- stores tuples in a list
- when no results left, returns empty list

Example:

```
# table R contains (1,2), (2,1), ...  
  
cur.execute("select * from R")  
while True:  
    tups = cur.fetchmany(3)  
    if tups == []:  
        break  
    for tup in tups:  
        x,y = tup  
        print(x,y)  
  
# prints  
1 2  
2 1  
...
```



## Python (ii)

- Python + Psycopg2 (recap)
- Examples
- Poor Usage of Python+SQL
- Calling PostgreSQL functions
- Other Psycopg2 Tricks

## ❖ Python + Psycopg2 (recap)

**psycopg2** is a Python module providing access to PostgreSQL DBs

Standard usage:

```
import psycopg2    # include the module definitions
try:
    connection = psycopg2.connect("dbname=Datatase")
    cursor = connection.cursor()
    cursor.execute("SQL Query")
    for tuple in cursor.fetchall():
        # do something with next tuple
    cursor.close()
    connection.close()
except:
    print("Database error")
```

These slides aim to give more details on how Pyscopg2 used in practice

## ❖ Python + Psycopg2 (recap) (cont)

### **connection**

- handle giving authenticated access for a given user on a given DB
- provides creation of **cursor**s to interact with database

### **cursor**

- pipeline between a Python program and a PostgreSQL DB
- send SQL statements down pipeline as strings
- read results up pipeline as Python (list of) tuples

## ❖ Python + Psycopg2 (recap) (cont)

Python vs PostgreSQL data types ...

Strings:

- in Python: written with "..." or '...', including \x
- converted to SQL strings e.g. "O'Reilly" → 'O''Reilly'
- Python supports """ ..... """ multi-line strings (useful for SQL queries)

Tuples:

- in Python: contain multiple heterogeneous values (cf. C **struct**)
- similar to PostgreSQL composite (tuple) types
- written as: ( val<sub>1</sub>, val<sub>2</sub>, ..., val<sub>n</sub> ) (note that ( val<sub>1</sub> ) is not a tuple)
- examples: (1, 2, 3), (1, "John", 3.14), (1,),

## ❖ Examples

Example database: **beers2**

```
Beers( id:int, name:text, brewer:int )  
  
Brewers( id:int, name:text, country:text )  
  
Bars( id:int, name:text, addr:text, license:int )  
  
Drinkers( id:int, name:text, addr:text, phone:text )  
  
Likes( drinker:int, beer:int )  
  
Sells( bar:int, beer:int, price:float )  
  
Frequents( drinker:int, bar:int )
```

## ❖ Examples (cont)

Assume that the following code samples are wrapped in

```
import sys
import psycopg2
conn = None
try:
    conn = psycopg2.connect("dbname=beers2")
    ... example code ...
except psycopg2.Error as err:
    print("database error:",err)
finally:
    if (conn):
        conn.close()
print("finished with database")
```

## ❖ Examples (cont)

Example: a list of brewers and their countries as **brewers.py**

```
cur = conn.cursor()  
cur.execute("""  
    select name, country from Brewers order by name  
""")  
for tuple in cur.fetchall():  
    name, country = tuple  
    print(name + ", " + country)
```

\$ **python3 brewers.py**

```
Brew Dog, Scotland  
Bridge Road Brewers, Australia  
Caledonian, Scotland  
Carlton, Australia  
Cascade, Australia  
...  
...
```

## ❖ Examples (cont)

Example: a list of brewers and their countries as **bfrom.py**

```
cur = conn.cursor()
qry = "select name from Brewers where country = %s"
country = sys.argv[1]
cur.execute(qry, [country])
for tuple in cur.fetchall():
    print(tuple[0])

$ python3 bfrom.py Scotland
Caledonian
Brew Dog
```

## ❖ Examples (cont)

Example: print beers preceded by the brewer as **beers.py**

```
cur = conn.cursor()
qry = """
select b.name, r.name
from   Brewers r join Beers b on (b.brewer=r.id)
"""

cur.execute(qry)
for tuple in cur.fetchall():
    print(tuple[1] + " " + tuple[0])

$ python3 beers.py
Caledonian 80/-
James Squire Amber Ale
Sierra Nevada Bigfoot Barley Wine
... 
```

## ❖ Examples (cont)

Example: most expensive beer as **expensive.py**

```
cur = conn.cursor()
qry = """
select b.name, s.price
from Beers b join Sells s on (b.id = s.beer)
where s.price = (select max(price) from Sells)
"""

cur.execute(qry)
for tuple in cur.fetchall():
    print(tuple[0] + " @ " + str(tuple[1]))
```

\$ **python3 beers.py**

Sink the Bismarck @ 25.0

## ❖ Examples (cont)

Example: list beers, bar+price where sold, average price as  
**beers1.py**

```
$ python3 beers1.py
...
New
    Australia Hotel @ 3.0
    Coogee Bay Hotel @ 2.25
    Lord Nelson @ 3.0
    Marble Bar @ 2.8
    Regent Hotel @ 2.2
    Royal Hotel @ 2.3
    Average @ 2.591666666666667
Nirvana Pale Ale
    Not sold anywhere
Old
    Coogee Bay Hotel @ 2.5
    Marble Bar @ 2.9
    Royal Hotel @ 2.65
    Average @ 2.683333333333336
Old Admiral
    Lord Nelson @ 3.75
    Average @ 3.75
...
```

## ❖ Examples (cont)

```
cur = conn.cursor()
qry = "select id, name from Beers"
cur.execute(qry)
for tuple in cur.fetchall():
    q2 = """select b.name, s.price
            from Bars b join Sells s on (b.id=s.bar)
            where s.beer = %s"""
    print(tuple[1])
    cur.execute(q2, [tuple[0]])
    n, tot = 0, 0.0
    for t in cur.fetchall():
        print("\t"+t[0], "@", t[1])
        n = n + 1
        tot = tot + t[1]
    if n > 0:
        print("\tAverage @", tot/n)
    else:
        print("\tNot sold anywhere")
```

## ❖ Poor Usage of Python+SQL

Should generally avoid

```
cur.execute("select x,y from R")
for tup in cur.fetchall():
    q = "select * from S where id=%s"
    cur.execute(q, [tup[0]])
    for t in cur.fetchall():
        ... process t ...
```

More efficiently done as e.g.

```
qry = """
select *
from   R join S on (R.x = S.id)
"""

for tup in cur.fetchall():
    ... process tup ...
```

## ❖ Calling PostgreSQL functions

Two ways to call PostgreSQL functions

```
# using a standard function call from SQL
cur.execute("select * from brewer(5)")
t = cur.fetchone()
print(t[0])

# using special callproc() method
# parameters supplied as a list of values/vars
cur.callproc("brewer",[5])
t = cur.fetchone()
print(t[0])
```

**brewer(int) returns text** returns a brewer's name, given their id

## ❖ Other Psycopg2 Tricks

`cur.execute(SQL Statement)`

- clearly the SQL statement can be **SELECT**
- can also be **UPDATE** or **DELETE**
- can also be a meta-data statement, e.g.
  - **CREATE TABLE**, **DROP TABLE**, **CREATE VIEW**, ...

`cur.fetchmany(#tuples)`

- gets a list of the next **#tuples** tuples
- could replace PLpgSQL **LIMIT** in some contexts

For many more examples, see Psycopg2 documentation and tutorials



# Relational Design

---

- Relational Design Theory
- Relational Design and Redundancy
- Database Design (revisited)

## ❖ Relational Design Theory

The aim of studying relational design theory:

- improve understanding of relationships among data
- gain enough formalism to assist practical database design

What we study here:

- basic theory and definition of **functional dependencies**
- methodology for improving schema designs  
**(normalisation)**

Functional dependencies

- describe relationships between attributes within a relation
- have implications for "good" relational schema design

## ❖ Relational Design and Redundancy

A **good** relational database design:

- must capture *all* necessary attributes/associations
- do this with *minimal* amount of stored information

Minimal stored information  $\Rightarrow$  no redundant data.

In database design, **redundancy** is generally a "bad thing":

- causes problems maintaining consistency after updates

But ... redundancy may give performance improvements

- e.g. avoid a join to collect pieces of data together

## ❖ Relational Design and Redundancy

(cont)

Consider the following relation defining bank accounts/branches:

<b>accountNo</b>	<b>balance</b>	<b>customer</b>	<b>branch</b>	<b>address</b>	<b>assets</b>
A-101	500	1313131	Downtown	Brooklyn	9000000
A-102	400	1313131	Perryridge	Horseneck	1700000
A-113	600	9876543	Round Hill	Horseneck	8000000
A-201	900	9876543	Brighton	Brooklyn	7100000
A-215	700	1111111	Mianus	Horseneck	400000
A-222	700	1111111	Redwood	Palo Alto	2100000
A-305	350	1234567	Round Hill	Horseneck	8000000
...	...	...	...	...	...

Careless updating of this data may introduce inconsistencies.

## ❖ Relational Design and Redundancy

(cont)

If we add \$300 to account A-113 ...

<b>accountNo</b>	<b>balance</b>	<b>customer</b>	<b>branch</b>	<b>address</b>	<b>assets</b>
A-101	500	1313131	Downtown	Brooklyn	9000000
A-102	400	1313131	Perryridge	Horseneck	1700000
A-113	900	9876543	Round Hill	Horseneck	8000300
A-201	900	9876543	Brighton	Brooklyn	7100000
A-215	700	1111111	Mianus	Horseneck	400000
A-222	700	1111111	Redwood	Palo Alto	2100000
A-305	350	1234567	Round Hill	Horseneck	8000000
...	...	...	...	...	...

## ❖ Relational Design and Redundancy

(cont)

If we add a new account A-306 at the Round Hill branch ...

<b>accountNo</b>	<b>balance</b>	<b>customer</b>	<b>branch</b>	<b>address</b>	<b>assets</b>
A-101	500	1313131	Downtown	Brooklyn	9000000
A-102	400	1313131	Perryridge	Horseneck	1700000
A-113	900	9876543	Round Hill	Horseneck	8000300
A-201	900	9876543	Brighton	Brooklyn	7100000
A-215	700	1111111	Mianus	Horseneck	400000
A-222	700	1111111	Redwood	Palo Alto	2100000
A-305	350	1234567	Round Hill	Horseneck	8000000
<b>A-306</b>	<b>500</b>	<b>7654321</b>	<b>Round Hill</b>	<b>Horseneck</b>	<b>8000500?</b>
...	...	...	...	...	...

## ❖ Relational Design and Redundancy

(cont)

If we close account A-101 ...

<b>accountNo</b>	<b>balance</b>	<b>customer</b>	<b>branch</b>	<b>address</b>	<b>assets</b>
A-101	500	1313131	Downtown	Brooklyn	9000000
A-102	400	1313131	Perryridge	Horseneck	1700000
A-113	900	9876543	Round Hill	Horseneck	8000300
A-201	900	9876543	Brighton	Brooklyn	7100000
A-215	700	1111111	Mianus	Horseneck	400000
A-222	700	1111111	Redwood	Palo Alto	2100000
A-305	350	1234567	Round Hill	Horseneck	8000000
<b>A-306</b>	500	7654321	Round Hill	Horseneck	8000500?
...	...	...	...	...	...

What is the address of the Downtown branch?

## ❖ Relational Design and Redundancy

(cont)

Insertion anomaly:

- when we insert a new record, we need to check that branch data is consistent with existing tuples

Update anomaly:

- if a branch changes address, we need to update all tuples referring to that branch

Deletion anomaly:

- if we remove information about the last account at a branch, all of the branch information disappears

Insertion/update anomalies can be handled, e.g. by triggers

- but this requires extra DBMS work on every change to the database

## ❖ Database Design (revisited)

To avoid these kinds of update problems:

- need a schema with "minimal overlap" between tables
- each table contains a "coherent" collection of data values

Such schemas have little/no redundancy

ER → SQL mapping tends to give non-redundant schemas

- but does not guarantee no redundancy

The methods we describe in this section

- can reduce redundancy in schemas ⇒ eliminate update anomalies



# Functional Dependency

---

- Notation/Terminology
- Functional Dependency
- Exercise: Functional Dependencies (i)
- Functional Dependency (again)
- Inference Rules

## ❖ Notation/Terminology

Most texts adopt the following terminology:

Attributes	upper-case letters from start of alphabet (e.g. $A, B, C, \dots$ )
Sets of attributes	concatenation of attribute names (e.g. $X=ABCD, Y=EFG$ )
Relation schemas	upper-case letters, denoting set of all attributes (e.g. $R$ )
Relation instances	lower-case letter corresponding to schema (e.g. $r(R)$ )
Tuples	lower-case letters (e.g. $t, t', t_1, u, v$ )
Attributes in tuples	tuple[attrSet] (e.g. $t[ABCD], t[X]$ )

## ❖ Functional Dependency

A relation instance  $r(R)$  satisfies a dependency  $X \rightarrow Y$  if

- for any  $t, u \in r$ ,  $t[X] = u[X] \Rightarrow t[Y] = u[Y]$

In other words, if two tuples in  $R$  agree in their values for the set of attributes  $X$ , then they must also agree in their values for the set of attributes  $Y$ .

We say that " $Y$  is **functionally dependent** on  $X$ ".

Attribute sets  $X$  and  $Y$  may overlap; it is trivially true that  $X \rightarrow X$ .

Notes:

- $X \rightarrow Y$  can also be read as " $X$  determines  $Y$ "
- the single arrow  $\rightarrow$  denotes **functional dependency**
- the double arrow  $\Rightarrow$  denotes logical implication

## ❖ Functional Dependency (cont)

Consider the following (redundancy-laden) relation:

Title	Year	Len	Studio	Star
King Kong	1933	100	RKO	Fay Wray
King Kong	1976	134	Paramount	Jessica Lange
King Kong	1976	134	Paramount	Jeff Bridges
Mighty Ducks	1991	104	Disney	Emilio Estevez
Wayne's World	1995	95	Paramount	Dana Carvey
Wayne's World	1995	95	Paramount	Mike Meyers

Some functional dependencies in this relation

- **Title Year → Len, Title Year → Studio**

**Not** a functional dependency

- **Title Year → Star**

## ❖ Functional Dependency (cont)

Consider an instance  $r(R)$  of the relation schema  $R(ABCDE)$ :

<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>
<i>a1</i>	<i>b1</i>	<i>c1</i>	<i>d1</i>	<i>e1</i>
<i>a2</i>	<i>b1</i>	<i>c2</i>	<i>d2</i>	<i>e1</i>
<i>a3</i>	<i>b2</i>	<i>c1</i>	<i>d1</i>	<i>e1</i>
<i>a4</i>	<i>b2</i>	<i>c2</i>	<i>d2</i>	<i>e1</i>
<i>a5</i>	<i>b3</i>	<i>c3</i>	<i>d1</i>	<i>e1</i>

Since  $A$  values are unique, the definition of  $fd$  gives:

- $A \rightarrow B, A \rightarrow C, A \rightarrow D, A \rightarrow E, \text{ or } A \rightarrow BCDE$

Since all  $E$  values are the same, it follows that:

- $A \rightarrow E, B \rightarrow E, C \rightarrow E, D \rightarrow E$

## ❖ Functional Dependency (cont)

Other observations:

- combinations of  $BC$  are unique, therefore  $BC \rightarrow ADE$
- combinations of  $BD$  are unique, therefore  $BD \rightarrow ACE$
- if  $C$  values match, so do  $D$  values, therefore  $C \rightarrow D$
- however,  $D$  values don't determine  $C$  values, so  $!(D \rightarrow C)$

We could derive many other dependencies, e.g.  $AE \rightarrow BC, \dots$

In practice, choose a minimal set of *fds* (**basis**)

- from which all other *fds* can be derived
- which captures useful problem-domain information

## ❖ Exercise: Functional Dependencies (i)

Real estate agents conduct visits to rental properties

- need to record which property, who went, when, results
- each property is assigned a unique code (P#, e.g. P4)
- each staff member has a staff number (S#, e.g. S43)
- staff members use company cars to conduct visits
- a visit occurs at a specific time on a given day
- notes are made on the state of the property after each visit

The company stores all of the associated data in a spreadsheet.

## ❖ Exercise: Functional Dependencies (i) (cont)

The spreadsheet ...

P#	When	Address	Notes	S#	Name	CarReg
P4	03/06 15:15	55 High St	Bathroom leak	S44	Rob	ABK754
P1	04/06 11:10	47 High St	All ok	S44	Rob	ABK754
P4	03/07 12:30	55 High St	All ok	S43	Dave	ATS123
P1	05/07 15:00	47 High St	Broken window	S44	Rob	ABK754
P1	05/07 15:00	47 High St	Leaking tap	S44	Rob	ABK754
P2	13/07 12:00	12 High St	All ok	S42	Peter	ATS123
P1	10/08 09:00	47 High St	Window fixed	S42	Peter	ATS123
P3	11/08 14:00	99 High St	All ok	S41	John	AAA001
P4	13/08 10:00	55 High St	All ok	S44	Rob	ABK754
P3	05/09 11:15	99 High St	Bathroom leak	S42	Peter	ATS123

Functional dependencies:  $P \rightarrow A$ ,  $A \rightarrow P$ ,  $S \rightarrow m$ ,  $S \rightarrow C$

## ❖ Functional Dependency (again)

Above examples consider dependency within a relation instance  $r(R)$ .

More important for *design* is dependency across all possible instances of the relation (i.e. a schema-based dependency).

This is a simple generalisation of the previous definition:

- for any  $t, u \in \text{any } r(R)$ ,  $t[X] = u[X] \Rightarrow t[Y] = u[Y]$

Such dependencies tend to capture semantics of problem domain.

E.g. real estate example

- $P \rightarrow A$  suggests a property entity,  $S \rightarrow N$ ,  $S \rightarrow C$  suggest a staff entity
- Property(P#,addr), Staff(S#,name,car), Inspection(P#,S#,when,notes)

## ❖ Functional Dependency (again) (cont)

Can we generalise some ideas about functional dependency?

E.g. are there dependencies that hold for *any* relation?

- yes, but they're generally trivial, e.g.  $Y \subset X \Rightarrow X \rightarrow Y$

E.g. do some dependencies suggest the existence of others?

- yes, **rules of inference** allow us to **derive** dependencies
- allow us to reason about sets of functional dependencies

## ❖ Inference Rules

Armstrong's rules are general rules of inference on *fds*.

F1. **Reflexivity** e.g.  $X \rightarrow X$

- a formal statement of *trivial dependencies*; useful for derivations

F2. **Augmentation** e.g.  $X \rightarrow Y \Rightarrow XZ \rightarrow YZ$

- if a dependency holds, then we can expand its left hand side (along with RHS)

F3. **Transitivity** e.g.  $X \rightarrow Y, Y \rightarrow Z \Rightarrow X \rightarrow Z$

- the "most powerful" inference rule; useful in multi-step derivations

## ❖ Inference Rules (cont)

Armstrong's rules are complete, but other useful rules exist:

F4. **Additivity** e.g.  $X \rightarrow Y, X \rightarrow Z \Rightarrow X \rightarrow YZ$

- useful for constructing new right hand sides of *fds* (also called **union**)

F5. **Projectivity** e.g.  $X \rightarrow YZ \Rightarrow X \rightarrow Y, X \rightarrow Z$

- useful for reducing right hand sides of *fds* (also called **decomposition**)

F6. **Pseudotransitivity** e.g.  $X \rightarrow Y, YZ \rightarrow W \Rightarrow XZ \rightarrow W$

- shorthand for a common transitivity derivation

## ❖ Inference Rules (cont)

Example: determining validity of  $AB \rightarrow GH$ , given:

$$R = ABCDEFGHIJ$$

$$F = \{AB \rightarrow E, AG \rightarrow J, BE \rightarrow I, E \rightarrow G, GI \rightarrow H\}$$

Derivation:

1.  $AB \rightarrow E$  (given)
2.  $E \rightarrow G$  (given)
3.  $AB \rightarrow G$  (using F3 on 1,2)
4.  $AB \rightarrow AB$  (using F1)
5.  $AB \rightarrow B$  (using F5 on 4) *Productivity.*
6.  $AB \rightarrow BE$  (using F4 on 1,5) *Additivity.*
7.  $BE \rightarrow I$  (given)
8.  $AB \rightarrow I$  (using F3 on 6,7)
9.  $AB \rightarrow GI$  (using F4 on 3,8)
10.  $GI \rightarrow H$  (given)
11.  $AB \rightarrow H$  (using F3 on 9,10)
12.  $AB \rightarrow GH$  (using F4 on 3,11)



# Inference on FDs

---

- Closures
- Determining Keys
- Minimal Covers

## ❖ Closures

Given a set  $F$  of  $fds$ , how many new  $fds$  can we derive?

For a finite set of attributes, there must be a finite set of derivable  $fds$ .

The largest collection of dependencies that can be derived from  $F$  is called the **closure** of  $F$  and is denoted  $F^+$ .

Closures allow us to answer two interesting questions:

- is a particular dependency  $X \rightarrow Y$  derivable from  $F$ ?
- are two sets of dependencies  $F$  and  $G$  equivalent?

## ❖ Closures (cont)

---

For the question "is  $X \rightarrow Y$  derivable from  $F$ ?" ...

- compute the closure  $F^+$ ; check whether  $X \rightarrow Y \in F^+$

For the question "are  $F$  and  $G$  equivalent?" ...

- compute closures  $F^+$  and  $G^+$ ; check whether they're equal

Unfortunately, closures can be very large, e.g.

$$R = ABC, \quad F = \{AB \rightarrow C, \quad C \rightarrow B\}$$

$$\begin{aligned} F^+ = & \{A \rightarrow A, \quad AB \rightarrow A, \quad AC \rightarrow A, \quad AB \rightarrow B, \quad BC \rightarrow B, \quad ABC \rightarrow B, \\ & C \rightarrow C, \quad AC \rightarrow C, \quad BC \rightarrow C, \quad ABC \rightarrow C, \quad AB \rightarrow AB, \quad \dots\dots, \\ & AB \rightarrow ABC, \quad AB \rightarrow ABC, \quad C \rightarrow B, \quad C \rightarrow BC, \quad AC \rightarrow B, \quad AC \rightarrow AB\} \end{aligned}$$

## ❖ Closures (cont)

Algorithms based on  $F^+$  rapidly become infeasible.

To solve this problem ...

- use closures based on sets of attributes rather than sets of *fds*.

Given a set  $X$  of attributes and a set  $F$  of *fds*, the **closure** of  $X$  (denoted  $X^+$ ) is

- the largest set of attributes that can be derived from  $X$  using  $F$

Determining  $X^+$  from  $\{X \rightarrow Y, Y \rightarrow Z\} \dots X \rightarrow XY \rightarrow XYZ = X^+$

For computation,  $|X^+|$  is bounded by the number of attributes.

## ❖ Closures (cont)

Algorithm for computing attribute closure:

```
Input: F (set of FDs), X (starting attributes)
Output: X+ (attribute closure)
```

```
Closure = X
while (not done) {
    OldClosure = Closure
    for each A → B such that A ⊂ Closure
        add B to Closure
    if (Closure == OldClosure) done = true
}
```

## ❖ Closures (cont)

For the question "is  $X \rightarrow Y$  derivable from  $F$ ?" ...

- compute the closure  $X^+$ , check whether  $Y \subset X^+$

For the question "are  $F$  and  $G$  equivalent?" ...

- for each dependency in  $G$ , check whether derivable from  $F$
- for each dependency in  $F$ , check whether derivable from  $G$
- if true for all, then  $F \Rightarrow G$  and  $G \Rightarrow F$  which implies  $F^+ = G^+$

For the question "what are the keys of  $R$  implied by  $F$ ?" ...

- find subsets  $K \subset R$  such that  $K^+ = R$

## ❖ Determining Keys *⇒ Able to determine all keys.*

Example: determine primary keys for each of the following:

1.  $FD = \{A \rightarrow B, C \rightarrow D, E \rightarrow FG\}$

- A?  $A^+ = AB$ , so no ... AB?  $AB^+ = ABCD$ , so no
- ACE?  $ACE^+ = ABCDEFG$ , so yes!

2.  $FD = \{A \rightarrow B, B \rightarrow C, C \rightarrow D\}$

- B?  $B^+ = BCD$ , so no ... A?  $A^+ = ABCD$ , so yes!

3.  $FD = \{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$

- A?  $A^+ = ABC$ , so yes! ... B?  $B^+ = ABC$ , so yes!

## ❖ Minimal Covers

For a given application, we can define many different sets of *fds* with the same closure (e.g.  $F$  and  $G$  where  $F^+ = G^+$ )

Which one is best to "model" the application?

- any model has to be complete (i.e. capture entire semantics)
- models should be as small as possible  
(we use them to check DB validity after update; less checking is better)

If we can ...

- determine a number of candidate *fd* sets,  $F$ ,  $G$  and  $H$
- establish that  $F^+ = G^+ = H^+$
- we would then choose the smallest one for our "model"

Better still, can we *derive* the smallest complete set of *fds*?

## ❖ Minimal Covers (cont)

Minimal cover  $F_c$  for a set  $F$  of  $fd$ s:

- $F_c$  is equivalent to  $F$
- all  $fd$ s have the form  $X \rightarrow A$  (where  $A$  is a single attribute)
- it is not possible to make  $F_c$  smaller
  - either by deleting an  $fd$
  - or by deleting an attribute from an  $fd$

An  $fd$   $d$  is redundant if  $(F - \{d\})^+ = F^+$

An attribute  $a$  is redundant if  $(F - \{d\} \cup \{d'\})^+ = F^+$   
(where  $d'$  is the same as  $d$  but with attribute  $A$  removed)

## ❖ Minimal Covers (cont)

Algorithm for computing minimal cover:

**Inputs:** set  $F$  of *fds*

**Output:** minimal cover  $F_C$  of  $F$

$F_C = F$

Step 1: put  $f \in F_C$  into canonical form

Step 2: eliminate redundant attributes from  $f \in F_C$

Step 3: eliminate redundant *fds* from  $F_C$

Step 1: put *fd's* into canonical form

```
for each  $f \in F_C$  like  $X \rightarrow \{A_1, \dots, A_n\}$ 
    remove  $X \rightarrow \{A_1, \dots, A_n\}$  from  $F_C$ 
    add  $X \rightarrow A_1, \dots, X \rightarrow A_n$  to  $F_C$ 
end
```

## ❖ Minimal Covers (cont)

Step 2: eliminate redundant attributes

```

for each  $f \in F_C$  like  $X \rightarrow A$ 
    for each  $b$  in  $X$ 
         $f' = (X - \{b\}) \rightarrow A$ ;  $G = F_C - \{f\} \cup \{f'\}$ 
        if  $(G^+ == F_C^+)$   $F_C = G$ 
    end
end

```

Step 3: eliminate redundant functional dependencies

```

for each  $f \in F_C$ 
     $G = F_C - \{f\}$ 
    if  $(G^+ == F_C^+)$   $F_C = G$ 
end

```

## ❖ Minimal Covers (cont)

Example: compute minimal cover

E.g.  $R = ABC, F = \{A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C\}$

Working ...

- canonical *fds*:  $A \rightarrow B, A \rightarrow C, B \rightarrow C, AB \rightarrow C$
- redundant attrs:  $A \rightarrow B, A \rightarrow C, B \rightarrow C, A\textcolor{red}{B} \rightarrow C$
- redundant *fds*:  $A \rightarrow B, \textcolor{red}{A \rightarrow C}, B \rightarrow C$

This gives the minimal cover  $F_C = \{A \rightarrow B, B \rightarrow C\}$ .

---

Produced: 4 Nov 2020

# Normal Forms

---

- Normalisation
- Normal Forms
- Boyce-Codd Normal Form
- Third Normal Form

## ❖ Normalisation

**Normalisation:** branch of relational theory providing design insights.

The goals of normalisation:

- be able to characterise the level of redundancy in a relational schema
- provide mechanisms for transforming schemas to remove redundancy

Normalisation draws heavily on the theory of functional dependencies.

Normalisation algorithms reduce the amount of redundancy in a schema

- by decomposition (break schema into connected pieces)

## ❖ Normal Forms

Normalisation theory defines six **normal forms** (NFs).

- First,Second,Third Normal Forms (1NF,2NF,3NF) (Codd 1972)
- Boyce-Codd Normal Form (BCNF) (1974)
- Fourth Normal Form (4NF) (Zaniolo 1976, Fagin 1977)
- Fifth Normal Form (5NF) (Fagin 1979)

We say that "a schema is in xNF", which ...

- tells us something about the level of redundancy in the schema

1NF allows most redundancy; 5NF allows least redundancy.

For most practical purposes, BCNF (or 3NF) are acceptable NFs.

## ❖ Normal Forms (cont)

- |      |   |
|------|---|
| 1NF  | all attributes have atomic values<br>we assume this as part of relational model,<br>so <i>every</i> relation schema is in 1NF |
| 2NF  | all non-key attributes fully depend on key<br>(i.e. no partial dependencies)<br>avoids much redundancy                        |
| 3NF  | no attributes dependent on non-key attrs  |
| BCNF | (i.e. no transitive dependencies)<br>avoids most remaining redundancy   |

## ❖ Normal Forms (cont)

In practice, BCNF and 3NF are the most important.

Boyce-Codd Normal Form (BCNF):

- eliminates all redundancy due to functional dependencies
- but may not preserve original functional dependencies

Third Normal Form (3NF):

- eliminates most (but not all) redundancy due to *fds*
- guaranteed to preserve all functional dependencies

## ❖ Boyce-Codd Normal Form

A relation schema  $R$  is in BCNF w.r.t a set  $F$  of functional dependencies iff:

for all  $fds X \rightarrow Y$  in  $F^+$

- either  $X \rightarrow Y$  is trivial (i.e.  $Y \subset X$ )
- or  $X$  is a superkey (i.e. non-strict superset of attributes in key)

A DB schema is in BCNF if all of its relation schemas are in BCNF.

Observations:

- any two-attribute relation is in BCNF
- any relation with key  $K$ , other attributes  $Y$ , and  $K \rightarrow Y$  is in BCNF

## ❖ Boyce-Codd Normal Form (cont)

If we transform a schema into BCNF, we are guaranteed:

- no update anomalies due to *fd*-based redundancy
- lossless join decomposition

However, we are **not** guaranteed:

- the new schema preserves all *fds* from the original schema

This may be a problem if the *fds* contain significant semantic information about the problem domain (use 3NF to preserve dependencies)

A dependency  $A \rightarrow C$  is not preserved if, e.g.

- $X = ABC$  and  $ABC$  are all in relation  $R$
- after decomposition into  $S$  and  $T$ ,  $AB$  is in  $S$  and  $BC$  is in  $T$

# BCNF Example.

```
z9300035 - cs3311@williams - /import/adams/t/cs3311/web/21T3/online/week10-tuesday -- ss
GradeBook(id, name, subj, term, title, mark, grade)
GradeBook(a, b, c, d, e, f, g)
```

FDs

$\text{id} \rightarrow \text{name}$   
 $\text{subj} \rightarrow \text{title}$

$\text{id}, \text{subj}, \text{term} \rightarrow \text{mark}, \text{grade}$

$\text{key} = \text{id}, \text{subj}, \text{term}$

} Determine the dependency.

→ key ⇒ Can determine all dependency.

BCNF

$R = \text{id}, \text{name}, \text{subj}, \text{term}, \text{title}, \text{mark}, \text{grade}$   
 $\text{key} = \text{id}, \text{subj}, \text{term}$

Original schema.

$R_0 = \text{id}, \text{subj}, \text{term}, \text{title}, \text{mark}, \text{grade}$   
 $\text{key}_0 = \text{id}, \text{subj}, \text{term}$

✓ decompose.

$R_1 = \text{id}, \text{name}$   
 $\text{key}_1 = \text{id}$   
now in bcnf

$R_2 = \text{id}, \text{subj}, \text{term}, \text{mark}, \text{grade}$   
 $\text{key}_2 = \text{id}, \text{subj}, \text{term}$   
now in bcnf

$R_3 = \text{subj}, \text{title}$   
 $\text{key}_3 = \text{subj}$   
now in bcnf

Schema is bcnf if Full key on left.

## ❖ Boyce-Codd Normal Form (cont)

Example: schema in BCNF

$R = ABCD, F = \{A \rightarrow B, A \rightarrow C, A \rightarrow D\}$

$\text{key}(R) = A$ , all fds have key on RHS

Example: schema *not* in BCNF

$R = ABCD, F = \{A \rightarrow BCD, D \rightarrow B, BC \rightarrow AD\}$

if  $\text{key}(R) = A$ ,  $D \rightarrow B$  does not have key on LHS

if  $\text{key}(R) = BC$ ,  $D \rightarrow B$  does not have key on LHS

## ❖ Third Normal Form

A relation schema  $R$  is in 3NF w.r.t a set  $F$  of functional dependencies iff:

for all  $fds X \rightarrow Y$  in  $F^+$

- either  $X \rightarrow Y$  is trivial (i.e.  $Y \subset X$ )
- or  $X$  is a superkey
- or  $Y$  is a single attribute from a key

A DB schema is in 3NF if all relation schemas are in 3NF.

The extra condition represents a slight weakening of BCNF requirements.

## ❖ Third Normal Form (cont)

If we transform a schema into 3NF, we are guaranteed:

- lossless join decomposition
- the new schema preserves all of the *fds* from the original schema

However, we are **not** guaranteed:

- no update anomalies due to *fd*-based redundancy

Whether to use BCNF or 3NF depends on overall design considerations.

## ❖ Third Normal Form (cont)

Example: schema in 3NF

$$R = ABCDE, F = \{B \rightarrow ACDE, E \rightarrow B\}$$

$\text{key}(R) = B$ , in  $E \rightarrow B$ ,  $E$  is not a key, but  $B$  is

Example: schema *not* in 3NF

$$R = ABCDE, F = \{B \rightarrow ACDE, E \rightarrow D\}$$

$\text{key}(R) = B$ , in  $E \rightarrow D$ ,  $E$  is not a key, neither is  $D$

---

Produced: 5 Nov 2020

# Normalisation

---

- Normalisation
- Relation Decomposition
- Schema (Re)Design
- BCNF Normalisation Algorithm
- BCNF Normalisation Example
- 3NF Normalisation Algorithm
- 3NF Normalisation Example
- Database Design Methodology

## ❖ Normalisation

Normalisation aims to put a schema into  $xNF$

- by ensuring that all relations in the schema are in  $xNF$

How normalisation works ...

- decide which normal form  $xNF$  is "acceptable"
  - i.e. how much redundancy are we willing to tolerate?
- check whether each relation in schema is in  $xNF$
- if a relation is not in  $xNF$ 
  - **partition** into sub-relations where each is "closer to"  $xNF$
- repeat until all relations in schema are in  $xNF$

## ❖ Normalisation (cont)

In practice, BCNF and 3NF are the most important normal forms.

Boyce-Codd Normal Form (BCNF):

- eliminates all redundancy due to functional dependencies
- but may not preserve original functional dependencies

Third Normal Form (3NF):

- eliminates most (but not all) redundancy due to *fds*
- guaranteed to preserve all functional dependencies

## ❖ Relation Decomposition

The standard transformation technique to remove redundancy:

- decompose relation  $R$  into relations  $S$  and  $T$

We accomplish decomposition by

- selecting (overlapping) subsets of attributes
- forming new relations based on attribute subsets

Properties:  $R = S \cup T$ ,  $S \cap T \neq \emptyset$  and  $r(R) = s(S) \bowtie t(T)$

May require several decompositions to achieve acceptable NF.

Normalisation algorithms tell us how to choose  $S$  and  $T$ .

## ❖ Schema (Re)Design

Consider the following relation for *BankLoans*:

<b>branchName</b>	<b>branchCity</b>	<b>assets</b>	<b>custName</b>	<b>loanNo</b>	<b>amount</b>
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-15	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
North Town	Rye	3700000	Hayes	L-16	1300

This schema has all of the update anomalies mentioned earlier.

## ❖ Schema (Re)Design (cont)

To improve the design, decompose the *BankLoans* relation.

The following decomposition is not helpful:

*Branch(branchName, branchCity, assets)*  
*CustLoan(custName, loanNo, amount)*

because we lose information (which branch is a loan held at?)

Another possible decomposition:

*BranchCust(branchName, branchCity, assets, custName)*  
*CustLoan(custName, loanNo, amount)*

## ❖ Schema (Re)Design (cont)

The *BranchCust* relation instance:

<b>branchName</b>	<b>branchCity</b>	<b>assets</b>	<b>custName</b>
Downtown	Brooklyn	9000000	Jones
Redwood	Palo Alto	2100000	Smith
Perryridge	Horseneck	1700000	Hayes
Downtown	Brooklyn	9000000	Jackson
Mianus	Horseneck	400000	Jones
Round Hill	Horseneck	8000000	Turner
North Town	Rye	3700000	Hayes

## ❖ Schema (Re)Design (cont)

The *CustLoan* relation instance:

<b>custName</b>	<b>loanNo</b>	<b>amount</b>
Jones	L-17	1000
Smith	L-23	2000
Hayes	L-15	1500
Jackson	L-15	1500
Jones	L-93	500
Turner	L-11	900
Hayes	L-16	1300

## ❖ Schema (Re)Design (cont)

Now consider the result of (*BranchCust*  $\bowtie$  *CustLoan*)

<b>branchName</b>	<b>branchCity</b>	<b>assets</b>	<b>custName</b>	<b>loanNo</b>	<b>amount</b>
Downtown	Brooklyn	9000000	Jones	L-17	1000
Downtown	Brooklyn	9000000	Jones	L-93	500
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Perryridge	Horseneck	1700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Jackson	L-15	1500
Mianus	Horseneck	400000	Jones	L-93	500
Mianus	Horseneck	400000	Jones	L-17	1000
Round Hill	Horseneck	8000000	Turner	L-11	900
North Town	Rye	3700000	Hayes	L-16	1300
North Town	Rye	3700000	Hayes	L-15	1500

## ❖ Schema (Re)Design (cont)

This is clearly not a successful decomposition.

The fact that we ended up with extra tuples was symptomatic of losing some critical "connection" information during the decomposition.

Such a decomposition is called a **lossy decomposition**.

In a good decomposition, we should be able to reconstruct the original relation exactly:

if  $R$  is decomposed into  $S$  and  $T$ , then  $S \bowtie T = R$

Such a decomposition is called **lossless join decomposition**.

## ❖ BCNF Normalisation Algorithm

The following algorithm converts an arbitrary schema to BCNF:

```
Inputs: schema  $R$ , set  $F$  of  $fds$ 
Output: set  $Res$  of BCNF schemas
```

```
Res = { $R$ };
while (any schema  $S \in Res$  is not in BCNF) {
    choose any fd  $X \rightarrow Y$  on  $S$  that violates BCNF
    Res = ( $Res - S$ )  $\cup$  ( $S - Y$ )  $\cup$   $XY$ 
}
```

The last step means: make a table from  $XY$ ; drop  $Y$  from table  $S$

The "choose any" step means that the algorithm is non-deterministic

## ❖ BCNF Normalisation Example

Recall the *BankLoans* schema:

*BankLoans(branchName, branchCity, assets, custName, loanNo, amount)*

Rename to simplify ...

$B = \text{branchName}$ ,  $C = \text{branchCity}$ ,  $A = \text{assets}$ ,  $N = \text{CustName}$ ,  $L = \text{loanNo}$ ,  $M = \text{amount}$

So ...  $R = BCANLM$ ,  $F = \{B \rightarrow CA, L \rightarrow MN\}$ ,  $\text{key}(R) = BL$

$R$  is not in BCNF, because  $B \rightarrow CA$  is not a whole key

Decompose into

- $S = BCA$ ,  $F_S = \{B \rightarrow CA\}$   $\text{key}(S) = B$
- $T = BNLM$ ,  $F_T = \{L \rightarrow NM\}$ ,  $\text{key}(T) = BL$

(continued)

## ❖ BCNF Normalisation Example (cont)

$S = BCA$  is in BCNF, only one  $fd$  and it has key on LHS

$T = BLNM$  is not in BCNF, because  $L \rightarrow NM$  is not a whole key

Decompose into ...

- $U = LNM, F_U = \{L \rightarrow NM\}, \text{key}(U) = L$ , which is BCNF
- $V = BL, F_V = \{\}, \text{key}(V) = BL$ , which is BCNF

Result:

- $S = (\text{branchName}, \text{branchCity}, \text{assets}) = \text{Branches}$
- $U = (\text{loanNo}, \text{custName}, \text{amount}) = \text{Loans}$
- $V = (\text{branchName}, \text{loanNo}) = \text{BranchOfLoan}$

## ❖ 3NF Normalisation Algorithm

The following algorithm converts an arbitrary schema to 3NF:

**Inputs:** schema  $R$ , set  $F$  of  $fds$   
**Output:** set  $R_i$  of 3NF schemas

```
let  $F_C$  be a reduced minimal cover for  $F$ 
 $Res = \{\}$ 
for each  $fd X \rightarrow Y$  in  $F_C$  {
    if (no schema  $S \in Res$  contains  $XY$ ) {
         $Res = Res \cup XY$ 
    }
}
if (no schema  $S \in Res$  contains a key for  $R$ ) {
     $K =$  any candidate key for  $R$ 
     $Res = Res \cup K$ 
}
```

## ❖ 3NF Normalisation Example

Recall the *BankLoans* schema:

*BankLoans*(*branchName*, *branchCity*, *assets*, *custName*, *loanNo*, *amount*)

Rename to simplify ...

$R = BCANLM, F = \{B \rightarrow CA, L \rightarrow MN\}, \text{key}(R) = BL$

Compute minimal cover =  $\{B \rightarrow C, B \rightarrow A, L \rightarrow M, L \rightarrow N\}$

Reduce minimal cover =  $\{B \rightarrow CA, L \rightarrow MN\}$

Convert into relations:  $S = BCA, T = LNM$

No relation has key  $BL$ , so add new table containing key  $U = BL$

Result is  $S = BCA, T = LNM, U = BL$  ... same as BCNF

## ❖ Database Design Methodology

To achieve a "good" database design:

- identify attributes, entities, relationships → ER design
- map ER design to relational schema
- identify constraints (including keys and functional dependencies)
- apply BCNF/3NF algorithms to produce normalised schema

Note: may subsequently need to "denormalise" if the design yields inadequate performance.

---

Produced: 5 Nov 2020

# Relational Algebra

---

- Relational Algebra
- Notation
- Describing RA Operations
- Example Database #1
- Example Database #2
- Rename
- Selection
- Projection

## ❖ Relational Algebra

Relational algebra (RA) can be viewed as ...

- mathematical system for manipulating relations, or
- data manipulation language (DML) for the relational model

Relational algebra consists of:

- **operands**: relations, or variables representing relations
- **operators** that map relations to relations
- rules for combining operands/operators into expressions
- rules for evaluating such expressions

Why is it important?

- because it forms the basis for DBMS implementation
- relational algebra ops are like the machine code for DBMSs

## ❖ Relational Algebra (cont)

Core relational algebra operations:

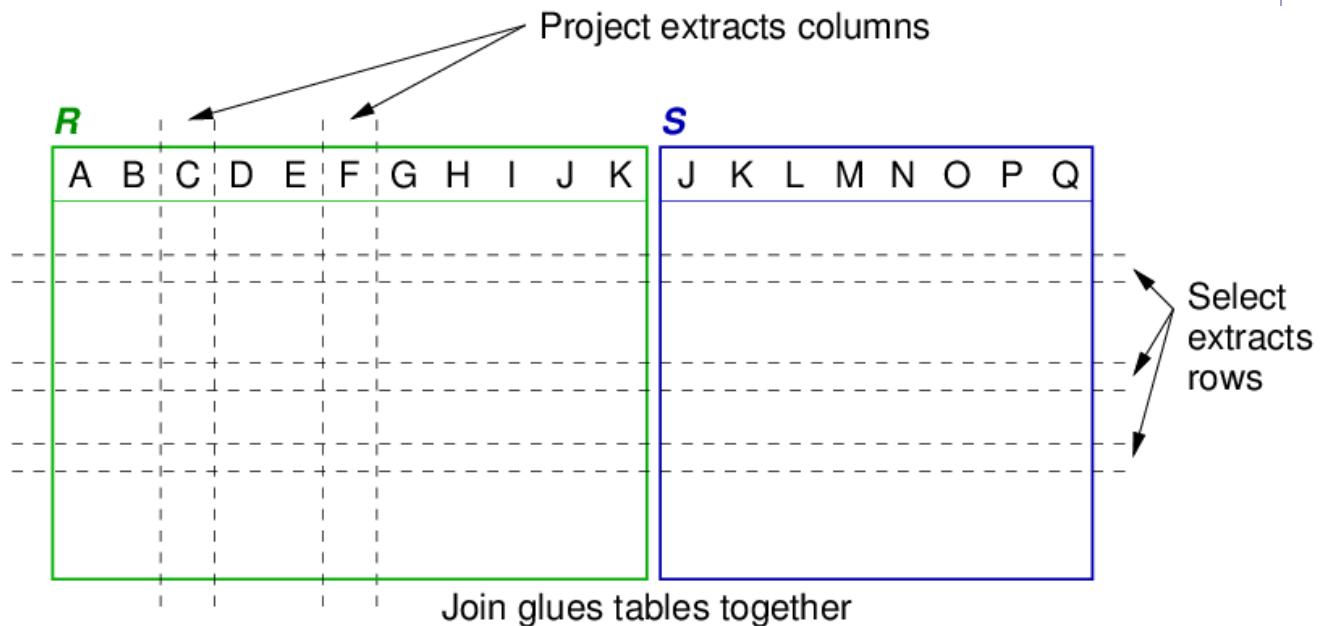
- **rename**: change names of relations/attributes
- **selection**: choosing a subset of tuples/rows
- **projection**: choosing a subset of attributes/columns
- **union, intersection, difference**: combining relations
- **product, join**: combining relations

Common extensions include:

- **aggregation, projection++, division**

## Relational Algebra (cont)

Select, project, join provide a powerful set of operations for building relations and extracting interesting data from them.



Adding set operations and renaming makes RA complete.

## ❖ Notation

---

Standard treatments of relational algebra use Greek symbols.

We use the following notation (because it is easier to reproduce):

<b>Operation</b>	<b>Standard Notation</b>	<b>Our Notation</b>
Selection	$\sigma_{expr}(Rel)$	$Sel[expr](Rel)$
Projection	$\pi_{A,B,C}(Rel)$	$Proj[A,B,C](Rel)$
Join	$Rel_1 \bowtie_{expr} Rel_2$	$Rel_1 \text{ Join}[expr] Rel_2$
Rename	$\rho_{schema}Rel$	$Rename[schema](Rel)$

For other operations (e.g. set operations) we adopt the standard notation.  
Except when typing in a text file, where \* = intersection, + = union

## ❖ Describing RA Operations

We define the semantics of RA operations using

- "conditional set" expressions e.g.  $\{X / \text{condition on } X\}$
- tuple notations:
  - $t[AB]$  (extracts attributes  $A$  and  $B$  from tuple  $t$ )
  - $(x,y,z)$  (enumerated tuples; specify attribute values)
- quantifiers, set operations, boolean operators

For each operation, we also describe it operationally:

- give an algorithm to compute the result, tuple-by-tuple
- the algorithm is not generally how it will be computed in practice

## ❖ Describing RA Operations (cont)

All RA operators return a result of type **relation**.

For convenience, we can name a result and use it later.

E.g.

```
Temp = R op1 S op2 T  
Res  = Temp op3 Z  
-- which is equivalent to  
Res  = (R op1 S op2 T) op3 Z
```

Each "intermediate result" has a well-defined schema.

## ❖ Example Database #1

R

A	B	C	D
a	1	x	4
b	2	y	5
c	4	z	4
d	8	x	5
e	1	y	4
f	2	x	5

S

D	E	F
1	a	x
2	b	y
3	c	x
4	a	y
5	b	x

## ❖ Example Database #2

**Beers(name,manf)**

(VB, Carlton)  
 (New, Tooheys)  
 (Porter, Maltshovel)  
 ...

**Beers(name,addr,licence)**

(CBH, Coogee,433122)  
 (Royal, Randwick, 632987)  
 (Regent, Kingsford,112112)  
 ...

**Frequents(drinker,bar)**

(John, CBH)  
 (Gernot, CBH)  
 (Gernot, Regent)  
 ...

**Likes(drinker, beer)**

(Andrew, New)  
 (Gernot, Porter)  
 (John, Pale Ale)  
 ...

**Beers(name,addr,phone)**

(John, Alexandria, 93111139)  
 (Gernot, Newtown, 92422429)  
 (Andrew, Glebe, 90411049)  
 ...

**Sells(beer,bar,price)**

(CBH, New, 2.50)  
 (CBH, VB, 1.99)  
 Royal, Porter, 3.00  
 ...

## ❖ Rename

Rename provides "schema mapping".

If expression  $E$  returns a relation  $R(A_1, A_2, \dots, A_n)$ , then

$\text{Rename}[S(B_1, B_2, \dots, B_n)](E)$

gives a relation called  $S$

- containing the same set of tuples as  $E$
- but with the name of each attribute changed from  $A_i$  to  $B_i$

Rename is like the identity function on the *contents* of a relation

The only thing that Rename changes is the schema.

## ❖ Rename (cont)

Rename can be viewed as a "technical" apparatus of RA.

We can also use implicit rename/project in sequences of RA operations, e.g.

```
-- R(a,b,c), S(c,d)
Res = Rename[Res(b,c,d)](Project[b,c](Sel[a>5](R)) Join S)
-- vs
Tmp1 = Select[a>5](R)
Tmp2 = Project[b,c](Tmp1)
Tmp3 = Rename[Tmp3(cc,d)](S)
Tmp4 = Tmp2 Join[c=cc] Tmp3
Res = Rename[Res(b,c,d)](Tmp4)
-- vs
Tmp1(b,c) = Select[a>5](R)
Tmp2(cc,d) = S
Res(b,c,d) = Tmp1 Join[c=cc] Tmp2
```

In SQL, we achieve a similar effect by defining a set of views

## ❖ Selection

---

**Selection** returns a subset of the tuples in a relation  $r(R)$  that satisfy a specified condition  $C$ .

$$\sigma_C(r) = Sel[C](r) = \{ t \mid t \in r \wedge C(t) \}$$

$C$  is a boolean expression on attributes in  $R$ .

Result size:  $|\sigma_C(r)| \leq |r|$

Result schema: same as the schema of  $r$  (i.e.  $R$ )

Algorithmic view:

```
result = {}
for each tuple t in relation r
    if (C(t)) { result = result U {t} }
```

## ❖ Selection (cont)

Examples of selection:

R

A	B	C	D
a	1	x	4
b	2	y	5
c	4	z	4
d	8	x	5
e	1	y	4
f	2	x	5

**Sel[B=1](R)**

A	B	C	D
a	1	x	4
e	1	y	4

**Sel[B>=D](R)**

A	B	C	D
c	4	z	4
d	8	x	5

**Sel[A=C](R)**

A	B	C	D

**Sel[A=b or A=c](R)**

A	B	C	D
b	2	y	5
c	4	z	4

## ❖ Selection (cont)

Querying with relational algebra (selection) ...

- Details of all bars in The Rocks

```
Result = Sel[addr=The Rocks](Bars)
```

- Beers made by Sierra Nevada

```
SNBeers = Sel[manf=Sierra Nevada](Beers)
Result = Rename[beer](Proj[name](SNBeers))
```

## ❖ Projection

**Projection** returns a set of tuples containing a subset of the attributes in the original relation.

$$\Pi_X(r) = \text{Proj}[X](r) = \{ t[X] \mid t \in r \}, \text{ where } r(R)$$

$X$  specifies a subset of the attributes of  $R$ .

Note that removing key attributes can produce duplicates.

In RA, duplicates are removed from the result **set**.

(In RDBMS's, duplicates are retained (i.e. they use bags, not sets))

Result size:  $|\Pi_X(r)| \leq |r|$  Result schema:  $R'(X)$

Algorithmic view:

```
result = {}
for each tuple t in relation r
    result = result ∪ {t[X]}
```

## ❖ Projection (cont)

Examples of projection:

R

A	B	C	D
a	1	x	4
b	2	y	5
c	4	z	4
d	8	x	5
e	1	y	4
f	2	x	5

Proj[A,B,C](R)

A	B	C
a	1	x
b	2	y
c	4	z
d	8	x
e	1	y
f	2	x

Proj[B,D](R)

B	D
1	4
2	5
4	4
8	5

Proj[D](R)

D
4
5

## ❖ Projection (cont)

Querying with relational algebra (projection)...

- Names of all beers

```
Result = Proj[name](Beers)
```

- Names of drinkers who live in Newtown

```
Result = Proj[name](Sel[addr=Newtown](Drinkers))
```

- What are all of the breweries?

```
Result(brewer) = Proj[manf](Beers)
```



# RA Set Operations

---

- RA Set Operations
- Union
- Intersection
- Difference

## ❖ RA Set Operations

Relational algebra defines three set operations

- union ...  $R \cup S$  ... (Query<sub>1</sub>) **UNION** (Query<sub>2</sub>)
- intersection ...  $R \cap S$  ... (Query<sub>1</sub>) **INTERSECT** (Query<sub>2</sub>)
- difference ...  $R - S$  ... (Query<sub>1</sub>) **EXCEPT** (Query<sub>2</sub>)

All relations involved must have the same schema (union-compatible)

All operations give a *set* of results (i.e. no duplicates)

To get *bag* semantics, use **UNION ALL**, etc.

## ❖ Union

Union combines two compatible relations into a single relation via set union of sets of tuples.

$$r_1 \cup r_2 = \{ t \mid t \in r_1 \vee t \in r_2 \}, \text{ where } r_1(R), r_2(R)$$

Result size:  $|r_1 \cup r_2| \leq |r_1| + |r_2|$  Result schema:  $R$

Algorithmic view:

```
result = r1
for each tuple t in relation r2
    result = result ∪ {t}
```

## ❖ Intersection

Intersection combines two compatible relations into a single relation via set intersection of sets of tuples.

$$r_1 \cap r_2 = \{ t \mid t \in r_1 \wedge t \in r_2 \}, \text{ where } r_1(R), r_2(R)$$

Result size:  $|r_1 \cap r_2| \leq \min(|r_1|, |r_2|)$  Result schema:  $R$

Algorithmic view:

```
result = {}
for each tuple t in relation r1
    if (t ∈ r2) { result = result ∪ {t} }
```

## ❖ Intersection (cont)

Examples of union and intersection:

$T = \text{Sel}[B=1](R)$

A	B	C	D
a	1	x	4
e	1	y	4

$U = \text{Sel}[C=x](R)$

A	B	C	D
a	1	x	4
d	8	x	5

$T \text{ union } U$

A	B	C	D
a	1	x	4
d	8	x	5
e	1	y	4

$T \text{ intersect } U$

A	B	C	D
a	1	x	4

## ❖ Intersection (cont)

Querying with relational algebra (set operations)...

- Bars where either John or Gernot drinks

```
JohnBars = Proj[bar](Sel[drinker=John](Frequents))
GernotBars = Proj[bar](Sel[drinker=Gernot](Frequents))

Result = JohnBars union GernotBars
```

- Bars where both John and Gernot drink

```
Result = JohnBars intersect GernotBars
```

## ❖ Difference

---

**Difference** finds the set of tuples that exist in one relation but do not occur in a second **compatible** relation.

$$r_1 - r_2 = \{ t \mid t \in r_1 \wedge t \notin r_2 \}, \text{ where } r_1(R), r_2(R)$$

Uses same notion of relation compatibility as union.

Note: tuples in  $r_2$  but not  $r_1$  do not appear in the result

- i.e. set difference != complement of set intersection

Algorithmic view:

```
result = {}
for each tuple t in relation r1
    if (!(t ∈ r2)) { result = result ∪ {t} }
```

## ❖ Difference (cont)

Examples of difference:

$$T = \text{Sel}[B=1](R)$$

A	B	C	D
a	1	x	4
e	1	y	4

$$U = \text{Sel}[C=x](R)$$

A	B	C	D
a	1	x	4
d	8	x	5

$$T - U$$

A	B	C	D
e	1	y	4

$$U - T$$

A	B	C	D
d	8	x	5

Clearly, difference is not symmetric.

## ❖ Difference (cont)

Querying with relational algebra (difference) ...

- Bars where John drinks and Gernot doesn't

```
JohnBars = Proj[bar](Sel[drinker=John](Frequents))
GernotBars = Proj[bar](Sel[drinker=Gernot](Frequents))

Result = JohnBars - GernotBars
```

- Bars that sell VB but not New

```
VBBars = Proj[bar](Sel[beer=VB](Sells))
NewBars = Proj[bar](Sel[beer>New](Sells))

Result = VBBars - NewBars
```



# RA Join Operations

---

- RA Operations to Combine Relations
- Product
- Natural Join
- Theta Join
- Outer Join
- Division

## ❖ RA Operations to Combine Relations

Relational algebra has several ways to combine info from relations

- product ...  $R \times S$  ... select \* from R join S on (true)
- natural join ...  $R \bowtie S$  ... select \* from R natural join S
- (inner) join ...  $R \bowtie_C S$  ... select \* from R join S on (C)
- outer join ...  $R \bowtie_C S$  ... select \* from R left outer join S on (C)
- division ...  $R / S$  ... see SQL slides for examples

Join conditions involve related attributes from the two relations

Frequently, primary-key joined with foreign-key

## ❖ Product

**Product** (Cartesian product) combines information from two relations pairwise on tuples.

$$r \times s = \{ (t_1:t_2) \mid t_1 \in r \wedge t_2 \in s \}, \text{ where } r(R), s(S)$$

Each tuple in the result contains all attributes from  $r$  and  $s$ , possibly with some fields renamed to avoid ambiguity.

If  $t_1 = (A_1 \dots A_n)$  and  $t_2 = (B_1 \dots B_n)$  then  $(t_1:t_2) = (A_1 \dots A_n, B_1 \dots B_n)$

Result size is **large**:  $|r \times s| = |r|.|s|$  Schema:  $R \cup S$

Algorithmic view:

```
result = {}
for each tuple  $t_1$  in relation  $r$ 
    for each tuple  $t_2$  in relation  $s$ 
        result = result  $\cup$   $\{(t_1:t_2)\}$ 
```

## ❖ Product (cont)

Example of product:

R

A	B	C
a	m	x
e	k	y
f	p	z

S

X	Y
1	1
2	2

R  $\times$  S

A	B	C	X	Y
a	m	x	1	1
a	m	x	2	2
e	k	y	1	1
e	k	y	2	2
f	p	z	1	1
f	p	z	2	2

## ❖ Natural Join

Natural join is a specialised product:

- containing only pairs that match on **common** attributes
- with one of each pair of common attributes eliminated

Consider relation schemas  $R(ABC..J\textcolor{green}{KLM})$ ,  
 $S(\textcolor{green}{KLMN}..XYZ)$ .

The natural join of relations  $r(R)$  and  $s(S)$  is defined as:

$$r \bowtie s = r \text{Join } s = \{ (t_1[ABC..J]: t_2[K..XYZ]) \mid t_1 \in r \wedge t_2 \in s \wedge \text{match} \}$$

where  $\text{match} = t_1[K] = t_2[K] \wedge t_1[L] = t_2[L] \wedge t_1[M] = t_2[M]$

Algorithmic view:

```

result = {}
for each tuple  $t_1$  in relation  $r$ 
    for each tuple  $t_2$  in relation  $s$ 
        if ( $\text{matches}(t_1, t_2)$ )
            result = result U { $\text{combine}(t_1, t_2)$ }

```

## ❖ Natural Join (cont)

Example of natural join:

R

A	B	C
a	m	x
e	k	y
f	p	z

R Join S

A	B	C	D
a	m	x	1
f	p	z	2

S

C	D
x	1
z	2

## ❖ Theta Join

---

The **theta join** is a specialised product containing only pairs that match on a supplied condition  $C$ .

$$r \bowtie_C s = \{ (t_1 : t_2) \mid t_1 \in r \wedge t_2 \in s \wedge C(t_1 : t_2) \},$$

where  $r(R), s(S)$

Examples:  $(r1 \text{Join}[B>E] r2) \dots (r1 \text{Join}[E<D \wedge C=G] r2)$

All attribute names are required to be distinct (cf natural join)

Can be defined in terms of other RA operations:

$$r \bowtie_C s = r \text{Join}[C] s = \text{Sel}[C](r \times s)$$

Note that  $r \bowtie_{\text{true}} s = r \times s$ .

## ❖ Theta Join (cont)

Example theta join:

R

A	B	C
a	m	x
e	k	y
f	p	z

S

D	E
x	1
x	2
y	3

R **Join[C>D]** S

A	B	C	D	E
e	k	y	x	1
e	k	y	x	2
f	p	z	x	1
f	p	z	x	2
f	p	z	y	3

(Theta join is the most frequently-used join in SQL queries)

## ❖ Theta Join (cont)

Querying with relational algebra (join) ...

- Who drinks in Newtown bars?

```
NewtownBars(nbar) = Sel[addr=Newtown](Bars)
Tmp = Frequent Join[bar=nbar] NewtownBars
Result(drinker) = Proj[drinker](Tmp)
```

- Who drinks beers made by Carlton?

```
CarltonBeers = Sel[manf=Carlton](Beers)
Tmp = Likes Join[beer=name] CarltonBeers
Result(drinker) = Proj[drinker]Tmp
```

Reminder: projection removes duplicates

## ❖ Outer Join

$R \bowtie_C S$  does not include in its result

- values from any R tuples that do not match some S tuple under C
- values from any S tuples that do not match some R tuple under C

$R \bowtie_C S$  (left outer join) includes

- all tuples that would result from a theta join
- values from all R tuples, even with no matching S tuple

For tuples with no match, assign NULL to "unmatched" attributes

Variants are [right outer join](#) and [full outer join](#)

## ❖ Outer Join (cont)

Example left outer join:

**R**

A	B	C
a	m	x
e	k	y
f	p	z

**S**

D	E
x	1
x	2
y	3

**R Left Outer Join[C=D] S**

A	B	C	D	E
a	m	x	x	1
a	m	x	x	2
e	k	y	y	3
f	p	z	NULL	NULL

## ❖ Division

---

Consider two relation schemas  $R$  and  $S$  where  $S \subset R$ .

The **division** operation is defined on instances  $r(R)$ ,  $s(S)$  as:

$$r/s = r \text{Div } s = \{ t \mid t \in r[R-S] \wedge \text{satisfy} \}$$

where  $\text{satisfy} = \forall t_s \in S (\exists t_r \in R (t_r[S] = t_s \wedge t_r[R-S] = t))$

Operationally:

- consider each subset of tuples in  $R$  that match on  $t[R-S]$
- for this subset of tuples, take the  $t[S]$  values from each
- if this covers all tuples in  $S$ , then include  $t[R-S]$  in the result

## ❖ Division (cont)

Example of division:

R

A	B
4	x
4	y
4	z
5	x
5	y
5	z

S

A	B
4	x
4	y
4	z
5	x
5	y
5	z

T

B
x
y

R / T

A
4
5

S / T

A
4

## ❖ Division (cont)

Querying with relational algebra (division) ...

Division handles queries that include the notion "for all".

E.g. Which beers are sold in all bars?

We can answer this as follows:

- generate a relation of beers and bars where they are sold
  - $r1 = \text{Proj}[\text{beer}, \text{bar}](\text{Sold})$
- generate a relation of all bars
  - $r2 = \text{Rename}[r2(\text{bar})](\text{Proj}[\text{name}](\text{Bars}))$
- find which beers appear in tuples with **every** bar
  - $\text{res} = r1 \text{ Div } r2$

---

Produced: 11 Nov 2020

# Query Evaluation

---

- DBMS Architecture
- Query Evaluation
- Mapping SQL to RA
- Mapping Example
- Query Cost Estimation
- Implementations of RA Ops
- Query Optimisation

## ❖ DBMS Architecture

COMP3311 is not a course on DBMS Architecture (that's COMP9315)

But knowing just a little about how DBMSs work can help

- to avoid/fix inefficiencies in database applications

DBMSs attempt to handle this issue in modules for ...

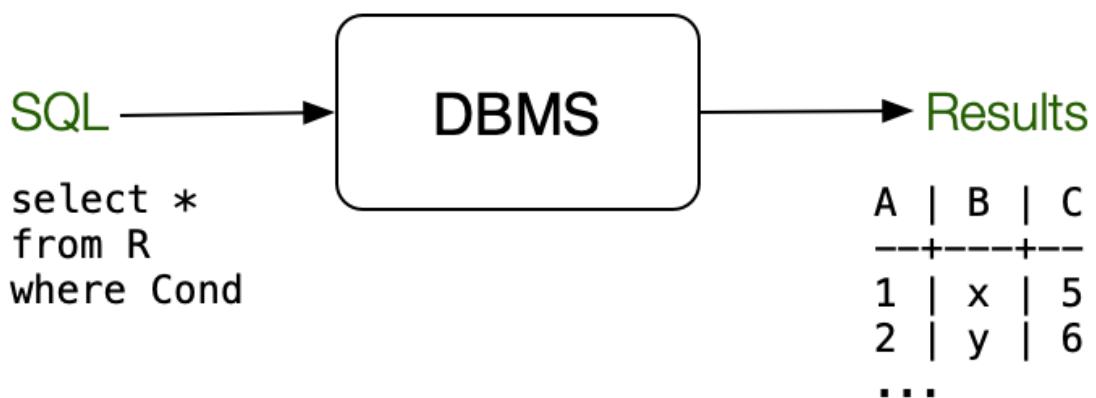
- query processing (QP) .. methods for evaluating queries

As a programmer, you cede a lot of control to the DBMS, but can

- use QP knowledge to make DB applications **efficient**

## ❖ DBMS Architecture (cont)

Our view of the DBMS so far ...



A machine to process SQL queries.

## ❖ DBMS Architecture (cont)

One view of DB engine: "relational algebra virtual machine"

Machine code for such a machine:

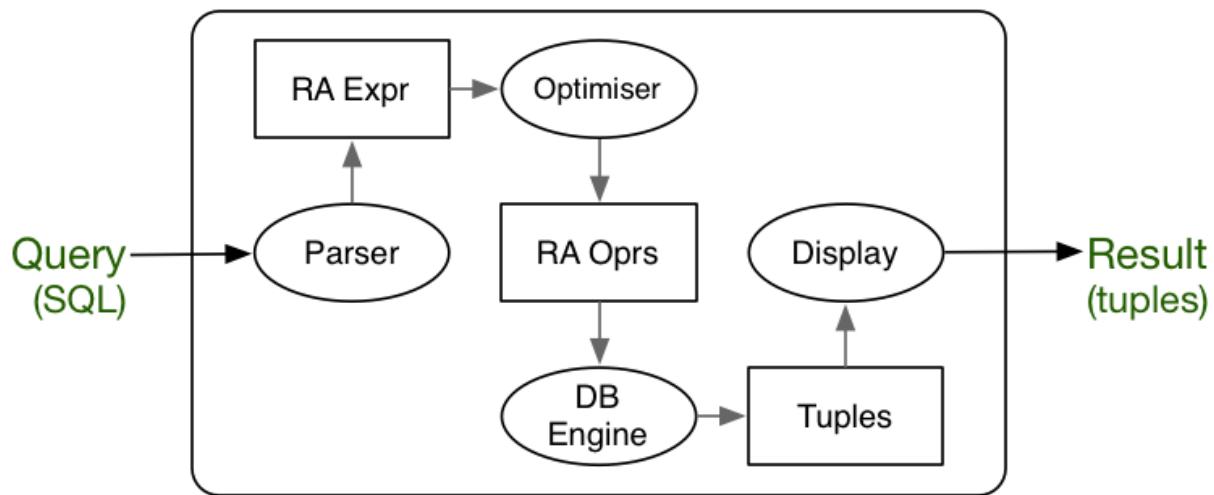
selection ( $\sigma$ )	projection ( $\pi$ )	join ( $\bowtie, \times$ )
union ( $\cup$ )	intersection ( $\cap$ )	difference (-)
sort	insert	delete

For each of these operations:

- various data structures and algorithms are available
- DBMSs may provide only one, or may provide a choice

## ❖ Query Evaluation

The path of a query through its evaluation:



## ❖ Mapping SQL to RA

Mapping SQL to relational algebra, e.g.

```
-- schema: R(a,b) S(c,d)
select a as x
from   R join S on (b=c)
where   d = 100
-- could be mapped to
Tmp1(a,b,c,d) = R Join[b=c] S
Tmp2(a,b,c,d) = Sel[d=100](Tmp1)
Tmp3(a)         = Proj[a](Tmp2)
Res(x)          = Rename[Res(x)](Tmp3)
```

In general:

- **SELECT** clause becomes *projection*
- **WHERE** condition becomes *selection* or *join*
- **FROM** clause becomes *join*

## ❖ Mapping Example

Consider the database schema:

```
Person(pid, name, address, ...)
Subject(sid, code, title, uoc, ...)
Terms(tid, code, start, end, ...)
Courses(cid, sid, tid, ...)
Enrolments(cid, pid, mark, ...)
```

and the query: *Courses with more than 100 students in them?*

which can be expressed in SQL as

```
select s.sid, s.code
  from Course c join Subject s on (c.sid=s.sid)
                  join Enrolment e on (c.cid=e.cid)
 group by s.sid, s.code
 having count(*) > 100;
```

## ❖ Mapping Example (cont)

The SQL might be compiled to

```

Tmp1(cid,sid,pid) = Course Join[c.cid = e.cid] Enrolment
Tmp2(cid,code,pid) = Tmp1 Join[t1.sid = s.sid] Subject
Tmp3(cid,code,nstu) = GroupCount[cid,code](Tmp2)
Res(cid,code)       = Sel[nstu > 100](Tmp3)
  
```

or, equivalently ✓

*create smaller set*

```

Tmp1(cid,code)      = Course Join[c.sid = s.sid] Subject
Tmp2(cid,code,pid)  = Tmp1 Join[t1.cid = e.cid] Enrolment
Tmp3(cid,code,nstu) = GroupCount[cid,code](Tmp2)
Res(cid,code)        = Sel[nstu > 100](Tmp3)
  
```

Which is better?

## ❖ Query Cost Estimation

The cost of evaluating a query is determined by

- the operations specified in the query execution plan
- size of relations (database relations and temporary relations)
- access mechanisms (indexing, hashing, sorting, join algorithms)
- size/number of main memory buffers (and replacement strategy)

Analysis of costs involves *estimating*:

- the size of intermediate results
- then, based on this, cost of secondary storage accesses

Accessing data from disk is the dominant cost in query evaluation

## ❖ Query Cost Estimation (cont)

An **execution plan** is a sequence of relational operations.

Consider execution plans for:  $\sigma_c(R \bowtie_d S \bowtie_e T)$

```
tmp1    := hash_join[d](R,S)
tmp2    := sort_merge_join[e](tmp1,T)
result  := binary_search[c](tmp2)
```

or

```
tmp1    := sort_merge_join[e](S,T)
tmp2    := hash_join[d](R,tmp1)
result  := linear_search[c](tmp2)
```

or

```
tmp1    := btree_search[c](R)
tmp2    := hash_join[d](tmp1,S)
result  := sort_merge_join[e](tmp2)
```

All produce same result, but have different costs.

## ❖ Implementations of RA Ops

**Sorting** (quicksort, etc. are not applicable)

- **external merge sort** (cost  $O(N \log_B N)$  with  $B$  memory buffers)

**Selection** (different techniques developed for different query types)

- **sequential scan** (worst case, cost  $O(N)$ )
- **index-based** (e.g. B-trees, cost  $O(\log N)$ , tree nodes are pages)
- **hash-based** ( $O(1)$  best case, only works for equality tests)

**Join** (fast joins are critical to success of relational DBMSs)

- **nested-loop join** (cost  $O(N \cdot M)$ , buffering can reduce to  $O(N+M)$ )
- **sort-merge join** (cost  $O(N \log N + M \log M)$ )
- **hash-join** (best case cost  $O(N+M \cdot N/B)$ , with  $B$  memory buffers)

## ❖ Query Optimisation

What is the "best" method for evaluating a query?

Generally, *best* = lowest cost = fastest evaluation time

**Cost** is measured in terms of pages read/written

- data is stored in fixed-size blocks (e.g. 4KB)
- data transferred disk  > memory in whole blocks
- cost of disk  memory transfer is highest cost in system
- processing in memory is very fast compared to I/O

## ❖ Query Optimisation (cont)

A DBMS query optimiser works as follows:

```
Input: relational algebra expression
Output: execution plan (sequence of RA ops)

bestCost = INF; bestPlan = none
while (more possible plans) {
    plan = produce a new evaluation plan
    cost = estimated_cost(plan)
    if (cost < bestCost) {
        bestCost = cost; bestPlan = plan
    }
}
return bestPlan
```

Typically, there are very many possible plans

- smarter optimisers generate likely subset of possible plans



# Performance Tuning

---

- DB Application Performance
- Indexes
- Query Tuning
- PostgreSQL Performance Analysis
- EXPLAIN Examples

## ❖ DB Application Performance

In order to make DB applications efficient, it is useful to know:

- what operations on the data does the application require  
(which queries, updates, inserts and how frequently is each one performed)
- how much each implementation will cost  
(in terms of the amount of data transferred between memory and disk ⇒ time)

and then, "encourage" the DBMS to use the most efficient methods

Achieve by using indexes and avoiding certain SQL query structures

## ❖ DB Application Performance (cont)

Application programmer choices that affect query cost:

- how queries are expressed
  - generally join is faster than subquery
  - especially if subquery is correlated
  - filter first, then join (avoids large intermediate tables)
  - avoid applying functions in where/group-by clauses
- creating **indexes** on tables
  - index will speed-up filtering based on indexed attributes
  - indexes generally only effective for equality, gt/lt
  - mainly useful if filtering much more frequent than update

## ❖ DB Application Performance (cont)

Whatever you do as a DB application programmer

- the DBMS **query optimiser** will transform your query
- attempt to make it execute as efficiently as possible

You have no control over the optimisation process

- but choices you make can block certain options
- limiting the query optimiser's chance to improve

## ❖ DB Application Performance (cont)

Example: query to find sales people earning more than \$50K

```
select name from Employee
where salary > 50000 and
      empid in (select empid from WorksIn
                  where dept = 'Sales')
```

A query evaluator might use the strategy

```
SalesEmps = (select empid from WorksIn where dept='Sales')
foreach e in Employee {
    if (e.empid in SalesEmps && e.salary > 50000)
        add e to result set
}
```

Needs to examine *all* employees, even if not in Sales

This is not a good expression of the query.

## ❖ DB Application Performance (cont)

A different expression of the same query:

```
select name
from Employee join WorksIn using (empid)
where Employee.salary > 5000 and
WorksIn.dept = 'Sales'
```

Query evaluator might use the strategy

```
SalesEmps = (select * from WorksIn where dept='Sales' )
foreach e in (Employee join SalesEmps) {
    if (e.salary > 50000)
        add e to result set
}
```

Only examines Sales employees, and uses a simpler test

This is a good expression of the query.

## ❖ DB Application Performance (cont)

A very poor expression of the query (correlated subquery):

```
select name from Employee e
where salary > 50000 and
      'Sales' in (select dept from WorksIn where empid=e.id)
```

A query evaluator would be forced to use the strategy:

```
foreach e in Employee {
    Depts = (select dept from WorksIn where empid=e.empid)
    if ('Sales' in Depts && e.salary > 50000)
        add e to result set
}
```

Needs to run a query for *every* employee ...

## ❖ Indexes

Indexes provide efficient content-based access to tuples.

Can build indexes on any (combination of) attributes.

Definining indexes:

```
CREATE INDEX name ON table ( attr1, attr2, ... )
```

*attr<sub>i</sub>* can be an arbitrary expression (e.g. **upper(name)**).

**CREATE INDEX** also allows us to specify

- that the index is on **UNIQUE** values
- an access method (**USING** btree, hash, ...)

## ❖ Indexes (cont)

Indexes can significantly improve query costs.

Considerations in applying indexes:

- is an attribute used in frequent/expensive queries?  
(note that some kinds of queries can be answered from index alone)
- should we create an index on a collection of attributes?  
(yes, if the collection is used in a frequent/expensive query)
- is the table containing attribute frequently updated?
- should we use B-tree or Hash index?

```
-- use hashing for (unique) attributes in equality tests, e.g.  
select * from Employee where id = 12345  
-- use B-tree for attributes in range tests, e.g.  
select * from Employee where age > 60
```

## ❖ Query Tuning

Sometimes, a query can be re-phrased to affect performance:

- by helping the optimiser to make use of indexes
- by avoiding unnecessary/expensive operations

Examples which *may* prevent optimiser from using indexes:

```
select name from Employee where salary/365 > 100
    -- fix by re-phrasing condition to (salary > 36500)
select name from Employee where name like '%ith%'
select name from Employee where birthday is null
    -- above two are difficult to "fix"
select name from Employee
where dept in (select id from Dept where ...)
    -- fix by using Employee join Dept on (e.dept=d.id)
```

## ❖ Query Tuning (cont)

Other tricks in query tuning (effectiveness is DBMS-dependent)

- **select distinct** typically requires a sort ...  
is the **distinct** really necessary? (at this stage in the query?)
- if multiple join conditions are available ...  
choose join attributes that are indexed, avoid joins on strings

```
select ... Employee join Customer on (s.name = p.name)
vs
select ... Employee join Customer on (s.ssn = p.ssn) ✓
```

- sometimes **or** prevents index from being used ...  
replace the **or** condition by a union of non-**or** clauses

```
select name from Employee where Dept=1 or Dept=2
vs
(select name from Employee where Dept=1)
union
(select name from Employee where Dept=2) ✓
```

## ❖ PostgreSQL Performance Analysis

PostgreSQL provides the **explain** statement to

- give a representation of the query execution plan
- with information that may help to tune query performance

Usage:

```
EXPLAIN [ANALYZE] Query
```

Without **ANALYZE**, **EXPLAIN** shows plan with estimated costs.

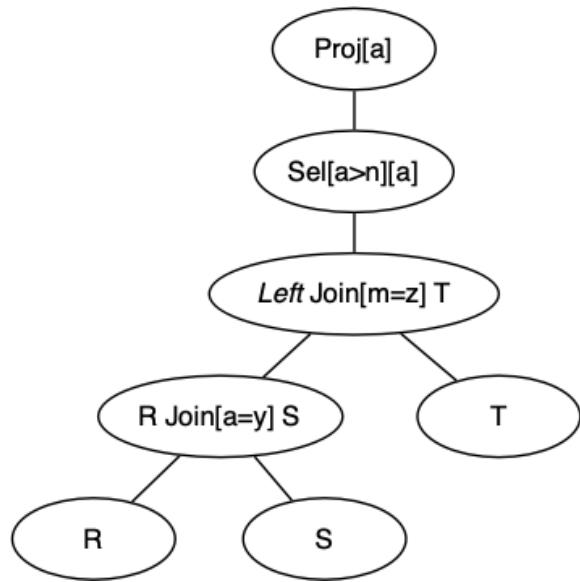
With **ANALYZE**, **EXPLAIN** executes query and prints real costs.

Note that runtimes may show considerable variation due to buffering.

If simply knowing the runtime is ok, maybe **\timing** is good enough

## ❖ EXPLAIN Examples

Note that PostgreSQL builds a query evaluation tree, rather than a linear plan, e.g.



```

select a
from   R
       join S on (R.a = S.y)
       join T on (T.m = S.z)
where  R.a > T.n
  
```

$\text{Tmp1} = \text{R Join}[a=y] S$   
 $\text{Tmp2} = \text{Tmp1 Join}[m=z] T$   
 $\text{Tmp3} = \text{Sel}[a > n](\text{Tmp2})$   
 $\text{Res} = \text{Proj}[a](\text{Tmp3})$

**EXPLAIN** effectively shows a pre-order traversal of the plan tree

## ❖ EXPLAIN Examples (cont)

Example: Select on indexed attribute

```
db=# explain analyze select * from Students where id=100250;
                                QUERY PLAN
-----
Index Scan using student_pkey on student
  (cost=0.00..5.94 rows=1 width=17)
    (actual time=3.209..3.212 rows=1 loops=1)
  Index Cond: (id = 100250)
Total runtime: 3.252 ms
```

Example: Select on non-indexed attribute

```
db=# explain analyze select * from Students where stype='local';
                                QUERY PLAN
-----
Seq Scan on student  (cost=0.00..70.33 rows=18 width=17)
  (actual time=0.061..7.784 rows=2512 loops=1)
  Filter: ((stype)::text = 'local'::text)
Total runtime: 7.554 ms
```

## ❖ EXPLAIN Examples (cont)

Example: Join on a primary key (indexed) attribute

```
db=# explain
db-# select s.sid,p.name
db-# from Students s join People p on s.id=p.id;

                                QUERY PLAN
-----
Hash Join  (cost=70.33..305.86 rows=3626 width=52)
  Hash Cond: ("outer".id = "inner".id)
    -> Seq Scan on people p
        (cost=0.00..153.01 rows=3701 width=52)
    -> Hash  (cost=61.26..61.26 rows=3626 width=8)
      -> Seq Scan on student s
          (cost=0.00..61.26 rows=3626 width=8)
```



# Transactions

---

- Transactions, Concurrency, Recovery
- Transactions
- Example Transaction
- Transaction Concepts
- Transaction Consistency

## ❖ Transactions, Concurrency, Recovery

DBMSs maintain valuable information in an environment that is:

- **shared** - concurrent access by multiple users
- **unstable** - potential for hardware/software failure

Each user should see the system as:

- unshared - their work is not inadvertently affected by others
- stable - the data survives in the face of system failures

Ultimate goal: data integrity is maintained at all times.

## ❖ Transactions, Concurrency, Recovery (cont)

### Transaction processing

- techniques for managing "logical units of work" which may require multiple DB operations

### Concurrency control

- techniques for ensuring that multiple concurrent transactions do not interfere with each other

### Recovery mechanisms

- techniques to restore information to a consistent state, even after major hardware shutdowns/failures

COMP3311 only looks at the first of these

## ❖ Transactions

A **transaction** is

- an atomic "unit of work" in an application
- which may require multiple database changes

Transactions happen in a multi-user, unreliable environment.

To maintain integrity of data, transactions must be:

- **Atomic** - either fully completed or completely rolled-back
- **Consistent** - map DB between consistent states
- **Isolated** - transactions do not interfere with each other
- **Durable** - persistent, restorable after system failures

## ❖ Example Transaction

Bank funds transfer

- move  $N$  dollars from account  $X$  to account  $Y$
- **Accounts (id, name, balance, heldAt, ...)**
- **Branches (id, name, address, assets, ...)**
- maintain **Branches.assets** as sum of balances via triggers
- transfer operation is implemented by a function which
  - has three parameters: amount, source acct, dest acct
  - checks validity of supplied accounts
  - checks sufficient available funds
  - returns a unique transaction ID on success

## ❖ Example Transaction (cont)

## Example function to implement bank transfer ...

```
create or replace function
    transfer(N integer, Src text, Dest text)
    returns integer
declare
    SID integer; dID integer; avail integer;
begin
    select id,balance into SID,avail
    from Accounts where name=Src;
    if (SID is null) then
        raise exception 'Invalid source account %',Src;
    end if;
    select id into dID
    from Accounts where name=Dest;
    if (dID is null) then
        raise exception 'Invalid dest account %',Dest;
    end if;
...

```

## ❖ Example Transaction (cont)

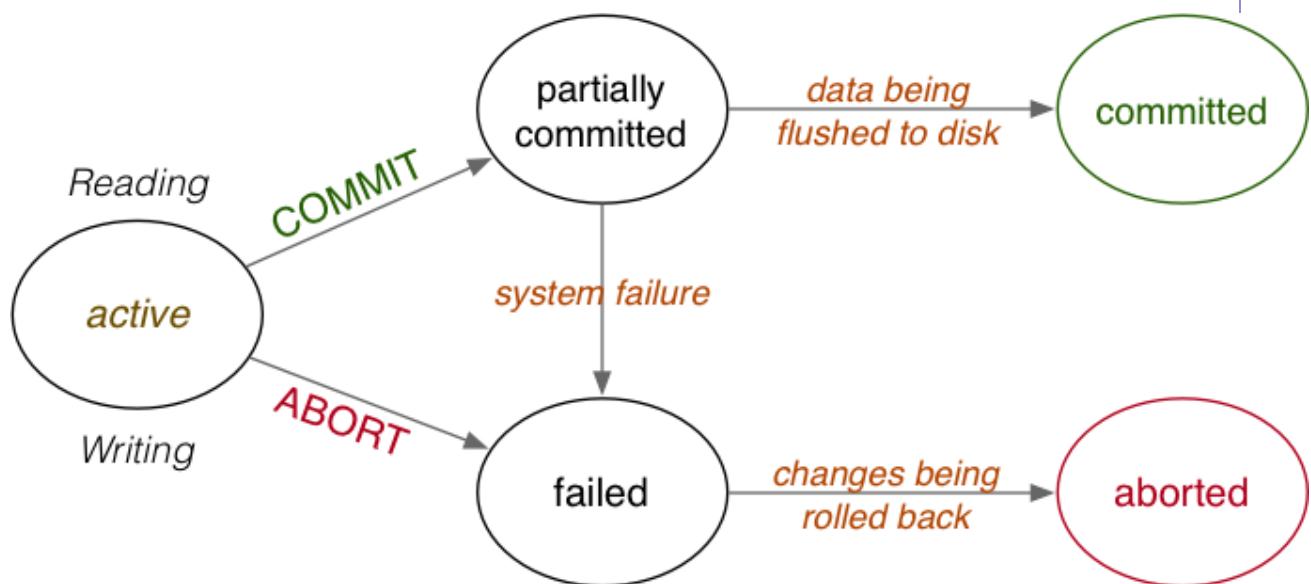
Example function to implement bank transfer (cont)...

```
...
if (avail < N) then
    raise exception 'Insufficient funds in %',Src;
end if;
-- total funds in system = NNNN
update Accounts set balance = balance-N
where id = sID;
-- funds temporarily "lost" from system
update Accounts set balance = balance+N
where id = dID;
-- funds restored to system; total funds = NNNN
return nextval('tx_id_seq');
end;
```

## ❖ Transaction Concepts

A transaction must always terminate, either:

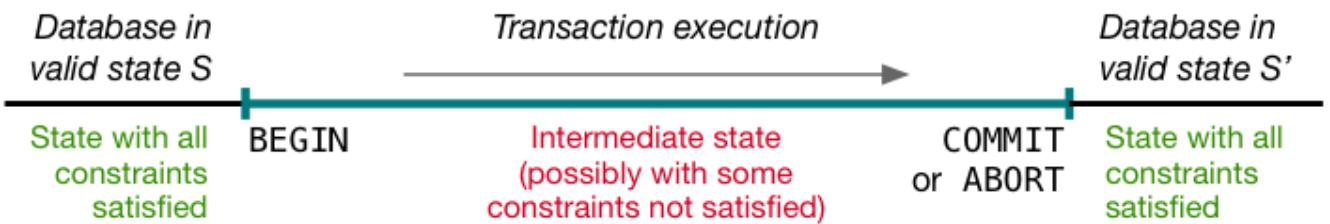
- successfully (**COMMIT**), with all changes preserved
- unsuccessfully (**ABORT**), with database unchanged



## ❖ Transaction Consistency

Transactions typically have intermediate states that are invalid.

However, states **before** and **after** transaction must be valid.



Valid = consistent = satisfying all stated constraints on the data

---

Produced: 15 Nov 2020

# Transaction Schedules

---

- Transaction Schedules
- Serial Schedules
- Concurrent Schedules
- Example Update Anomaly

## ❖ Transaction Schedules

When reasoning about transactions, we consider only

- **READ** - transfer data item from database to memory
- **WRITE** - transfer data item from memory to database
- **BEGIN** - start a transaction
- **COMMIT** - successfully complete a transaction
- **ABORT** - fail a transaction and unwind effects

All other operations are ignored (e.g. addition, testing, ...)

- take place in the memory space of one transaction
- have no affect on other transactions

## ❖ Transaction Schedules (cont)

Relating SQL to database reads/writes ...

- **SELECT** produces **READ** operations on the database
- **INSERT** produces **WRITE** operations
- **UPDATE, DELETE** produce both **READ + WRITE** operations

Assume: each operation involves one database item (e.g. one tuple)

Notation: items denoted **X, Y**, etc; operations denoted **R, W, C, A**

Thus, we see notation like: **R(X), R(Y), W(X), W(Y)**, etc.

Notes:

- items with same name in different transactions refer to a shared item
- typically don't use explicit **BEGIN** or **COMMIT** or **ABORT**

## ❖ Transaction Schedules (cont)

Showing SQL→Schedule, using bank transfer example

```
get balance in source account
get balance in destination account
if (source balance sufficient):
    update source by subtracting amount transferred
    update destination by adding amount transferred
```

If X = source account, Y = destination account, can be summarized as

```
R(X) R(Y) W(X) W(Y)
```

Note: we treat the **updates** simply as writes ...

- assume **UPDATE** = **R;W**, and **R;W** is atomic, so overall effect is just **W**

## ❖ Transaction Schedules (cont)

When multiple transactions run in parallel

- each transaction runs its own operations in a well-defined order
- but operations from different transactions interleave differently

Possible execution orders for operations of two transactions

```
-- no concurrency
T1: R(X) W(X) R(Y) W(Y)
T2:           R(X) W(X) R(Y) W(Y)

-- with concurrent execution
T1: R(X)       W(X)       R(Y)       W(Y)
T2:       R(X)       W(X)       R(Y)       W(Y)
```

## ❖ Transaction Schedules (cont)

Executing a single correct transaction ...

- maps the DB from a **consistent** state to another **consistent** state

Similarly, executing transactions sequentially ...



Arbitrary interleaving of operations can cause **anomalies**, so that ...

- two consistency-preserving transactions, running concurrently
- produce a final state which is not consistent

## ❖ Serial Schedules

Serial execution: **T1** then **T2** or **T2** then **T1**

T1: R(X) W(X) R(Y) W(Y)

T2: R(X) W(X)

or

T1: R(X) W(X) R(Y) W(Y)

T2: R(X) W(X)

Serial execution guarantees a consistent final state if

- the initial state of the database is consistent
- **T1** and **T2** are consistency-preserving

## ❖ Concurrent Schedules

Concurrent schedules interleave T1,T2,... operations

Some concurrent schedules are ok, e.g.

T1:	R(X)	W(X)	R(Y)	W(Y)
T2:		R(X)		W(X)

Other concurrent schedules cause anomalies, e.g.

T1:	R(X)	W(X)	R(Y)	W(Y)
T2:		R(X)		W(X)

Want the system to ensure that only valid schedules occur.

## ❖ Example Update Anomaly

Two concurrent transfers from same source account:

- T1 transfers \$200 X→Y, T2 transfers \$100 X→Y
- initial values: X=500, Y=100; final values: X=200, Y=400

T1	T2	X <sub>T1</sub>	X <sub>T2</sub>	X <sub>db</sub>	Y <sub>T1</sub>	Y <sub>T2</sub>	Y <sub>db</sub>
R(X)		500		500			100
X-200		300					
	R(X)		500				
W(X)		300		300			
	X-100		400				
	W(X)		400	400			
	R(Y)					100	
R(Y)					100		
Y+200					300		
W(Y)					300		300
	Y+100					200	
	W(Y)					200	200

---

Produced: 16 Nov 2020

# Serializability

---

- Serializability
- Conflict Serializability
- Conflict Serializability Example
- View Serializability
- View Serializability Example

## ❖ Serializability

Serializable schedule:

- concurrent schedule for  $T_1..T_n$  with final state  $S$
- $S$  is also a final state of a possible serial schedule for  $T_1..T_n$

Abstracting this needs a notion of **schedule equivalence**.

Two common formulations of **serializability**:

- **conflict serializability** (read/write operations occur in the "right" order)
- **view serializability** (read operations see the correct version of data)

## ❖ Conflict Serializability

Consider two transactions  $T_1$  and  $T_2$  acting on data item  $X$ .

Possible orders for read/write operations by  $T_1$  and  $T_2$ :

$T_1$ first	$T_2$ first	Equiv?
$R_1(X) R_2(X)$	$R_2(X) R_1(X)$	yes
$R_1(X) W_2(X)$	$W_2(X) R_1(X)$	no
$W_1(X) R_2(X)$	$R_2(X) W_1(X)$	no
$W_1(X) W_2(X)$	$W_2(X) W_1(X)$	no

If  $T_1$  and  $T_2$  act on different data items, result is always equivalent.

## ❖ Conflict Serializability (cont)

Two transactions have a potential **conflict** if

- they perform operations on the same data item
- at least one of the operations is a write operation

In such cases, the order of operations affects the result.

If no conflict, can swap order without affecting the result.

If we can transform a schedule

- by swapping the order of non-conflicting operations
- such that the result is a serial schedule

then we say that the schedule is **conflict serializable**.

## ❖ Conflict Serializability (cont)

Example: transform a concurrent schedule to serial schedule

T1:	R(A)	W(A)		R(B)		W(B)	
T2:			R(A)		W(A)		R(B) W(B)
<b>swap</b>							
T1:	R(A)	W(A)	R(B)			W(B)	
T2:				R(A)	W(A)		R(B) W(B)
<b>swap</b>							
T1:	R(A)	W(A)	R(B)		W(B)		
T2:				R(A)		W(A)	R(B) W(B)
<b>swap</b>							
T1:	R(A)	W(A)	R(B)	W(B)			
T2:					R(A)	W(A)	R(B) W(B)

## ❖ Conflict Serializability (cont)

Checking for conflict-serializability:

- show that ordering in concurrent schedule
- cannot be achieved in any serial schedule

Method for doing this:

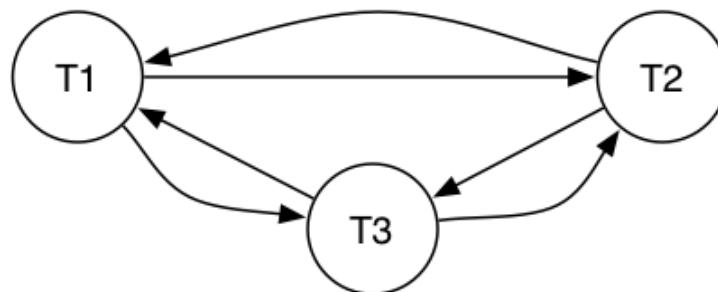
- build a **precedence-graph**
- nodes represent transactions
- arcs represent order of action on shared data
- arc from  $T_1 \rightarrow T_2$  means  $T_1$  acts on  $X$  before  $T_2$
- a cycle indicates *not* conflict-serializable.

## ❖ Conflict Serializability Example

Example schedule which is not conflict serializable:

T1:	R(X)		R(Y)	W(X)		W(Y)
T2:		R(X)			W(X)	
T3:		R(X)				W(X)
<b>attempted swaps</b>						
T1:		R(X)	W(X)		R(Y)	W(Y)
T2:		R(X)		W(X)		
T3:	R(X)				W(X)	

Precendence graph for the above schedule:



## ❖ View Serializability

**View Serializability** is

- an alternative formulation of serializability
- that is less conservative than conflict serializability (CS)  
(some safe schedules that are view serializable are not conflict serializable)

As with CS, it is based on a notion of schedule equivalence

- a schedule is "safe" if *view equivalent* to a serial schedule

The idea: if, across the two schedules ...

- they read the same version of a shared object
- they write the same final version of an object

then they are **view equivalent**

## ❖ View Serializability (cont)

Two schedules  $S$  and  $S'$  on  $T_1..T_n$  are **view equivalent** iff

- for each shared data item  $X$ 
  - if, in  $S$ ,  $T_j$  reads the initial value of  $X$ ,  
then, in  $S'$ ,  $T_j$  also reads the initial value of  $X$
  - if, in  $S$ ,  $T_j$  reads  $X$  written by  $T_k$ ,  
then, in  $S'$   $T_j$  also reads the value of  $X$  written by  $T_k$  in  $S'$
  - if, in  $S$ ,  $T_j$  performs the final write of  $X$ ,  
then, in  $S'$ ,  $T_j$  also performs the final write of  $X$

To check serializability of  $S$  ...

- find a serial schedule that is *view equivalent* to  $S$
- from among the  $n!$  possible serial schedules

## ❖ View Serializability Example

Example: consider the following concurrent schedule

T1:	R(A)	W(A)	R(B)	W(B)
T2:		R(A)	W(A)	R(B) W(B)

If view serializable, the read/write behaviour must be like one of

1. T1: R(A) W(A) R(B) W(B)  
T2: R(A) W(A) R(B) W(B)
  
2. T1: R(A) W(A) R(B) W(B)  
T2: R(A) W(A) R(B) W(B)

## ❖ View Serializability Example (cont)

Reminder of concurrent schedule

T1:	R(A)	W(A)	R(B)	W(B)
T2:		R(A)	W(A)	R(B) W(B)

In the concurrent schedule

- A: T1 reads initial, T2 reads T1's write, T2 writes final
- B: T1 reads initial, T2 reads T1's write, T2 writes final

In T1;T2

- A: T1 reads initial, T2 reads T1's write, T2 writes final
- B: T1 reads initial, T2 reads T1's write, T2 writes final

So, concurrent schedule is view equivalent to T1;T2

---

Produced: 15 Nov 2020