Innehållsförteckning

Relational database

- Simple and familiar data model
- The database is a collection of tables
- Each table has columns and rows
- **Relations**: mathematically a set of fixed length tuples, tables are basically relations.
- **Constraints:** a limitation on what values you can put in a table
    - Uniqueness constraints (values must be unique in the table)
    - Value constraints (a value must satisfy some simple condition)
    - Reference constraint (a value must be present in another table)

SQL (Postgres)

1 SQL: CREATE TABLE

- The subset of SQL that deals with creating tables is called the **Data Definition Language**, SQL DDL
- Where every table element is either:
    - a column
    - a constraint

The basic syntax is:

```
CREATE TABLE TableName (
    <list of table elements>
);
```

Types:

- INT – (a.k.a. INTEGER) for 32 bit signed integers
- REAL – (a.k.a. FLOAT) for 32 bit floating point values
- NUMERIC(p,s) – numbers with p digits before and s digits after '.'
- CHAR(n) – for fixed size strings of size n (like character arrays)
- TEXT – for variable sized strings
- VARCHAR(n) – for variable sized strings with max size n
- TIMESTAMP – for date+time (microsecond resolution)
- DATE and TIME – for dates and times of days independently

Primary key constraint:

- Every table should have a single **primary key** constraint.
- Can only be declare once… "PRIMARY KEY (…)"
- The primary key is the set of attributes used to identify individual rows
    - If multiple columns together form the key, it is called a *compound key*. The **combination** of columns must be unique.

Reference (foreign key) constraint:

- A set of references to attributes in another table which must exists.
- Unlike primary keys, a table can have any number of foreign key constraints.
    - Each constraint is checked independently, and valid data must satisfy all constraints.
- Like primary keys, multiple columns can together reference another table, a *compound reference*. The **combination of attributes is together a reference** to another table.

Unique constraints:

- Some tables have several keys (but one of them is always primary)

- Additional keys can be marked as UNIQUE, and the DBMS will prevent inserting rows with duplicate values.
- Like primary key constraints and reference constraints, multiple values can together create a uniqueness constraint.
    - **UNIQUE(c2, c3)**: The **combination of values in column c2 and c3 will be unique** across the whole table. The **value of the column c2 or c3 needs NOT to be unique**.

Value constraint (CHECK):

- Constraints on the values of columns in a table.
- Declared inside the table declarations.
- Value constraints can **NOT** check outside the values in the row being inserted

```
CHECK (pnumber > 0 AND pnumber < 100),
CHECK (grade IN (0,3,4,5)),
CHECK (price > discounted_price)
```

NOT NULL constraint:

- The NOT NULL constraint is a special constraint that says that a column cannot have the magic NULL non-value.
- **Should be added everywhere** unless you specifically want NULL-values.
    - Rule of thumb: NULL values are evil and will corrupt your data and soul.
- Added after the type of each column.
- Not needed for primary key attributes, they are automatically NOT NULL.

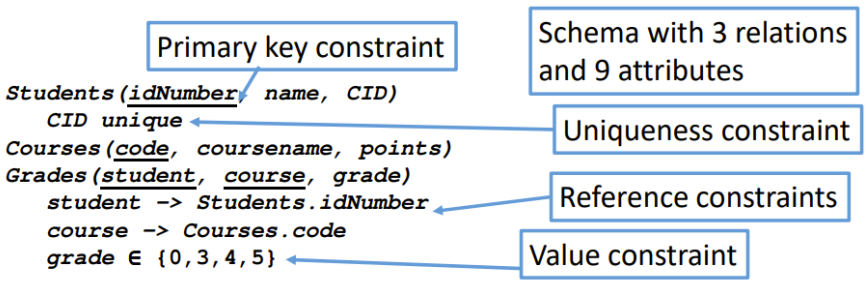
Default values (DEFAULT):

- You can add default values when **creating columns** and **in inserts**.
- The "default default value" for nullable columns is NULL.

```
lastName TEXT NOT NULL DEFAULT 'Doe',
incident TIMESTAMP DEFAULT CURRENT_TIMESTAMP
```

Schemas:

- Schemas are compact **'blueprints'** of relations, not part of SQL.
- Like CREATE TABLE, but with less technical detail and informal syntax.
- Format we use:



Translating schemas into SQL:

- Each relation becomes a table
- Each attribute becomes a column, decide an appropriate type
- All underlined attributes together make a single PRIMARY KEY
- References become foreign keys
- Unique/value constraints become UNIQUE/CHECK
- Add NOT NULL everywhere

Views:

- A **view** is a way of **giving a name to a query**. Views can be used much like tables.

General form:

CREATE VIEW <name> AS <query>;

- You can also drop (delete) views:
  DROP VIEW;
  This **does not delete any actual data**, since the view does not contain data, it only displays data from actual tables.

Removing tables (DROP):

- The SQL command DROP TABLE <table name>;
  Removes a table, including all the data stored in it.
  - This **will fail if other tables have references to the removed table**
  - There is **no confirmation dialogue**, no undo-button. Only the light humming of your hard drive as it deletes your carefully collected data

2 SQL DML: INSERT/MODIFY/DELETE

- The subset of SQL that deals with inserting, modifying, or deleting rows from tables is called the **Data Manipulation Language** (SQL DML)

Insert:

General form INSERT INTO <table name> VALUES (<expressions>);

- The **order of values should match the order of columns** in the CREATE TABLE statement.
- Can **fail due to constraints** on the table.

Delete:

General form:

```
DELETE FROM <table name> WHERE <condition on rows>
```

- Cannot "fail", if nothing to delete matching the condition, it will simply delete 0 rows.

Update:

- Used to modify any number of values in a table.

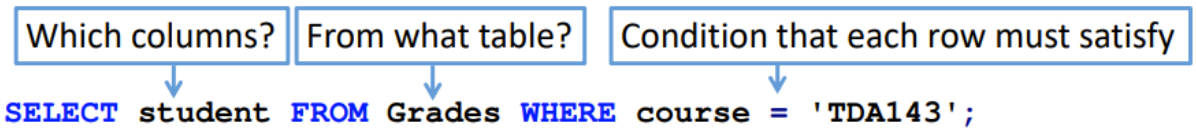
```
UPDATE <table name> SET <attribute = expression>
  WHERE <condition on rows>
```

- UPDATE **never removes or adds any rows**.

3 SQL: SELECT [Queries]

Queries:

- The result of each query is **essentially a table** although it has columns and rows of data, but **no constraints** and is **not persistent**.



- Compute the result by:
  - Taking the Cartesian product of the tables in FROM
  - Removing rows not matching WHERE
  - Removing columns not in SELECT

Cartesian product:

- Operation in set theory (thus applicable to relations!)
- If S = {1,2,3} T = {A, B, C} then the product S ⨯ T is all combinations (pairs):
  {(1,A), (1,B), (1,C), (2,A), (2,B), (2,C), (3,A), (3,B), (3,C)}
- In general, |S⨯T| = |S|*|T|
  (Example: if S has three elements and T has four, S⨯T has twelve element)

student	course
790401-1234	TDA357
790401-1234	TDA143
810509-0123	TDA143

⨯

code	points
TDA357	7.5
TDA143	7.5

=

student	course	code	points
790401-1234	TDA357	TDA357	7.5
790401-1234	TDA357	TDA143	7.5
790401-1234	TDA143	TDA357	7.5
790401-1234	TDA143	TDA143	7.5
810509-0123	TDA143	TDA357	7.5
810509-0123	TDA143	TDA143	7.5

3 rows	* 2 rows	= 6 rows
2 columns	+ 2 columns	= 4 columns

- In SQL, a query: “SELECT * FROM table1, table2;” will select the cartesian product of table1 and table2.

Qualified names:

- Problem when trying to work with columns with the same name in two different tables.
  - Solutions:
    - Use unique names in all tables.
    - Use **qualified names** to distinguish attributes.
      For example: Students.idNumber

JOIN keyword:

In general:

```
FROM TableA,TableB WHERE x=y
```

is the same as

```
FROM TableA JOIN TableB ON (x=y)
```

USING keyword:

Instead of:

```
SELECT Students.idNumber, name, grade
 FROM Students JOIN Grades
          ON Students.idNumber=Grades.idNumber;
```

I can write:

```
SELECT idNumber, name, grade
 FROM Students JOIN Grades USING (idNumber);
```

Magic!

- Translates to the condition:
  Students.idNumber=Grades.idNumber
- Also magically **removes the duplicate occurrence of idNumber** in the cartesian product! (So I don't need to use qualified names in SELECT).

NATURAL JOIN keyword:

Writing

```
SELECT *
 FROM Students NATURAL JOIN Grades;
```

Translates into

```
USING <all attribute with the same name in both tables>
```

- May accidentally join on the wrong attributes (Course.name = Student.name).
- Sensitive to renaming – it may work one day and fail horribly the next.
- Very difficult to describe in terms of simple operations (and thus unnatural).
- Loved by the masses because it is the most compact way of joining tables (Especially for things like "A NATURAL JOIN B NATURAL JOIN C" …).'

Aliasing tables and columns (AS):

- Both columns in SELECT and tables in FROM can be named/renamed:

```
SELECT C.name AS theName FROM Courses AS C;
```

Shorter qualified names ☺

Column name is 'theName' in result

Select from courses, but call it C

(FULL) OUTER JOIN keyword:

- Basic idea: Take the 'missing rows' from both joined tables (the ones that are not matched with any rows from the other in the result of the join) and add them with NULL for the attributes of the other table

Table: Phones

name	phone
Bart	11111
Lisa	22222
Maggie	33333

Table: Emails

name	email
Bart	bart
Lisa	lisa
Homer	homer

```
SELECT * FROM Phones FULL OUTER JOIN Emails
  ON (Phones.name=Emails.name);
```

Regular (inner) join

Extra outer-join rows

Phones.name	phone	Emails.name	email
Bart	11111	Bart	bart
Lisa	22222	Lisa	lisa
Maggie	33333	(null)	(null)
(null)	(null)	Homer	homer

- The weird column-merging stuff that NATURAL JOIN (and USING) does works sort of nicely here:
  - No nulls in joined columns
  - Looks like what I would expect a combination of the two tables to look like

```
SELECT * FROM Phones NATURAL FULL OUTER JOIN Emails;
```

name	phone	email
Bart	11111	bart
Lisa	22222	lisa
Maggie	33333	(null)
Homer	(null)	homer

LEFT/RIGHT OUTER JOIN keyword:

- Specifying left/right outer join (instead of full) means only missing rows from the left/right operand of JOIN are added:
  - No extra row for homer
  - Never any new null values in Phones.x (left side of result)

```
SELECT * FROM Phones LEFT OUTER JOIN Emails
  ON (Phones.name=Emails.name);
```

Phones.name	phone	Emails.name	email
Bart	11111	Bart	bart
Lisa	22222	Lisa	lisa
Maggie	33333	(null)	(null)

COALESCE keyword:

- COALESCE takes a list of values and returns the first non-null value.
- Typical use case: Replaces null values with constants (of matching type)

```
SELECT name, COALESCE(email, 'no email') AS email
 FROM Emails FULL OUTER JOIN …
```

name	email
Bart	bart
Lisa	lisa
Maggie	no email
Homer	homer

Summary, Outer/inner joins

Phones(name, phone)
Emails(name, email)

- Informally, which names are included in these queries?

```
SELECT * FROM Phones NATURAL JOIN Emails;
```

- Answer: Everyone with a phone and an email

```
SELECT * FROM Phones NATURAL LEFT OUTER JOIN Emails;
```

- Answer: Everyone with a phone

```
SELECT * FROM Phones NATURAL RIGHT OUTER JOIN Emails;
```

- Answer: Everyone with an email

```
SELECT * FROM Phones NATURAL FULL OUTER JOIN Emails;
```

- Answer: Everyone with a phone or an email

- In each case, the magical nature of NATURAL JOIN makes sure that the name columns are merged (result has three columns)

Set, Bags, or Lists

	Order matters	Order does not matter
Duplicates allowed	List	Bag (multiset)
No duplicates	Ordered set	Set

- Sets, Bags and Lists are three data structures for simple collections:
  - **Sets** have **no internal ordering** and **no duplicates**
  - **Bags** (a.k.a. multisets) have **no ordering** but **can have duplicates**
  - **Lists have ordering** (each value has a position in the list) **and duplicates**
- An **SQL table** is typically considered a **Set** (primary key ensures unique rows).
- **SQL Query** results are often **"morally" Sets**, but can also be bags or lists
- Often, we can ignore the difference, but sometimes this is important (when we care about ordering or there is a risk of duplicates).

Removing duplicates:

- By adding DISTINCT after SELECT, duplicate rows will be removed.
- Gives the query set-semantics.

Table: Grades

idNumber	course	grade
750202-2345	TDA357	4
790401-1234	TDA357	3
810509-0123	TDA357	3

```
SELECT course, grade
 FROM Grades;
```

course	grade
TDA357	4
TDA357	3
TDA357	3

```
SELECT DISTINCT course, grade
 FROM Grades;
```

course	grade
TDA357	4
TDA357	3

Set operations:

- There are three set-opertions in SQL:
  - UNION
    - Union just combines all rows from two queries
    - Removes all duplicate rows (because it's a set operation)
    - Does not preserve ordering (because it's a set operation)
    - Uses column names from left operator
  - INTERSECT
    - Takes the intersection of two queries (all rows that appear in both)
  - EXCEPT
    - Takes the difference of two queries (removing the contents of the second from the first)

Table: Lectures

ltime	lroom	teacher
11-06 8:00	GD	Jonas
11-08 10:00	GD	Matti

Table: Exercises

etime	eroom	subject
11-06 8:00	GD	SQL
11-06 13:00	HB	ER

```
(SELECT lroom,ltime,'lecture' AS topic
 FROM Lectures
 WHERE teacher = 'Matti')
UNION
(SELECT eroom,etime,subject
 FROM Exercises);
```

1 row

2 rows

Result:

lroom	ltime	topic
GD	11-06 8:00	SQL
GD	11-08 10:00	lecture
HB	11-06 13:00	ER

```
(SELECT lroom,ltime FROM Lectures)
INTERSECT
(SELECT eroom,etime FROM Exercises);
```

Result:

lroom	ltime
GD	11-06 8:00

```
(SELECT lroom,ltime FROM Lectures)
EXCEPT
(SELECT eroom,etime FROM Exercises);
```

Result:

lroom	ltime
GD	11-08 10:00

- If we want to keep (or don't care about) duplicates, we can use UNION ALL (also INTERSECT ALL and EXCEPT ALL) to keep duplicates (bag-semantics).
- Presumably, it is more efficient.
- Ordering is not necessarily preserved.

Ordering:

- Sometimes, the order of rows in the result is important for the user.
- An ORDER BY [ASC/DESC] clause can be added at the end of any SELECT.

Table: Numbers

owner	num
Bart	44444
Lisa	22222
Bart	33333
Homer	11111

```
SELECT *
 FROM Numbers
 ORDER BY num DESC;
```

Descending order

owner	num
Bart	44444
Bart	33333
Lisa	22222
Homer	11111

```
SELECT *
 FROM Numbers
 ORDER BY (owner,num) ASC;
```

Ascending order (first by owner, then num)

owner	num
Bart	33333
Bart	44444
Homer	11111
Lisa	22222

4 SQL: Aggregation

Aggregates:

- When you need to process **groups of values together**, such as counting, calculating averages, calculating totals, and calculating max/min.
- These operations are called **aggregates**: aggregate a set of values into a single value.
- Aggregate functions:
  - COUNT counts rows
  - AVG computes averages
  - MIN computes minimum
  - MAX computes maximum
  - SUM computes total
- Always gives a single row
- WHERE applied before aggregation
- Cannot mix columns and aggregates ~~(SELECT student, AVG(grade))~~

Table: Courses

name	points
Databases	10
Project	15

Table: Grades

student	course	grade
Lisa	Databases	4
Lisa	Project	5
Bart	Databases	3
Bart	Project	0

```
SELECT COUNT(*) AS Passing
  FROM Grades
  WHERE grade >= 3;
```

Passing
3

```
SELECT AVG(grade)
  FROM Grades
  WHERE grade >= 3;
```

AVG
4.0

```
SELECT MAX(points), MIN(points)
  FROM Courses;
```

MAX	MIN
15	10

Grouping:

- The GROUP BY clause divides the rows returned from the SELECT statement into groups.
- For each group, you can apply an aggregate function e.g. SUM to calculate the sum of items or COUNT to get the number of items in the groups

Table: Grades

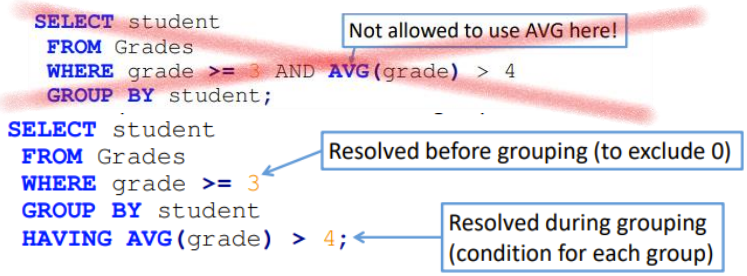
student	course	grade
Lisa	Databases	4
Lisa	Project	5
Bart	Databases	3
Bart	Project	0

```
SELECT student, AVG(grade)
 FROM Grades
 WHERE grade >= 3
 GROUP BY student;
```

The selected columns must be a subset of the columns we group by!
(Selecting course here would not make sense)

HAVING keyword:

- The HAVING clause specifies a search condition for a group or an aggregate.
- The HAVING clause is often used with the GROUP BY clause to filter groups or aggregates based on a specified condition.



5 SQL: Subqueries

- Subqueries are queries inside another query.
- Can have subqueries in:
  - FROM, WHERE, SELECT, other subqueries
- The **EXISTS** operator is a boolean operator that tests for existence of rows in a subquery. The **NOT EXISTS** operator does the opposite.

```
SELECT name FROM Courses AS C
 WHERE NOT EXISTS
   (SELECT * FROM Grades WHERE grade=5 AND course = C.name)
```

Refers to a value in the superquery

This query selects all courses that have no student with a grade of 5

- Note how the condition in the inner query refers to a value in the outer query (C.name), we say that the subquery is a **correlated query**.
  - The subquery cannot be executed by itself
  - The qualified name is not needed but highly recommended for readability

WITH keyword:

- The WITH clause offers a nice way to structure subqeries, by creating "helper tables" (similar to views, but only existing locally).

General syntax:

```
WITH <query1> AS <Name1>,
     <query2> AS <Name2>
SELECT … FROM <Name1>, <Name2>;
```

"helper queries"

Final result query

- Note that the whole thing is a single query that gives one result table, but it contains subqueries.

Complete SQL Queries:

SQL Querys

- A query with almost everything:

```
SELECT <columns/expressions>
 FROM <tables/subqueries/JOINS>
 WHERE <condition on rows>
 GROUP BY <columns>
 HAVING <condition on groups>
 ORDER BY (<columns/expressions>) [ASC/DESC];
```

- Set operations: <query1> **[UNION/INTERSECT/EXCEPT]** <query2>
- Expressions are built from: columns, constants (0,'hello',…), operators(+,-,…), functions (COALESCE, aggregates, …)
- Conditions can use columns,constants,AND/OR/NOT,IN,EXISTS,<,>,=, IS NULL …

Design using ER

1 ER: Modelling

Domain descriptions:

- The domain of a database is an informal description of everything a database should contain
- Typically given by a client (an expert in the domain, but not in databases)
- Natural language, no reliable automatic way to translate to SQL code.
- Something ambiguous on important details
- Sometimes hard to understand if you are not a domain expert

Modelling domains:

- The database we create is based on a model of the domain
- A model contains formal, well defined definitions

The Entity-Relationship model (ER)

- A high-level design model that expresses every aspect of the database as **entities** (entity sets to be more precise), **relationships** (not the same as relations) and **attributes.**
    - **Entities:** Things from the domain: courses, employees, car models, cars, …
    - **Relationships:** Connects the different entities, like "a car is of a car model", or "an employee has another employee as its boss", "courses have students", …
    - **Attributes:** Simple properties of entities: names of courses, salaries of employees, …

Entities:

- **Objects** with **attributes**.
- **RULE OF THUMB:** *Entities can exist independently from other entities*, for instance courses and teachers, or players and teams etc…
- Objects that are **inherently dependent** on other entities are **NOT** entities.

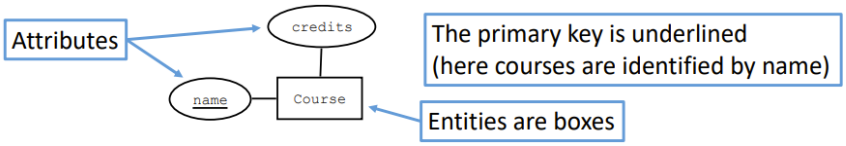
Attributes:

- "Simple" **properties** of entities.
    - Can be numbers, text strings, etc.
    - Values of every attribute should fit in a table cell.
- Something that **every instance** of that entity has, like each course having a name or each car having a license number.
- What are **NOT** attributes?
    - Collections of values (like a list of students)
    - Optional values (if a car *sometimes* has an owner, owner is not an attribute of the car entity)
    - Things that refer to attributes of other entities (see relationships)

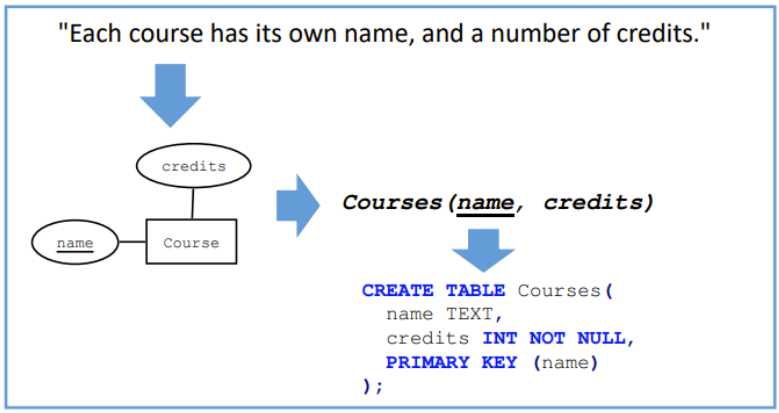
2 ER: Diagrams

Drawing ER-diagrams:

- ER-diagrams are visual representations of ER-models.
- They can easily/mechanically be translated into relational schemas.



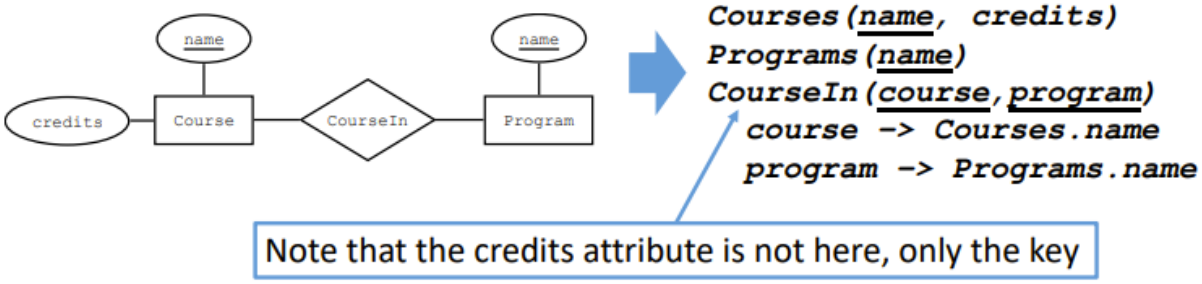
Translating to a relational schema:



- Entities are by convention named in singular (Course)
- Relations are named in plural (Courses)
- We also follow the convention we are using for relations/tables, meaning capitalized entity names and common attribute names.

Relationships in ER-diagrams

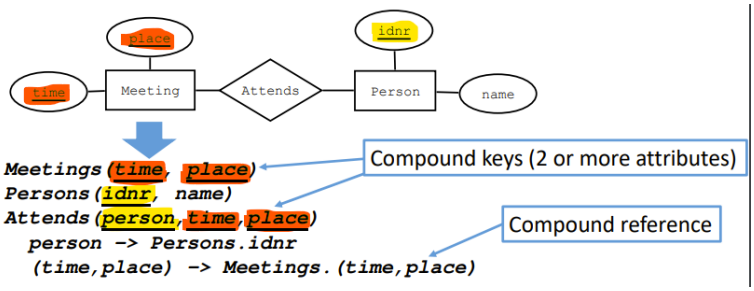
- Relationships are drawn using a diamond-shapes.
- Names typically describe the relationship (OwnedBy, RegisteredTo, …), or sometimes just adding the related entities names (like ProgramCourse).
    - Should relay the intent of the relationship, and work as a table name



- The **primary key of CourseIn is the PKs of both referenced relations**:
    - A course can be in several programs (same course, different program)
    - A program can have multiple courses (same program, different course)
    - A program cannot have the same course multiple times (both equal)

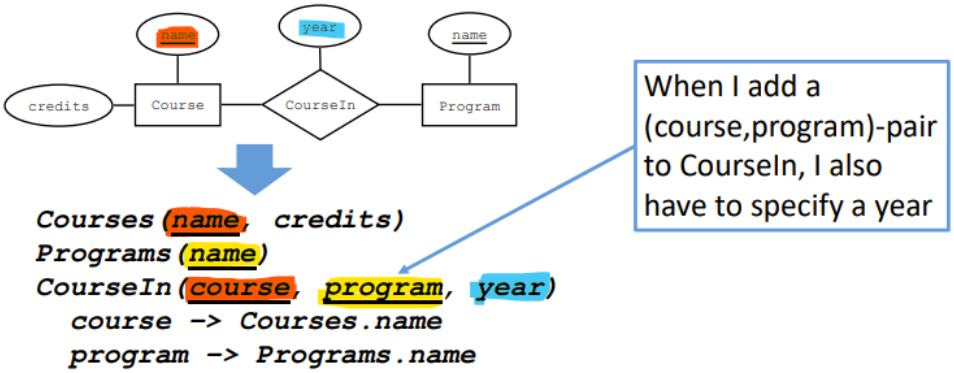
Compound keys and relationships:

- Always include the **WHOLE KEY** of both the related relations.



Attributes on relationships:

- What if we wanted to add an attribute stating which year in a program a certain course is?
  - The year is not an attribute of course (because different programs may have the course in different years)
  - The year is not an attribute of program (because different courses may be in different years of the program)
  - The year is an attribute of the relationship between them!



  - **Identifying attributes on relationships in domains**
    - Things like:

      "X may have a z in/for/at a Y"

      where **X and Y are entities** and **z is an attribute** typically signify that **z is an attribute of a relation.**
    - Example:

      "Teachers can be assigned roles in courses"

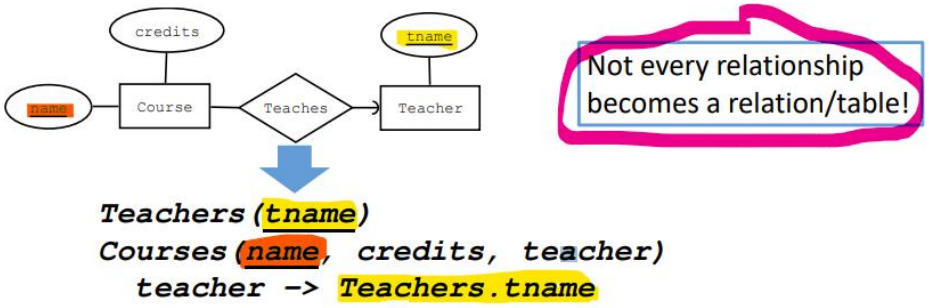
      (role is an attribute of a relationship (i.e., 'teaches') between teachers and courses, unless roles should be entity…)

3 ER: Multiplicity

- The relationships we have seen so far are called many-to-many.
  - E.g., A course can belong to many programs and programs have many courses.
- There are other variants…

Many-to-exactly-one:

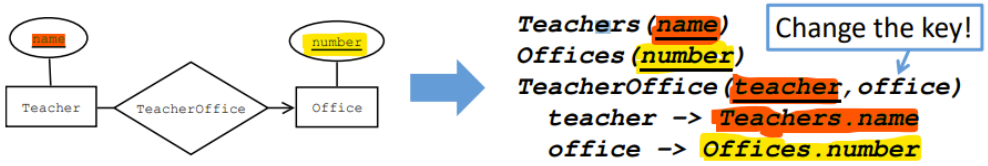
- Example:
  The Teaches relationship models *"each course has a single teacher".*
  If the arrow was in the **other direction** it would model *"each teacher has a single course".*
- Note the **exception to the guideline that entities can exist independently** (since courses cannot exist without teachers)



  - **Identifying many-to-exactly-one in domains**
    - Anything like
      "Every X has a Y"
      (**where Y is another entity**) are typically many-to-exactly-ones.
    - It's common that the language is ambiguous, e.g.,
      "Xs have Ys"
      can mean that **each X can have multiple ys**, or **that they each have one**.
    - If you **want to add an attribute**, but that **attribute is more accurately modelled as an entity**, you should use many-to-one relationships.

Many-to-at-most-one (many-to-one-or-zero):

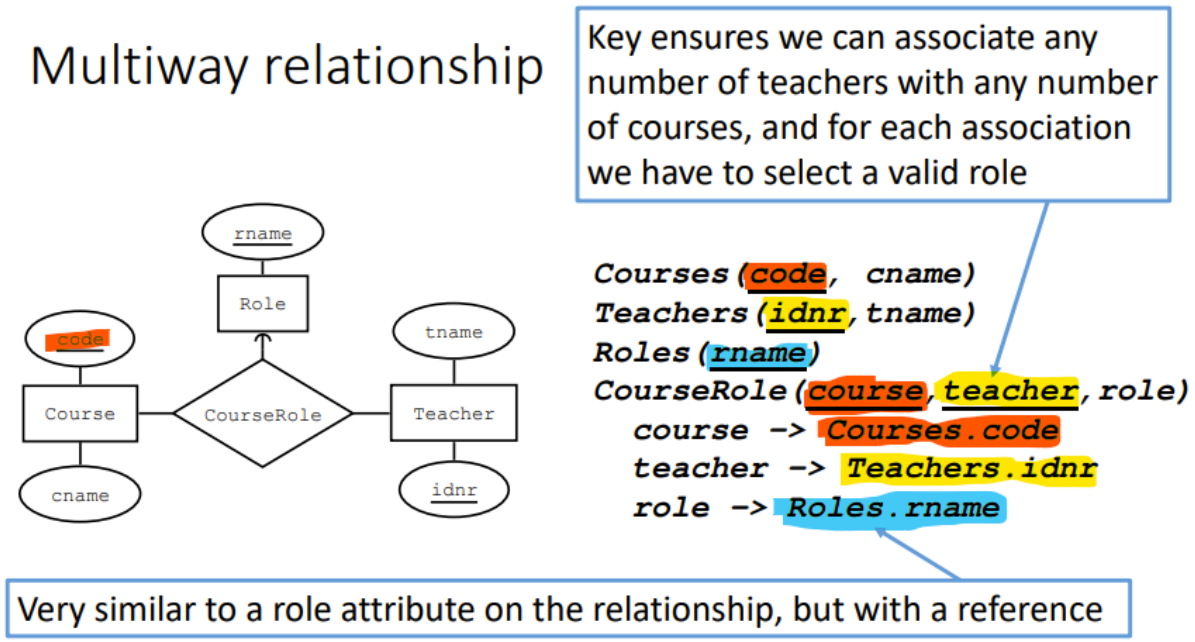
- Example:
  This diagram expresses *"Some teachers have an office*", or *"A teacher may have an office"* or equivalent.



  - **Identifying many-to-at-most-one in domains**
    - Anything like
      "X may have a Y" or *"Some Xs have a Y"*
      (where **Y is an entity**).
    - Often text is ambiguous about many/exactly one (e.g.
      "courses have teachers"
      could sometimes mean that many courses do, but not all)

Multiway relationships:

- A relationship can connect to more than two entities, although fairly rare.



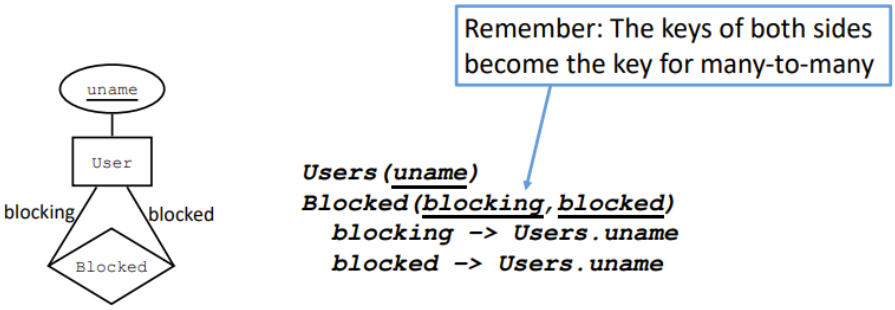
- Multiway relationships that are **many-to-many-to-many** and **many-to-many-to-one** seem to **make sense**, other combinations are more difficult to interpret.
    - **Identifying multiway relationships in domains**
        - Something like

          "Xs have Ys in Zs"

          where X, Y and Z are all entities.
        - Ambiguity: "Xs have Ys and Zs" **usually mean two separate relationships** (X-Y and X-Z) but **sometimes a multiway relationship**.
        - *"Teachers have roles in courses"* (assuming role is an entity not an attribute), *"A person must have a contract for each project they are involved in"* (assuming persons, contracts and projects are entities)
        - Identifying multiplicity of the various connects is often difficult

Self-relationships:

- Relationships between two entities of the same type.
- Some things that cannot be expressed in ER-diagrams:
  - Can a value be related to itself? (I'm my own boss? I block myself?)
  - Can a there be cycles (I'm the boss of my bosses' boss? I'm blocked by a person I block?)
  - Is the relationship symmetric, e.g. for a Sibling-relationship: If a is a sibling of b, then b must also be a sibling of a.
- These can be expressed in sidenotes/comments, but may be difficult to implement in SQL



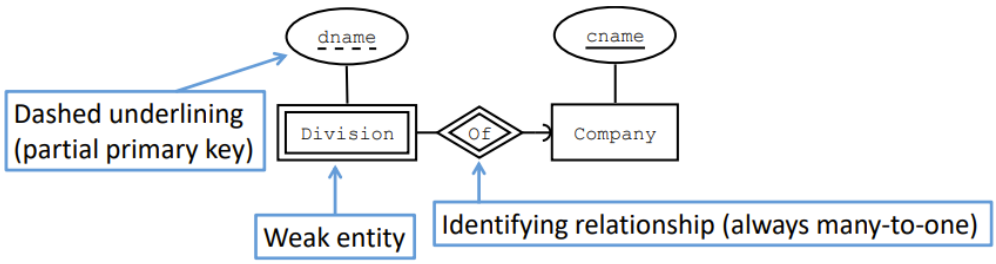
  - **Identifying self-relationships in domains**
    - Anything on the form

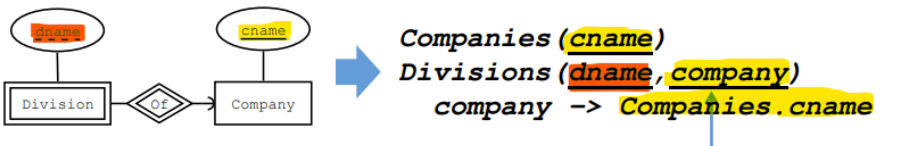
      "X has … to [an]other X".
    - If you want to model any kind of tree-structure (many-to-at-most-one) or graph structures (many-to-many) on values of an entity.

4 ER: More on entities

Weak entities:

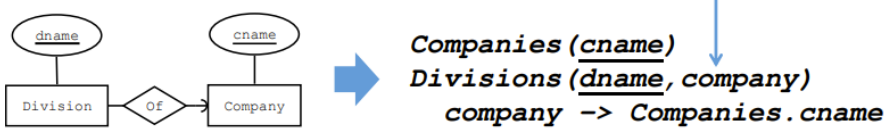
- A weak entity **cannot be identified only by its own attributes**
  - It requires support from at least one other entity.
- The diagram below expresses that a division is identified by its name along with the identity of the company it belongs to:





The key is the only difference!

- Compare to a regular many-to-one:

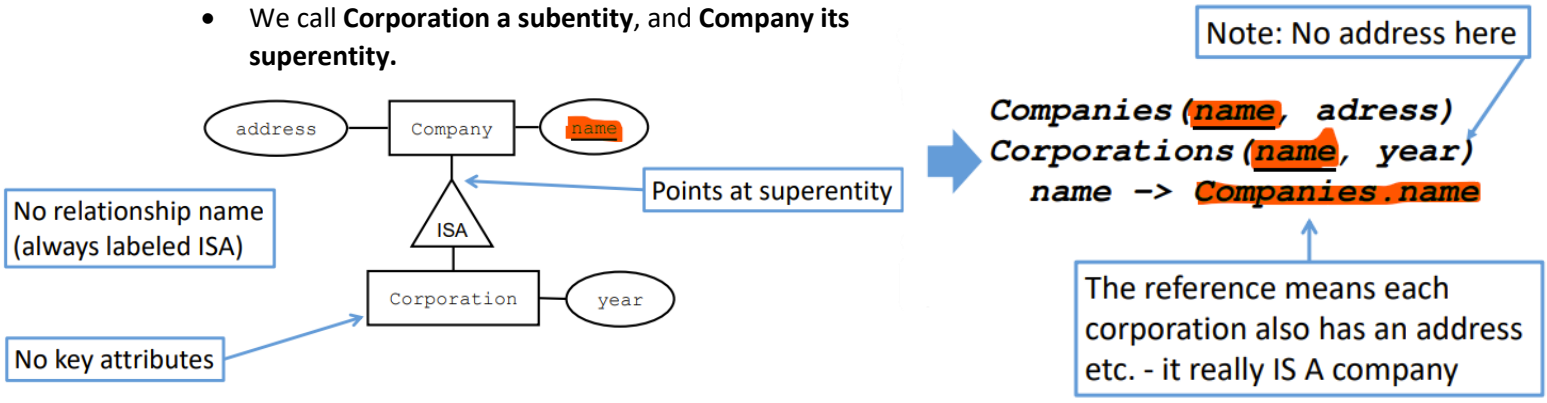


- **Identifying weak entities in domains**
  - Things like

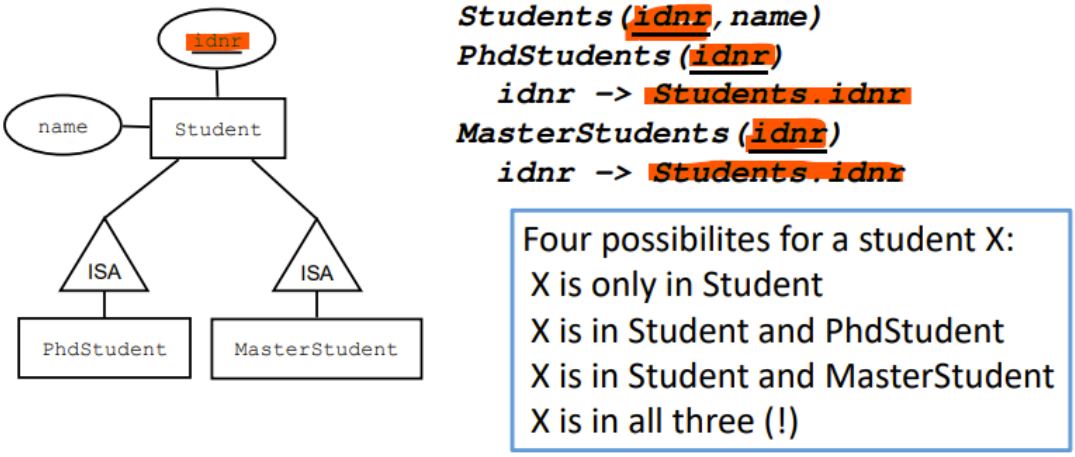
    "Player numbers are unique within teams" or *"players can have the same number assuming they are on different teams".*
  - If you notice that the **attributes you have determined for an entity** are **not sufficient to identify members**, perhaps it should be a weak entity

Inheritance in ER: ISA-relationships (ISA = "is a")

- Example:
  This ER diagram expresses *"Corporations are a special kind of companies, they have a year in addition to all properties of other companies."*
- We call **Corporation a subentity**, and **Company its superentity.**



- Make a new relation that **only has the primary key of the superentity** and the extra attributes added by the subentity, and a reference to the superentity.
- In ER (unlike most Object Oriented Programming), **a value can be a member of several subentities** (if it is also a member of the superentity).

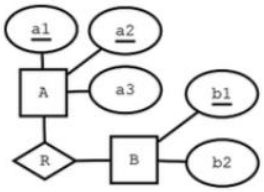


- Subentities can **never have any key attributes of their own**!
- **Defies the concept of inheritance**, if entity X is not identified by the same attributes as entity Y, we can never claim "X is a Y", it is something else.
- If you **need additional identifying attributes, use weak entities**
- In a way, weak entities are "subentities with extra identifying attributes

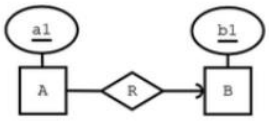
- **Identifying ISA-relationships in domains**
  - When it makes sense to say
    "X is a Y",
  - Like
    "Corporation is a Company"
    or
    "Employee is a Person"
    or
    "Car is a Vehicle"
  - In domain texts, it may also be stated as variations of *"some X have y"*, where **y is an attribute and not another entity** (optional attributes)
  - Sometimes when you want to model a subset of some entity even if they don't have extra attributes.

5 ER: Order of translation

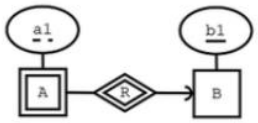
- Always **start with an entity whose translation does not depend on knowing the keys of other entities**
  - Example: If two entities have a many-to-exactly-one relationship, **always start with the entity on the exactly-one side**
  - Example2: Translate **superentities before you translate subentities**
  - Basically: Do **translations in reverse order of how the arrows are pointing**
- Do **many-to-many relationships last**
- <u>If you find yourself writing a reference to a relation you have not yet translated, you are doing things in the wrong order</u>



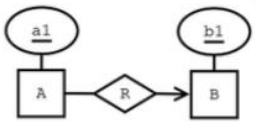

```
A(a1,a2,a3)
B(b1,b2)
R(a1,a2,b)
    (a1,a2) -> A.(a1,a2)
    b       -> B.b1
```



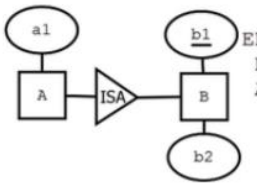

```
A(a1,b)
    b -> B.b1
B(b1)
```




```
A(a1,b)
    b -> B.b1
B(b1)
```




```
ER-Approach:
 A(a1)
 B(b1)
 R(a,b)
   a -> A.a1
   b -> B.b1

Null Approach:
 A(a1,b (or Null))
    b -> B.b1
 B(b1)
```

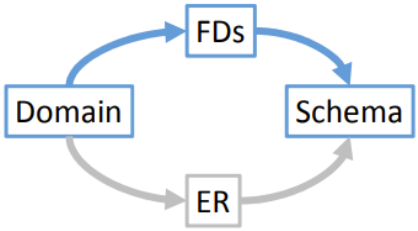



```
ER-Approach:
 B(b1,b2)
 A(b1,a1)
   b1 -> B.b1

Null-Approach:
 B(b1,b2,a1(or Null))

OO-Approach:
 A(b1,b2,a1)
 B(b1,b2)
```

Some things cannot be expressed in ER:

- Secondary keys/uniqueness constraints
    - Can be **identified using functional dependencies**
- Value constraints (like "grade is one of 'U','3','4','5' ")
    - Can be **added as sidenotes/comments**
- "Relationships between relationships"
    - Example: If we have one relationship for access to a company car, and one for currently using it, we cannot express that the latter is a subset of the former.
    - Can sometimes be implemented by cleverly modifying the schema (e.g. adding/modifying references) after translation from ER.

Design using Functional Dependencies and normal forms



1 FD: Normalization

Normalization in a nutshell:

- Extract a bunch of **formal statements from the domain description**.
- **Compute a normalised database schema** from the statements

- Highly systematic, almost mechanical process.
- By a carefully constructed normalization algorithm, the normal form that the schema ends up in will satisfy some important properties.

2 FD: Functional Dependencies (FD)

- A functional dependency is written as:
  - *<set of attributes>* → *<attribute>*
    - Example: **room time → course (DO NOT CONFUSE WITH REFERENCES)**
    - Pronounced: *"room and time **determines** course"*
    - Interpretations of the example:
      - ❖ If we **know room and a time**, we can **uniquely determine course**.
      - ❖ There can be **at most one course value for each (room,time)-pair**.
      - ❖ There **exists a partial function f** that **takes a room and a time** and **yields a course**.
    - In a domain it might have said something like: *"courses can book rooms at any free times"*
  - Can be **true** or **false**.
- Functional dependencies **use cases:**
  - ✓ Check if they **hold for a specific data set**.
  - ✓ Check if **a design ensures they hold for all data sets**.
  - ✓ Express **desired properties of a design**.

Functional dependencies as a property of data:

- One way of formally defining **functional dependency x1 x2 … → y:** *For **R(y x1 x2 …)**, if **two rows agree on x1, x2** … they must **also agree on y**.*
  - Two rows "agreeing on x" just means the x-column(s) have the same value.
- Knowing what it means for an FD to hold for a data set, we can determine if a design (schema) guarantees that it holds for all valid data sets.

Bookings ***(courseCode, name, day, timeslot, room, seats)***
(day, timeslot, room) UNIQUE

- Does the schema above guarantee …
- day timeslot room → courseCode
  - Yes (through UNIQUE constraint)
- day timeslot room coursCode → seats
  - Yes (through primary key and/or UNIQUE)
- room → seats
  - No ☹
- courseCode → name
  - No ☹

Counterexample of room → seats and courseCode → name

courseCode	name	day	timeslot	room	seats
CC1	N1	Tuesday	0	R1	0
CC1	N2	Tuesday	1	R1	1

Different timeslot, so no key violation

Single/Multiple FDs:

- It is common to have multiple attributes on the right-hand side of FDs
  x y z → a b c
- This means exactly the same as these three FDs:
  x y z → a
  x y z → b
  x y z → c
- It is **not the same with the left-hand side**!
  x y → a **does not mean** x → a!

Formal properties of FDs:

- FDs have 3 notable mathematical properties, known as **Armstrongs axioms**:
  - Transitivity
  - Augmentation
  - Reflexivity
- **Transitivty**
  - X → Y **AND** Y → Z => X → Z
  - Note that Y is an attribute set here, so X → Y may be multiple FDs with the same left-hand side.
- **Augmentation**
  - X1 X2 … → Y => **For all Z:** Z X1 X2 … → Y
  - Intuitively: You can add any attributes you want to the left-hand side of a valid FD and still get a valid FD.
    - Think: *"knowing an extra attribute never prevents us from finding Y"*
- **Reflexivity**
  - **For all X:** X → X
    (X determines itself)
  - By augmentation, X → Y whenever Y ∈ X
    - Example: a b c → b
    - These are called **trivial dependencies** and should be ignored since they don't add anything useful.

Minimal basis:

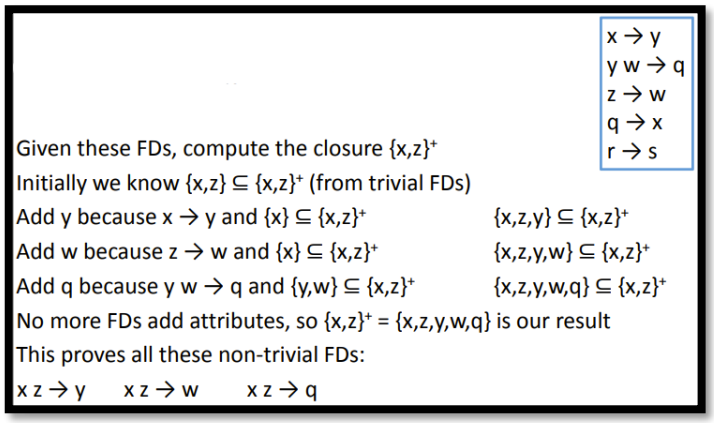
- The **minimal basis F⁻** of a **set of functional dependencies F** is a set equivalent to F but with the following properties:
  - F⁻ has **no trivial dependencies.**
  - No dependency in F⁻ follow from other dependencies in F⁻ **through augmentation or transitivity.**
- The **opposite of the transitive closure**, you are trying to reduce a set of functional dependencies to its smallest form. The original set can be derived from just the minimal basis.

Suppose we are given this set of FDs (5 total), what is a minimal basis?
a → b
b → c
a d → b c d

a d → d is removed because it is trivial
a d → b is removed because it is implied by a → b (augmentation)
a d → c is removed because it is implied by a → b and b → c (transitivity and augmentation)
Final set: a → b, b → c

Transitive closure:

- The **transitive closure X^+** of **set of attributes X** is the set of attributes that can be functionally determined by X.
- In other words:
  X^+ = All attributes Y such that X → Y
  - Includes ALL derived functional dependencies.
  - Includes trivial dependencies.
  - X^+ is closed in the sense that any FD from attributes in X^+ lead back to X^+
- Can be computed by a simple algorithm from any set of FDs:
  - Start with $X^+ = X$ (an under-approximation)
  - Repeat until done:
    For any FD Y → Z such that $Y \subseteq X^+$ | add z to X^+



- Basically ALL the functional dependencies you can derive from another set of functional dependencies… you're basically expanding the original set of functional dependencies and deriving everything you can from it. Think of it as the **opposite of the minimal closure**.

Keys and superkeys:

- We can define the property of being a key of a relation using FDs
- Intuitively: A set of attributes is a **superkey** if it **determines all other attributes**.
- Formally: The **attribute set X** is a **superkey of R** if **X^+ contains all attributes of R.**
- **Minimal (super-) Keys:**
  - X is a (**minimal**) **key** if removing any attribute from X makes it a non-superkey
    - Saying only *"key"* usually mean minimal key
    - Each superkey is a superset of at least one minimal key
    - Each key is a superkey (but not the other way around)
    - Adding any attribute to a superkey makes a new superkey
- Think like this: superkeys can be bloated and define R. Because they can be bloated you can remove an attribute from the superkey and still define R, assuming the superkey is bloated. Bloated mean they contain "unnecessary" attributes which can be derived from other attributes already in the superkey.
  On the other hand, minimalkeys are NOT bloated and you can thus not remove any attribute from them, or you will change R.
- **To find a key**: Start with all attributes (a superkey) and remove attributes until it is a key – finding all keys is more work though. (Technically the minimal key is also a superkey…)

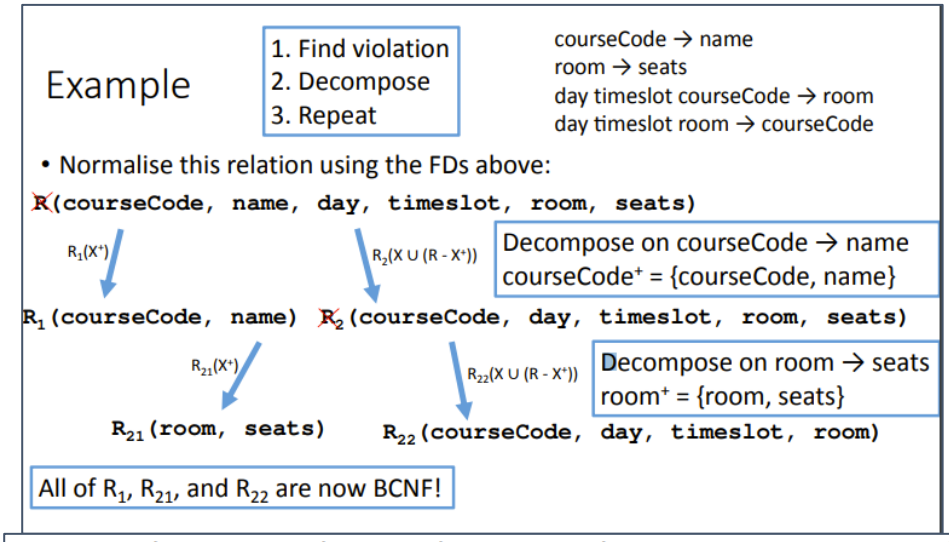
3 FD: Normal forms and normalization

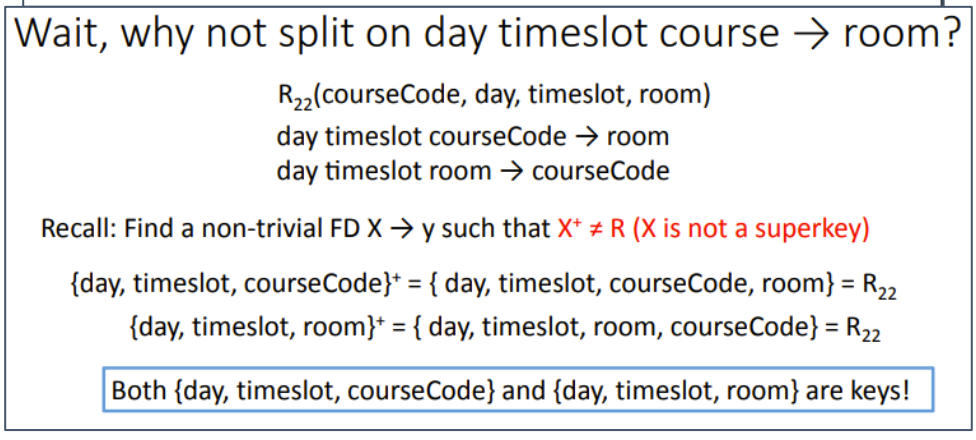
- **Design using normalization**:
    1. Identify **all the attributes in the domain** and place them in **one big relation**
       D(x, y, z, …)
    2. Collect/derive FDs from D
    3. Normalize D to get your design
- Normalization is a **recursive** procedure, **to normalize R:**
    1. Check if R is **already a normal form**, if it is we are done.
    2. Otherwise **decompose R into relations R_1 and R_2 and normalize both.**
- Note: A normal form is not the same as a canonical form, **there may be multiple normal forms derived from the same initial domain**.

4 FD: BCNF, the Boyce-Codd Normal Form

BCNF Normalisation algorithm:

1. Find a **non-trivial FD X → Y** such that **$X^+ \neq R$** (X is NOT a superkey).
2. If there is **no such FD**, then you are **done**! R is already in BCNF!
3. Otherwise:
   Decompose **R into $R_1(X^+)$** and **$R_2(X \cup (R - X^+))$** and normalize them.
   (Basically derive everything you can from X and remove everything from the derivation from R EXCEPT the original X that you derived from.)
4. (Note: R is replaced by R_1 and R_2, so R is NOT present in the final schema)





Determining keys from normalization:

- Keys can be determined using FDs (and closures) after decomposing (see previous page)

```
R1 (courseCode, name)
R21 (room, seats)
R22 (courseCode, day, timeslot, room)
    (day, timeslot, room) UNIQUE
```

Multiple keys: Use one as primary key, the other(s) UNIQUE

courseCode → name
room → seats
day timeslot courseCode → room
day timeslot room → courseCode

Determining references from normalization:

- General pattern: When decomposing R, add a reference X -> R_1.X to R_2
  (Will not always work, particularly if R_1 or R_2 is later decomposed)

```
R1 (courseCode, name)
R21 (room, seats)
R22 (courseCode, day, timeslot, room)
    (day, timeslot, room) UNIQUE
    courseCode -> R1.courseCode
    room -> R21.room
```

Finding all FDs:

- Consider this simple situation with four attributes R(x,y,z,w) and two functional dependencies: x → z and y z → w.
- When normalizing R it may be **important to know that there is another FD** that can be derived from these: x y → w
- **In principle, you should consider all non-trivial derived FDs** but sometimes this a large set and it is easy to miss FDs
- Essentially you have to **consider every left-hand side and compute closures**

A flaw of BCNF:

- Same example as before:
  R(x,y,z,w) where x → z and y z → w
  - If we decompose on x → z ({x}+ = {x,z}) we get
    - R1(x,z) {x} is the only key
    - R2(x,y,w) {x,y} is the only key
  - Both **relations are in BCNF** with regards to the given FDs, but now y z → w **is not guaranteed by the schema**.

5 FD: 4NF normalisation

- Another normal form.
- Has **multivalued dependencies (MVDs)**
- We write:

  x1 x2 x3 … ↠ y1 y2 y3 …
  - Note: **both sides are sets of values** and we **cannot split the right-hand side**.

Multivalued dependencies:

course	book	author	teacher
Databases	DTCB	Ullman	Jonas
Databases	DTCB	Ullman	Aarne
Reglerteknik	RTB 1	Author1	Teacher3
Reglerteknik	RTB 2	Author2	Teacher3

- For our example, we would say **course ↠ teacher**
- This means that **each course value has a set of teacher values** that is **independent** from all other values (author and book).
- This is exactly the same as saying **course ↠ book author**
- **Formally:**
  - The claim that **X↠Y holds for relation R** means:
    - For every **pair of rows row t and u in R that agree on X**, we can find **a row v** such that:
      - v agrees with both t and u on X
      - v agrees with t on Y
      - v agrees with u on R - X - Y (all attributes not in the MVD)

Example: course ↠ teacher

If we remove any row, the MVD won't hold

Table: Courses

course	book	author	teacher
Databases	DTCB	Ullman	Jonas
Databases	DTCB	Ullman	Aarne
Databases	DTCB	Widom	Jonas
Databases	DTCB	Widom	Aarne

Row t

Row u

Row v:
v.(book,author)=u.(book,author)
v.teacher = t.teacher

Verifying MVDs on data is hard:

- To check if an FD holds: Just **group values up by the left-hand side** and check that **all rows in each group** have the **same value for the right-hand side**
- To check if an MVD holds: Check **every individual pair of values with identical left-hand sides** and search for a **row with correct values**
- I find a more intuitive way of thinking is this: For X↠Y, every X needs to have every possible combination of Y and other attributes (R–X–Y)
  - Essentially the rows for a given X must be a cartesian product!
  - If teacher Jonas occurs with one book/autor, it must occur with all book/author combinations for that course
  - This is what makes (book,author) independent from teacher

Fourth normal form (4F):

- For a relation R to be in **fourth normal form**:
  - R must be in **BCNF**
  - For all non-trivial MVDs X ↠ Y on R:
    X is a **superkey** of R
- If X↠Y and **X is not a superkey**, we say X↠Y is a **4NF violation**
- **To normalize:**
  - **Find a violation** X ↠ Y and break R into:
    - $R_1(X \cup Y)$ (”**every attribute in the MVD**”)
    - $R_2(R - Y)$ (”**Left-hand side and every attribute NOT in the MVD**”)
    - Then normalize both R_1 and R_2

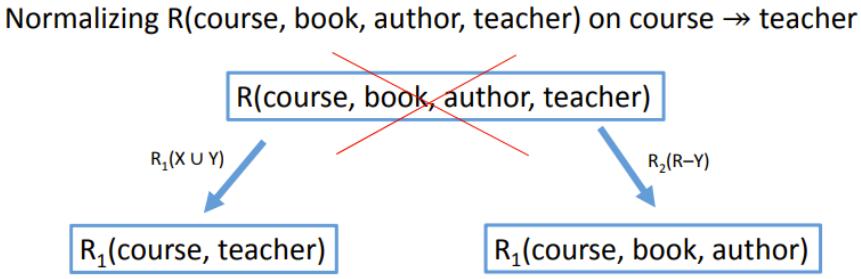


Table: Courses

course	book	author	teacher
Databases	DTCB	Ullman	Jonas
Databases	DTCB	Ullman	Aarne
Databases	DTCB	Widom	Jonas
Databases	DTCB	Widom	Aarne
Reglerteknik	RTB 1	AuthorX	TeacherX
Reglerteknik	RTB 2	AuthorX	TeacherX

Table: R_1 (a.k.a. CourseTeacher)

course	teacher
Databases	Jonas
Databases	Aarne
Reglerteknik	TeacherX

Table: R2 (a.k.a. CourseBooks)

course	book	author
Databases	DTCB	Ullman
Databases	DTCB	Widom
Reglerteknik	RTB 1	AuthorX
Reglerteknik	RTB 2	AuthorX

6 FD: FD vs ER

- FDs can find some things that ER cannot find
- ER can find a lot of things that FDs cannot find
    - Most many-to-many relationships cannot be expressed using FDs
    - Sentences like ”students can register for courses” do not express any FDs (but possibly some MVDs?)
- The two approaches **complement each other**, and **confirm each other** (or sometimes contradict each other which may indicate a problem)

Practical use of FDs combined with ER:

- FDs can be used to verify the correctness of an ER-design.
    - Is the result in BCNF with regards to the dependencies toy have identified?
    - Are the primary keys you identified sensible from your FDs?
    - If not, there may be an error in your ER-translation or your understanding/modelling of the domain.
- Sometimes FDs can be used to patch things up in your ER-design, particularly they are useful for finding secondary keys (UNIQUE constraints).
    - Every (minimal) key of each relation should be either the primary key or unique.

Finding functional dependencies:

- Determine all attributes
- Discover FD's either by looking at each attribute and ask, *"what do I need to know to determine this?"* or by looking at each fact in the domain description and asking, *"does this express a dependency?"*
- You can find multiple FDs determining the same attribute

Triggers and functions

1 Triggers and functions

- There are a few things high-level design cannot express:
  - Events
    - Example from domain: "when a student unregisters, the first student from the waiting list should be moved …"
    - ER describes what a database can contain, not operations on it
    - The design can only make sure to accommodate both the state before the event and after the event, but not the event itself
  - Advanced cross-table constraints
    - Example from domain "A student can only register if they have passed all prerequisites"
    - Cannot be expressed using reference/unique/value constraints
- These things are handled by **triggers**

Atomicity:

- All SQL modifications are **atomic**: If I execute a DELETE/UPDATE/INSERT and there is an **error**, **no rows will have changed**
  - For instance, for a delete, if just one matching row cannot be deleted, nothing gets deleted
  - **Intermediate changes are never visible to other users** of the database (rows do not disappear and then reappear again, they either change or they don't)
  - This says something about how intricate a DBMS is, imagine implementing this on a data structure in Java! (A method that removes all members satisfying a criteria, but rolls back all changes if there is an error)
  - Works even if the server loses power in the middle of an update(!) (The update will either be performed completely, or nothing is changed)

Cascading:

- Remember: deletes/updates **may fail due to references from other tables**
  - I cannot remove a student unless I first remove all that students' grades
- By default, a query that attempts to delete a referenced row fails and nothing is deleted, this can be changed when creating the reference.
- Delete all referencing row as well:

```
student TEXT REFERENCES Students(id) ON DELETE CASCADE
```

Can potentially lead to deleting the whole database

- Silently leave the referenced rows (possibly deleting other rows, which aren't referenced):

```
student TEXT REFERENCES Students(id) ON DELETE RESTRICT
```

Deliberately makes DELETE-operations slightly non-atomic (changing stuff even though there are errors)

Modes for ON DELETE/UPDATE (General syntax):

- **ON [DELETE/UPDATE] [CASCADE/RESTRICT/SET NULL]**
  - **ON DELETE CASCADE:** Delete this row if the referenced value is deleted.
  - **ON UPDATE CASCADE:** Update this value if the referenced value is updated
  - **ON UPDATE/DELETE RESTRICT:** "Silently" prevent deletes/updates to referenced value (do not raise an error).
  - **SET NULL:** Set this value to NULL if the referenced value is updated/deleted
- **ON UPDATE** is usually OK to **CASCADE**, for **ON DELETE**, be more careful!
- How to identify **ON DELETE/ON UPDATE:**
  - If you need to do something like *"When a student is deleted, it should automatically be unregistered from all courses"*
    - Note how this is clearly not something that we can model in ER

Assertions:

- Assertions are part of the SQL standard
- They allow us to **write conditions that should be globally true for the database**

Syntax:

```
CREATE ASSERTION <assertion name> AS
  CHECK <condition>;
```

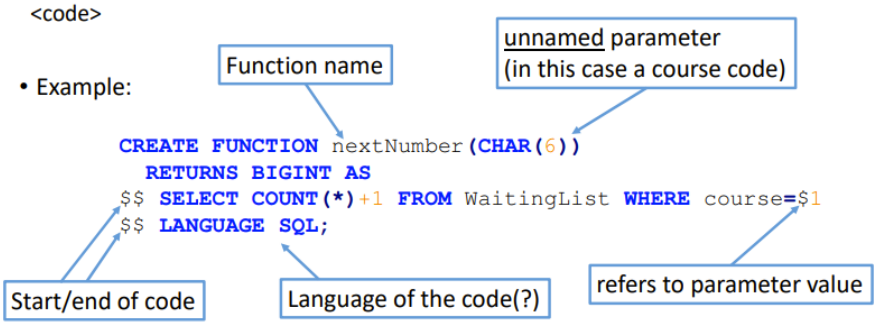
- Very difficult to implement efficiently in a DBMS
- For instance: We can write an assertion that states that all course registrations have happened within the last year from today
  - When should this be checked? What happens when it is suddenly false?
- Assertions are **not implemented in Postgres**, or in most major DBMS

User created functions (and syntax):

- Functions are stored in the DB server and executed in queries etc.
  - Sometimes called "stored procedures"
- Functions in Postgres can be written in different languages
- We will look at two of them:
  - SQL – you know this!
  - PL/pgSQL – new stuff!
    - Postgres version of Oracles PL/SQL (Procedural Language/SQL)
    - The language is procedural in the sense that (unlike SQL) programs are written as sequences of instructions
    - Similar to general purpose languages (like C, Java) in theory, similar to SQL in syntax

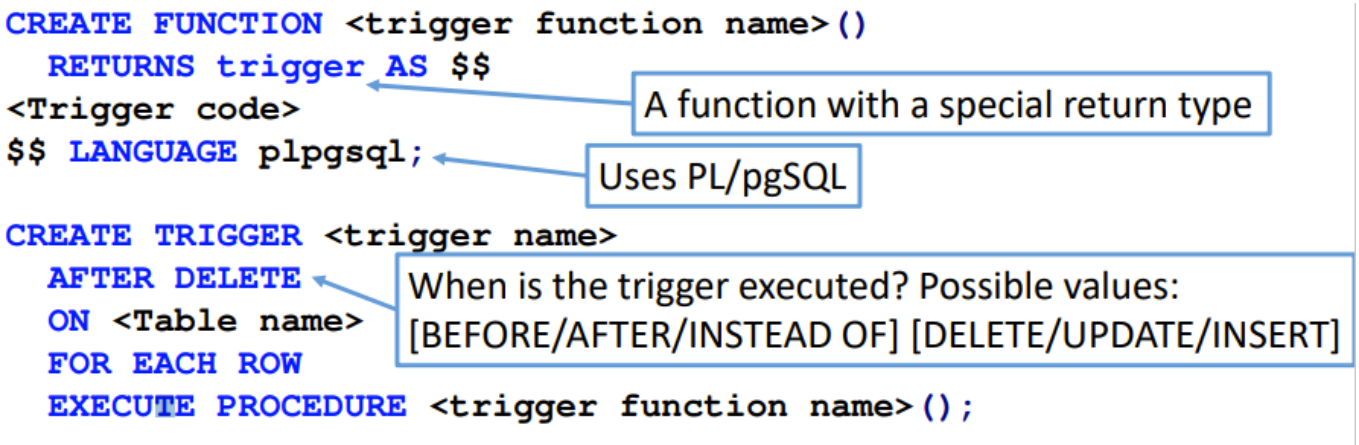
```
CREATE FUNCTION <name>(<parameter types>) RETURNS <return type> AS
 <code>
```

• Example:



Triggers:

- Triggers are procedures (functions) stored on the server, executing when certain actions are taken (like updating, inserting or deleting from a table)



- Triggers are useful when:
  - Modelling events:
    - Something is supposed to happen when a user takes a certain action
    - Like in the assignment: When a student is unregistered from a course, another student from the waiting list may take its place.
  - Cross-row or cross-table constraints:
    - Like a more powerful check constraint, ensure invariants across tables that are more complicated than uniqueness/reference constraint.
    - Work like assertions but we specify when they should be checked.

Triggers and errors:

- Updates/deletes that execute triggers are still **atomic**
  - If there is an error for any row, nothing is changed
  - This is true even if the **AFTER** keyword is use, if a trigger I called AFTER INSERT and the trigger function raises an error, everything including the insert will be rolled back.
- Triggers can raise **errors:**

```
RAISE EXCEPTION '<error message>';
```

Variables:

- Like local variables in Java and similar languages
- Declared in a single DECLARE block at the start of the code (after $$) and ended with ;.
- **SELECT … INTO** is used to run a query and store the result in a variable (declared earlier).
- Either **raises an error or gives null/first row if the query does not give exactly one row**, depending on DBMS and setting.
  - Note: Aggregates such as MAX, COUNT, MIN, AVG, SUM,… **always give one row**.

```
$$
DECLARE
  cnt INT;
  myBool BOOLEAN;
BEGIN
...
```

Declares two variables

```
SELECT MAX(credits) INTO creds
  FROM Courses WHERE code=x;
```

IF-statements (conditionals):

- Syntax:

```
IF (<condition>) THEN
   ...
ELSIF (<condition> THEN
   ...
ELSE
   ...
END IF;
```

2 Triggers on Views

- Views are amazing for providing a useful interface for applications
- We can select from "tables" that seem to have lots of redundancy, but actually they just reflect data redundancy-free tables
    - Example: the PassedCourses view that contains credits for each grade (in a proper table this would violate BCNF!)
- Unfortunately, we can SELECT from views, but not INSERT or DELETE
- Triggers changes that!
    - E.g. by writing a trigger INSTEAD OF INSERT ON, we can do INSERT INTO VALUES(…) to execute the trigger.

Triggers on tables or Views:

- One nice way to think about it: INSERT/UPDATE/DELETE on tables are our internal operations that need to be used very carefully (like private methods)
- Operations on views are our exported interface, and all operations are safe by design, there is no way to corrupt the database by inserting into our views
- When designing the triggers on views we modify the tables directly, but never from applications that use them

INSTEAD OF INSERT on Views:

- The **NEW** variable **contains the values for an inserted row**
    - Uses the **columns and types in the view**, not in any underlying tables
- Will not automatically insert anything anywhere, the trigger will have to execute INSERT on the underlying tables for anything to happen
    - The row that was added may not show up when selecting from the view
- Should **always return NEW** unless there is an error

```
CREATE FUNCTION Insert_function() RETURNS trigger AS $$
BEGIN
    -- Code goes here
    RETURN NEW;
END; $$ LANGUAGE plpgsql;

CREATE TRIGGER Insert_trigger
  INSTEAD OF INSERT ON <view name goes here>
  FOR EACH ROW EXECUTE PROCEDURE Insert_function();
```

INSTEAD OF DELETE on Views:

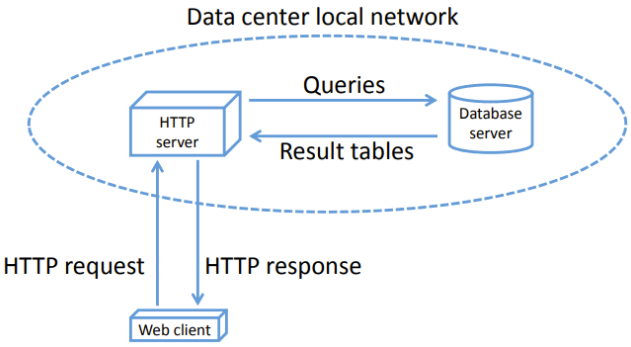
- The **OLD** variable **contains a row from the view that matches the WHERE clause** of an executed DELETE-query.
- Note: Will report "X rows deleted", which really means the trigger was executed for X rows in the view, the database may not have been changed at all.
- Should **always return OLD**

```
CREATE FUNCTION Delete_function() RETURNS trigger AS $$
BEGIN
   -- Code goes here
   RETURN OLD;
END; $$ LANGUAGE plpgsql;

CREATE TRIGGER Delete_trigger
  INSTEAD OF DELETE ON <view name goes here>
  FOR EACH ROW EXECUTE PROCEDURE Delete_function();
```

JDBC, Database security

1 JDBC & Infrastructure

Infrastructure:



- The web-server software communicates with the database server
- Impossible to connect directly to the database from the internet

JDBC:

- JDBC (Java Database Connectivity) is a library for connecting to databases from the Java programming language.
- Using JDBC step by step:
    1. Load the Postgres driver (or another DBMS driver)
    2. Initiate a single **Connection** object, by providing server URL, username etc.
    3. Create one or more **Statement** or **PreparedStatement** objects from the connection (each represents a 'channel' for executing a query or a statement)
    4. Executing queries through statements give a **ResultSet** object (represents a query result, that can be iterated row by row)
- Each of these objects is a resource that needs to be closed after use
- Any of these steps can fail for various reasons, throwing a SQLException
    - Error handling is important
- Boilerplate code:

```
String DATABASE = "jdbc:postgresql://localhost/portal
Class.forName("org.postgresql.Driver");
Properties props = new Properties();
props.setProperty("user", "postgres");
props.setProperty("password", "postgres");

// Try-with-resource (requires Java 7) closes connection automatically
try (Connection conn = DriverManager.getConnection(DATABASE, props)) {
    <actual code goes here>
} catch (SQLException e) {
    System.err.println(e);
    System.exit(2);
}
```

Load the driver

Open the connection

Deal with (unexpected) errors here

SQL-injection attack:

- Always use PreparedStatement to avoid SQL-injection attacks.
- Called **sanitizing your inputs.**

NoSQL Databases

1 Document databases & Semi structured data (SSD)

Semi structured Data (SSD)

- The relational model has a very rich structure
  - Allows us to have strong constraints on data
- This structure also limits flexibility
  - Much of the design work is centered on deliberately preventing users from being flexible (by enforcing constraints)
- In semi-structured data models, the schema is flexible
  - Data is still structured … but the structure is not necessarily uniform across the data
  - E.g. data does not fit in tables where every row has the same columns
- Examples of **document-based SSD**: **XML** and **JSON**
  - Both these are **document based**, a data set is most naturally described by a **text document rather than a table**
  - Both are **hierarchical**, the documents have a **tree-structure**
- XML and JSON are often used as **Data interchange formats:**
  - Data interchange formats facilitate the transfer of data from one database to another
  - Transforms data from one schema to another, via an intermediate format

2 XML

- Derived from pre-existing document markup languages
  - Compare with HTML: HTML uses tags to format a webpage, XML uses **tags to describe data**
- Documents are built from:
  - **Elements**
  - **Attributes**
  - **Text**

Hierarchical structure of XML:

- XML documents always have a **single root element**, that in turn may contain other elements with attributes/elements of their own etc.
- All **tags must be closed** and **properly nested**.
- It is valid to have **text and sub elements in the same element**, however it is considered **bad practice.**

Attributes vs elements:

- Two ways of representing a person in XML:

```
<Teacher>
     <Firstname>Jonas</Firstname>
     <Lastname>Duregård</Lastname>
     <Course>Databases</Course>
</Teacher>
```

Element-centric

Attribute-centric

```
<Teacher firstname="Jonas" lastname="Duregård" course="Databases"/>
```

- Advantages of **attributes**:
  - Compact syntax
  - Correspond naturally to attributes in relational databases
- Advantages of **elements**:
  - Can represent complex objects (with attributes, subelements etc.)
  - Can have arbitrarily many elements with the same tag
  - Easily extensible (remember we are using XML for flexibility!)
- Compare with ER-modelling: Anything that needs to have attributes of its own can never be an attribute
- Often, **elements are used to represent the actual data**, while **attributes are used to describe "modifiers" of tags**.

3 JSON

- Think of JSON documents as JavaScript objects without any methods
  - **Objects that can have variables** (that are objects or primitive types)
  - The **"variable names" are called keys** in JSON
  - The top level of a document can be **anything**, i.e., an object, an array, or just a primitive value.

Full syntax of JSON:

- Every JSON value is built from a combination of six types:
  - Structures:
    - **Objects**: { "key1" : JSON , "key2" : JSON , ...}
      Can have 0 or more **key:value** pairs, values can be **any JSON value**
    - **Arrays/lists**: [JSON , JSON , ...]
      Can have 0 or more items, each can be **any JSON value**
  - Literals (usually the "leaves" in the document tree):
    - **Strings**: "Hello world!\n"
    - **Numbers**: 7, 5.3
    - **Booleans**: true or false
    - **null**

An example document

```
[{"city":"Boston", "population":700000}
,{"city": "New York",
  "boroughs": [
          "The Bronx",
          "Brooklyn",
          "Manhattan",
          "Queens",
          "Staten Island"
        ]
 }
]
```

- An array containg two objects
- First object has two keys:
  city (string) and population (number)
- Second object has two keys:
  city (string) and boroughs (array)
- The array in boroughs contains five strings

- Arrays in JSON are **heterogenous**, they can contain a mix of types**:**

```
[1, "string", {"key":0}, true, [1,2]]
```

Examples of valid and non-valid JSON documents:

Which are well formed JSON documents?

[[[1]]]

• Yes, this is an array containing an array containing an array containing 1

{{"city" : "Gothenburg"}}

• No! The outer object is not on the form key:value

{"city" : "population" : 3000}

• No! This is on the form x:y:z which is never allowed in objects

{"city":{"boston"}}

• No! The object {"boston"} is not valid (contents should be key:value)

[{}, [], ""]

• Yes, it's an array containing the empty object, array and string

4 Building JSON using Postgres

JSON support in Postgres:

- **Two SQL types**:
  - **JSON** (text format)
  - **JSONB** (faster binary format)
- You can have JSON documents in table cells, views, query results, …
- Functions for **building JSON** documents
- Operators for **extracting values from JSON documents**
  (e.g. turn a JSON string to an SQL string to use it in a WHERE-clause)

```
CREATE TABLE Posts(
  id SERIAL PRIMARY KEY,
  author TEXT NOT NULL REFERENCES Users(uname),
  created TIMESTAMP NOT NULL,
  content JSONB NOT NULL
);
```

The only new stuff on this slide!

Inserting JSON data:

```
INSERT INTO Posts VALUES (
  DEFAULT,
  'Jonas',
  CURRENT_TIMESTAMP,
  '{"link" : "https://xkcd.com/327/", "preview":true}' :: JSONB
);
```

Sometimes, we need to use `::TYPE` to convert values to different SQL types

Basic querying with JSON:

```
SELECT * FROM Posts;

 id | author |          created          |                                content
----+--------+---------------------------+-----------------------------------------------------------------
  1 | Jonas  | 2019-11-28 15:01:31.247316 | {"link": "https://xkcd.com/327/", "preview": true}
  2 | Jonas  | 2019-11-28 15:01:31.252545 | {"prop": {"size": 15434}, "picture": "funnycat.gif"}

SELECT id, content->'link' AS url FROM Posts;
 id |           url
----+-------------------------
  1 | "https://xkcd.com/327/"
  2 |
```

Gives content.link as a JSONB value

This is a JSON string (not an SQL string)

This is a (SQL) null value (not a JSON null value!)

Using JSON in where clauses:

Find all picture posts:

```
SELECT id, content FROM Posts
  WHERE content->'picture' IS NOT NULL;
```

```
 id |                          content
----+-----------------------------------------------------
  2 | {"prop": {"size": 15434}, "picture": "funnycat.gif"}
```

As a shorthand, Postgres allows: `WHERE content ? 'picture';`

Even more JSON querying:

Two ways of finding all posts with enabled previews

Compare to the JSON true value

```
SELECT * FROM Posts
  WHERE (content->'preview') = 'true';
```

Convert the JSON boolean to an SQL boolean

```
SELECT * FROM Posts
  WHERE (content->'preview') :: BOOLEAN ;
```

Nested access (selecting property of element in another element):

Select the size property of all posts

```
SELECT id, content, content->'prop'->'size' AS postsize
 FROM Posts;
```

```
 id |                          content                          | postsize
----+-----------------------------------------------------------+----------
  1 | {"link": "https://xkcd.com/327/", "preview": true}        |
  2 | {"prop": {"size": 15434}, "picture": "funnycat.gif"} | 15434
```

A tiny JSON document (type JSONB), not an SQL number

Combining JSON features and SQL features:

```
SELECT
  id,
  COALESCE(content->'prop'->'size','0') :: NUMERIC AS postsize
 FROM Posts;
```

```
 id | postsize
----+----------
  1 |        0
  2 |    15434
```

SQL numbers

We can create a view from this, aggregate with SUM to compute total size for each user etc.

Building JSON from query results:

- Even if you have no tables with JSON values, you can still use JSON in postgres
    - We can run **queries that construct JSON documents from table data**

Building objects: jsonb_build_object(…):

- A built-in stored procedure (function) for **creating JSON objects**:

jsonb_build_object(key1,value1,key2,value2,…)

Example (three rows in Posts):

```
SELECT id, jsonb_build_object(
                'user',author,
                'postid',id
                ) AS jsondata
  FROM Posts;
```

Selects id, and also a little JSON object for each row in Post

```
 id |            jsondata
----+--------------------------------
  1 | {"user": "Jonas", "postid": 1}
  2 | {"user": "Jonas", "postid": 2}
  3 | {"user": "sanoJ", "postid": 3}
```

Table: Posts

id	author	…
1	Jonas	…
2	Jonas	…
3	sanoJ	…

Strings, numbers etc. are automatically converted from SQL to JSON

Aggregating into JSON arrays: jsonb_agg(…):

- jsonb_agg is an aggregation function (like SUM, COUNT, …) • Must either be the only thing selected, or have a GROUP BY

Simple example, build an array with all post authors (3 rows in Posts):

```
SELECT jsonb_agg(author) AS jsonarray
  FROM Posts;
```

Groups together all rows and takes the author value from each row in the group

```
           jsonarray
-----------------------------
 ["Jonas", "Jonas", "sanoJ"]
(1 row)
```

Always gives a single row (without GROUP BY)

Large example:

Going nuts with correlated queries

This beautiful query creates a JSON object for each user, containing their basic information and an array of posts

```
SELECT
  json_build_object(
    'uid', uname,
    'email', email,
    'posts', (SELECT COALESCE(jsonb_agg(jsonb_build_object(
                'postid',id,
                'time',created)
              ),'[]') FROM Posts WHERE author = U.uname)
  )
  FROM Users U;
```

A single item in the SELECT, building a massive JSON object for each row in Users

Refers to outer query

jsonb_agg gives (SQL) null for empty sets, so coalesce to []

One row from the result of the query


```
{"uid" : "Jonas",
 "email" : "jonas.duregard@chalmers.se",
 "posts" :
   [{"time": "2019-12-02T14:52:37.451796",
     "postid": 1}
   ,{"time": "2019-12-02T14:52:37.453225",
     "postid": 2}
   ]
}
```


A number in an object in an array in an object ☺
This is at position x.posts[1].postid
Tells us that one of the posts for user Jonas has ID 2

JSON validation & JSON document querying

1 JSON Validation

JSON Schema:

- A JSON schema is **either a root schema or a subschema**, with a **root schema being the top-level schema**, and a **subschema a schema that is within the root schema**.
- A JSON schema is itself a JSON object.
- We use **"keywords" as keys**, and the **value** for each keyword **tells us something about the schema.**
- We use these keywords to define the schema.
- The **empty object `{}` and true validates against anything**, i.e. you don't provide any information about what it should contain. A schema that says **`false` is always invalid**, no matter what.
- Example:

If we have the following schema, that says every branch has a name and a program:

```
{"type": "object", "title": "Branch",
 "properties": {"name": {"type": "string"},
                "program": {"type": "string"}},
 "required": ["name", "program"]}
```

The following are valid:

```
{"name": "IT", "program": "IE"}
{"name": "MPALG", "program": "CS", "numStudents": 20}
```

But the following are invalid:

```
{"name": "IT"}
{"name": "IT", "program": 5}
```

Schema keywords:

✓ **title** and **description** are annotations that are used to identify the schema in question but are not used for validation.

```
Schema: {"title": "Character",
         "description": "A Lord of the Rings character"}
Valid: everything
Invalid: nothing
But it's a kind of documentation for the users
```

✓ **type** is used to define the type of the JSON within, and can be any of array, boolean, integer, null, number, object, or string.

```
Schema: {"type": "number"}
Valid:  1
        2
        5.9
        6.022e+10
        …
Invalid: "a"
         true
         {"as": "hey"}
         ["a", "b"]
         …
```

✓ **enum** is used to enforce that a field should be any of specifc values.

```
Schema: {"type": "string","enum": ["u", "3", "4","5"]}
Valid:  "u"
        "3"
        "4"
        "5"
Invalid: 3
         4
         "uu"
         …
```

✓ **const** is a special case of **enum** that allows exactly on value (a constant).

```
Schema: {"const":42}
Valid: 42    Invalid: everything else
```

✓ **minimum** and **maximum** are **specific to numbers** and specify the minimum and maximum value that the number can take.

```
Schema: {"type": "integer","minimum": 1,"maximum": 6}
Valid: 1
       2
       3
       4
       5
       6
Invalid: 0
         7
         100
         "asd"
         {"number": 5}
         …
```

✓ **minLength** and **maxLength** are **specific to strings** and specify the minimum and maximum length of the string.

```
Schema: {"type": "string","minLength": 10,"maxLength": 10}
Valid: "abde284320"
       "1234567890"
       …
Invalid: "123"
         "1asd"
         25
         {"idnr": "1234567890"}
         …
```

- ✓ **properties** is used to define the schema for the properties of an object.

```
Schema: {"type": "object",
         "properties": {"name": {"type": "string"},
                        "age": {"type":"integer"}}}
Valid: {"name": "Matti", "age": 27}
       {"name": "Jonas"}
       {"name": "Frodo", "age": 50, "location": "Shire"}
       …
Invalid: {"name": 11, "age": 12}
         {"age": "23"}
         "1234"
         …
```

- ✓ **additionalProperties** is used to define the schema for any **properties** not present in properties. Can be used to enforce that the **properties** in properties are the only properties present.

```
Schema: {"type": "object",
         "properties": {"name": {"type": "string"}},
         "additionalProperties": false}}
Valid: {"name": "Jonas"}
       {"name": "Matti"}
       …
Invalid: {"name": 11, "age": 12}
         {"age": "23"}
         {"name": "Matti", "age": 27}
         {"name": "Frodo", "age": 50, "location": "Shire"}
         "1234"
         …
```

- ✓ **required** is used to define what properties a certain object must have.

```
Schema: {"type": "object", "required": ["name", "age"]}
Valid: {"name": "Matti", "age": 27}
       {"name": "Sauron", "age": "Not known"}
       {"name": 11, "age": "twelve", "favFood": "eggos"}
       …
Invalid: {"name": "Matti"}
         {"age": 2}
         "asda"
         …
```

✓ **minProperties** and **maxProperties** is used to define the maximum and minimum number of properties an object must have.

```
Schema: {"type": "object", "minProperties": 1, "maxProperties": 2}
Valid: {"name": "Matti", "age": 27}
       {"name": 9}
       {"lottery": [7,9,13,17], "winner": "Jonas" }
       …
Invalid: {}
         {"name": "McCartney", "age": 76, "band": "The Beatles"}
         "asdad"
         …
```

✓ **items** allow you to specify a schema for the items in the array.

```
Schema: {"type": "array", "uniqueItems": true}
Valid: [1,2,3]
       ["a", "b", "c"]
       [1]
       []
       …
Invalid: [1,1]
         ["a", "b", "a"]
         "asdf"
         1234
         …
```

✓ **uniqueItems** specifies that the items must be unique (i.e., no duplicates).

```
Schema: {"type": "array", "items": {"type": "number"}}
Valid: [1,2,3]
       [42,5,7e10]
       [323.8,2,1]
       …
Invalid: ["asd", "one"]
         [1,2,3,"four"]
         "asdf"
         24
         …
```

✓ **minItems** and **maxItems** specify the minimum and maximum number of items in the array.

```
Schema: {"type": "array", "minItems": 3, "maxItems": 5}
Valid: [1,2,3]
       ["a","q","e","t"]
       [8,4,2,9,0]
       …
Invalid: []
         [1,2,3,5,6,7]
         ["a","b"]
         "asdf"
         …
```

✓ **contains** allows you to specify a schema that at least one item in the array bust satisfy

```
Schema: {"type": "array", "contains": {"const": 42}}
Valid: [1,2,3,42]
       [42]
       ["a", 42, "b", "c", 42]
       …
Invalid: []
         [1, 2, 4]
         [[42]]
         {"contents": [12,2,3]}
         42
         "42"
         …
```

- **Meta-keywords**:
  - ✓ Subschemas can be combined using boolean logic operators: **allOf** (**all-of**), **anyOf** (**any-of**), **oneOf** (**one-of**), and **not**

```
Schema: {"title": "Grade",
         "oneOf": [{"type": "integer", "maximum": 5},
                   {"type": "integer", "minimum":3 }]}
Valid: -5
        2
       -15
        100
        …
Invalid: 3
         4
         5
         "asdf"
         5.8
         …
```

✓ **$ref** is a keyword you can use to refer and reuse schemas.
is used to refer to the schema itself!

Schema:

```
{"type": "object",
 "title": "A Non-empty linked list",
 "required": ["value", "next"],
 "properties": {
  "value": {"type": "integer"},
  "next": {"oneOf":[{"type": "null"},
                    {"$ref": "#"}]}}}
```

Valid: `{"value": 1, "next": {"value":2, "next": null}}`
`{"value": 1, "next": null}`
…

Invalid: `{"value": 2}`
`{"next": {"value": 2, "next":null}}`
`23, [1,2], "asdasd", …`

✓ **definitions** is used to define schemas to use with **$ref**

```
{"definitions": {"posInt": {"type": "integer","minimum":1}},
 "type": "array",
 "items": {"$ref": "#/definitions/posInt"}}
```

Valid: `[1,2,3]`
`[1]`
`[]`
`[1000,12]`
…

Invalid: `[-1]`
`[0]`
`[0,1,2]`
`5`
`"asd"`
…

Filesystem example:

```
{"name": "/", "contents": [
  {"name": "file1", "filetype": "txt", "size": 100},
  {"name": "a/", "contents": [
   {"name": "file2", "filetype": "jpg", "size": 200},
   {"name": "file3", "filetype": "mp4", "size": 600},
   {"name": "file4", "filetype": "png", "size": 300}]},
  {"name": "b/", "contents": [
   {"name": "c/", "contents": [
    {"name": "file5", "filetype": "jpg", "size": 400}]}]}]}
```

```
{"title": "Filesystem",
 "$ref": "#/definitions/directory",
 "definitions": {
   "file": {
     "type": "object",
     "properties": {
       "name": {"type": "string", "minLength": 1},
       "filetype": {"type": "string"},
       "size": {"type": "integer"}},
     "required": ["name", "size"]},
   "directory": {
     "type": "object",
     "properties": {
       "name": {"type": "string","minLength": 1},
       "contents": {"type":"array",
                    "items": {"oneOf": [
                                {"$ref": "#/definitions/file"},
                                {"$ref": "#/definitions/directory"}]}}},
     "required": ["name","contents"]}}}
```

2 Querying JSON Documents

The JSON Path Language:

- The schema guarantees a specific structure and we can now **query it**.
- **Language for queries: JSONPath**
- Defined at:
  https://www.postgresql.org/docs/12/functions-json.html,
  in this course we use the **'strict'** mode to avoid confusion.

How to use JSON Path in Postgres:

- Using the **jsonb_path_query**, we can use JSON Path expressions to **query json documents and get all resulting JSON items as Postgres rows**.
- Using **jsonb_path_query_array** does the same, except the **results are wrapped into a single JSON array.**
- Using **jsonb_path_query_first** returns **only the first result**.

```
WITH JsonEx AS (SELECT
'<json>':: jsonb AS val)
SELECT
jsonb_path_query_array(val,
 'strict <query>'
) FROM JsonEx;
```

JSONPath operators:

- The following examples use this JSON object for querying:

```
/file1.txt
/a/file2.jpg
/a/file3.mp4
/a/file4.png
/b/c/file5.jpg
```

```
{"name": "/", "contents": [
  {"name": "file1", "filetype": "txt", "size": 100},
  {"name": "a/", "contents": [
   {"name": "file2", "filetype": "jpg", "size": 200},
   {"name": "file3", "filetype": "mp4", "size": 600},
   {"name": "file4", "filetype": "png", "size": 300}]},
  {"name": "b/", "contents": [
   {"name": "c/", "contents": [
    {"name": "file5", "filetype": "jpg", "size": 400}]}]}]}
```

- **The operators:**
  - ✓ '$' is the root object, which we usually start our expression with.

```
SELECT jsonb_path_query_array(val,'strict $') FROM JsonEx
[{"name": "/", "contents": […]}]
```

  - ✓ '.' Is the child operator, used to access a property of an object.

```
'strict $.name'
["/"]
```

  - ✓ '[ ]' is the subscript operator, which is used to access elements in arrays or objects, or iterate over them.

```
'strict $.contents[1].contents[0].name'
["file2"]
```

```
'strict $.contents[2].contents[0].contents[0].size'
[400]
```

  - ✓ '*' is the wildcard operator, which returns **everything in the current object** or **each entry in an array [*]**.

```
'strict $.*'
["/", [{"name": "file1", "filetype": "txt", size: 100}, {"name": "a/", …},
 {"name": "b/",…}]]

'strict $.contents[1].*'
["a/", [{"name": "file2",…}, {"name": "file3",…}, {"name": "file4",…}]]

'lax $.contents[1].*[0]'
["a/", {"name": "file2", "filetype": "jpg", "size": 200}]
```

  - ✓ '**' is the recursive descent operator, which goes into all the children of the element, and then into all children of that element, and so on…

```
'strict $.**.name'
["/", "file1", "a/", "file2", "file3", "file4", "b/","c/","file5"]

'strict $.contents[1].**.name'
["a/",  "file2", "file3", "file4"]
```

- ✓ '**@**' is used to refer to the current element in expressions.
- ✓ '**?(<expression>)**' allows you to apply a filter expression.

```
'strict $.**?(@.filetype == "jpg").size'
[200, 400]

'strict $.**?(@.size < 300).name'
["file1","file2"]
```

Example of a full query:

- Don't forget to **explicitly typecast**, since the values are returned as **jsonb values**!
- This example-query will:
  - *Use JSON path to get the **sum of the prices of hamburgers** on the menu*

```
WITH JsonEx AS (SELECT
'<json>':: jsonb AS val)
SELECT SUM(query.value :: int)
FROM (SELECT jsonb_path_query( target: val,
 path: 'strict.$[*]?(@.category == "Burgers").**.price')
 AS value
FROM JsonEx) as query;
```

- From:

```
[{"category":"Starters",
  "contents":[
    {"dish":"Calamari", "price":8.50}]},
 {"category":"Salads",
  "contents":[
    {"dish":"Caesar",  "price":8.50},
    {"dish":"Chicken", "price":9.25}]},
 {"category":"Burgers",
  "contents":[
    {"dish":"Standard", "price":9},
    {"dish":"Bacon",    "price":10},
    {"category":"Vegetarian Burgers",
     "contents":[
       {"dish":"Haloumi",  "price":13},
       {"dish":"Mushroom", "price":10}]}]}]
```

- Result: **9 + 10 + 13 + 10 = 42**

Transactions & Authorization

1 Authorization & Privileges

- A database has:
  - Privileges on schema elements (Tables, View, Triggers, etc.)
  - **Nine different privileges**
  - Unlimited levels of access – each user can be given different access

Granting privileges (syntax):

Syntax:

```
GRANT <privilege-list> ON <object> TO <user-list>
```

Examples:

```
GRANT SELECT
 ON Inventory
 TO webshop_user, marketing;
```

Allows two users to SELECT from Inventory

```
GRANT SELECT, DELETE, INSERT
 ON Registrations
 TO study_administrator;
```

Allows one user to SELECT, DELETE and INSERT on a table (or view?)

Basic Table privileges:

- SELECT [(<list of column names>)]
- DELETE
- INSERT [(<list of column names>)]
- UPDATE [(<list of column names>)]
  - The square brackets '[' and ']' indicate optional syntax and are not part of the SQL-syntax.
    - UPDATE(idnr,name) is a valid privilege, and so is simply UPDATE
    - Adding the column list limits the privilege to updating those columns

Other, less frequently used, privileges:

- EXECUTE
  - Allows users to execute a function/procedure
- REFERENCES [(<list of column names>)]
  - Allows users to create foreign keys
  - Rarely granted, since foreign keys are usually part of the design
- TRIGGER [(<list of column names>)]
  - Allows users to create triggers
  - Rarely granted for the same reason as above

Revoking/Giving other users the privilege to grant other users privileges:

- Add "WITH GRANT OPTION" to the end of a GRANT statement.
- **Privilege to drop table cannot be given**
  - Only the owners of the schema element can do that.
- You can replace the privilege list with "ALL PRIVILEGES" to **grant all privileges**.
- Previously granted **privileges can be revoked**:
  - REVOKE *<list of privileges>*
    ON *<schema element>*
    FROM *<list of users>*

2 Transactions

A basic transaction example:

- This SQL code runs a sequence of inserts in a transaction
- If any INSERT fails, the whole transaction is rolled back
- The end result is either 8 successful inserts, or 0

```
BEGIN;
INSERT INTO Taken VALUES(4444444444,'CCC111','5');
INSERT INTO Taken VALUES(4444444444,'CCC222','5');
INSERT INTO Taken VALUES(4444444444,'CCC333','5');
INSERT INTO Taken VALUES(4444444444,'CCC444','5');
INSERT INTO Taken VALUES(1111111111,'CCC111','3');
INSERT INTO Taken VALUES(1111111111,'CCC222','3');
INSERT INTO Taken VALUES(1111111111,'CCC333','3');
INSERT INTO Taken VALUES(1111111111,'CCC444','3');
COMMIT;
```

Starts a transaction (Postgres syntax)

End the transaction and commit all changes to database

Explicitly rolling back:

- It is possible to issue the "**ROLLBACK;"**-statement any time during a transaction.
- Immediately ends the transaction and reverts all changes.
- Good use of rollback:
    - **BEGIN** *transaction*
      Do some modifications
      IF (*everything good?*)
      COMMIT
      ELSE
      ROLLBACK

Every query is a transaction

- Single queries and modifications are atomic
- If a **SQL query is executed outside a transaction**, a transaction is **automatically started and committed after the query is successful**.

Serializability:

- Two processes/transactions **run in serial if one ends before the other starts**
- They are **serializable** if the **give the same result as if they were run in serial**
    - Basically, you won't get any race-conditions conflicts when ran at the same time.

ACID Transactions:

- A DBMS is expected to support *"ACID Transactions"* which are:
    - **Atomic**: Either the whole transaction is run, or nothing
        - *Adding BEGIN/COMMIT*
    - **Consistent:** Database constraints are preserved
        - *Dealt with under the hood by the DBMS*
    - **Isolated:** Different transactions may not interact with each other
        - This one is a bit tricky, see below…
    - **Durable:** Effects of a transaction are not lost in case of a system crash
        - *Dealt with under the hood, by the DBMS*

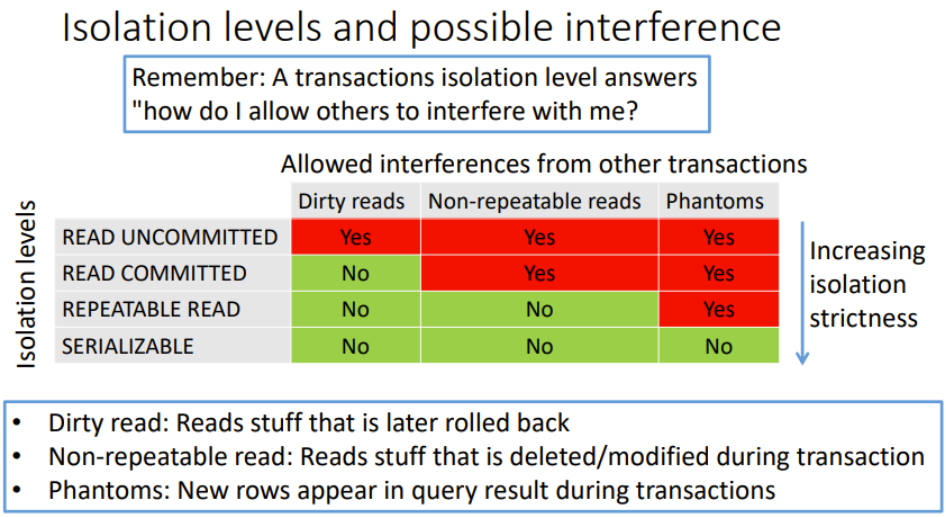
The isolations problem:

- Need to avoid race-conditions
    - Just run everything in serial?
        - NO! It can get very slow, especially when waiting for user inputs…
- Solutions: **Isolations levels!**

Transaction isolation levels and interference:

- Each transaction can choose an isolation level
- The isolation level decides **how other processes are allowed to interfere**
- Standard SQL defines **four transaction isolation levels**:
    1. **READ UNCOMMITTED**
    2. **READ COMMITTED**
    3. **REPEATABLE READ**
    4. **SERIALIZABLE**
- Isolation levels are defined by which of three common kinds of interference are allowed:
    1. ***Dirty reads***
        - Transaction T1 modifies/creates a data item
        - Transaction T2 then reads that data item **while T1 is still running.**
        - T1 then performs a **ROLLBACK,** T2 has **read a data item that was never committed**, and so **never really existed**.
            - Violation of the **atomicity of T1.**
            - **T2 performs the dirty read**, to prevent it **T2 need to change its isolation level.**
    2. ***Non-repeatable reads***
        - Transaction T1 reads a data item.
        - Another transaction **T2 then modifies or deletes that data item** and commits.
        - If **T1 then attempts to re-read (or modify) the data item**, it receives a **modified value or discovers that the data item has been deleted**.
            - This violates the **atomicity of T1**; it can observe outside changes.
            - **T1 performs the non-repeatable read**, to prevent it **T1 needs to change its isolation level**.
    3. ***Phantoms***
        - Transaction T1 performs a SELECT and gets a number oof rows.
        - Transaction T2 performs **at least one INSERT** and commits.
        - T1 **repeats the SELECT** and gets the **same rows as before, but also some additional rows.**
            - **Different from a non-repeatable read**, where rows could also have **disappeared or changed**.
            - This violates the **atomicity of T1.**
            - **T1 has the phantom problem** and can prevent it by **changing isolation level**.

Isolation levels and possible interference (CHART):



Summary of isolation levels:

- When I start a transaction using isolation level…
  - **… serializable**, I accept no interference at all.
  - **… repeatable read**, I accept that others make changes as long as they don't modify or delete data that I have already looked at.
  - **… read committed**, I accept that other make any changes to the database.
  - **… read uncommitted**, I accept that some of the data I get might actually never even be in the database.
- When should I use…
  - **Read uncommitted?**
    - When you only care about never waiting for other transactions, maximal performance.
      - ❖ You are not worried about reading incorrect values now and then.
      - ❖ You just want a transaction that can be rolled back.
  - **Read committed?**
    - When you are performing several queries, but the later do not really depend on the results of the first still being accurate.
      - ❖ They must have been accurate at some point (no dirty reads).
      - ❖ The "insert lots of grades at once"-example could be Read committed.
    - Basically, whenever a query encounters a row that has been modified by another still active transaction, it will wait for that transaction to commit or roll back, then proceed based on the outcome.
      - ❖ Possibly harmful for transactions that take long time
  - **Repeatable read?**
    - When you run several queries and the later depend on the data from the former to still be accurate, but not on it being *complete.*
    - An example: Read some rows from a table, then run a separate query for some of those rows
      - ❖ It's not important that none of the rows we read have been modified or deleted
      - ❖ It's not important that no new rows have been added, even if they would be in our initial selection.

- **Serializable?**
  - When you are running multiple queries and it's essential that no one else modifies the parts of the database you are using in any way while the transaction is running.
    - ❖ You are not too worried about locking others out of the database

Warning about long transactions:

- If you have more complex transaction with delays after modifications, other modifications may have to wait a long time
- This can cause deadlocks where your whole database is locked waiting for some commit/rollback that may never even happen
- Also makes your application vulnerable to denial of service attacks, where an attacker deliberately causes your database to deadlock
  - Even long *read uncommitted* transactions are problematic.
    - Remember that *read uncommitted* means others can interfere with you, but your interference may still cause others to lock.

Transactions in JDBC:

- To use transactions, run `conn.setAutoCommit(false);`
  - Where conn is your Connection object
  - Only done once for the connection
- End each transaction with `conn.commit();` or `conn.rollback();`
  - This also starts a new transaction
- To set isolation level, run something like:

```
conn.setTransactionIsolation(
                    Connection.TRANSACTION_REPEATABLE_READ);
```

  - This must be done at the start of the transaction (e.g. after last commit)
- If the connection closes without committing, the transaction is rolled back

Relational Algebra

1 Relational algebra

- **Algebra is a set of values**, and a **collection of operations** on those values.
- Formulas built from those operations (and constants) are called **expressions**

Relational algebra:

- Relational algebra is an algebra on the **infinite set of relations**, with operations like Cartesian product, union, etc.
- Relational algebra **expressions are essentially queries** (but not in SQL).
- Just like arithmetic and Booleans, this algebra is **closed** under its operations
    - If I apply addition to two numbers, I get a number
    - If I apply AND to two Booleans I get a Boolean
    - If I apply Cartesian product to two relations I get a relation
- There are at least two advantages to using Relational Algebra over SQL:
    - **Reasoning**: We can use hundreds of years of mathematical results and methods to prove that our queries do what we intend for them to do
    - **Simplification**: Similarly to how we can simplify (a+b*0+a) to (2*a), we can sometimes simplify complicated relational algebra expressions
        - Uses proven simplification rules
        - Can be used to make queries faster

What is a relation?

- Good enough **informal definition** for relational algebra:
    - ***Relations are tables***.
- Slightly more **formal**:
    - ***A relation is a schema (relation name + attribute list) and a collection of tuples, such that all tuples match the schema.***
        - Typically we abstract away the tuples, focusing on the structure/schema of the relation when writing relational algebra expressions.

2: Relational operators

OPERATOR: Projection (π):

- The π (pi) operator corresponds to the select clause in SQL.
- Called the **projection operator**, we **project a certain view of the relation**.

Syntax: $\pi_{\text{<attribute list>}}(R)$, where R is any relational algebra expression

In SQL: `SELECT <attribute list> FROM (<SQL for R>);`

Example: $\pi_{id,name}(Students)$

In SQL: `SELECT id,name FROM Students;`

Sets bags or lists? (Again)

- Remember:
  - A **Set** has no duplicates or internal ordering
  - **Bags** allow duplicates, still no internal ordering
  - **Lists** allow duplicates and each value has a position
- Traditionally, relations are considered sets of tuples in relational algebra
  - This makes them harder to translate to/from SQL where results are bags
- In this course **we use bag semantics**

Projection on Sets/Bags:

- Projection is one of the operators where set/bag semantics differ.
- If using set semantics, the number of tuples/rows may decrease, because duplicates are introduced when removing the attributes!
- One way to explain this in terms of SQL:
  - With bag semantics, projection corresponds to the **SELECT** clause
  - With set semantics, projection corresponds to **SELECT DISTINCT**
- **WE USE BAG SEMANTICS, THUS ALLOWING DULPICATES**

Table: WL

student	course	position
Student1	TDA357	1
Student2	TDA357	2
Student1	TDA143	1

→ $\pi_{student}(WL)$ →

set semantics
student
Student1
Student2

bag semantics
student
Student1
Student2
Student1

OPERATOR: Selection (σ):

- The σ (sigma) operator corresponds to the WHERE-clause in SQL.

Syntax: $\sigma_{\text{<condition on rows>}}(R)$

In SQL:

```
SELECT * FROM <SQL for R> WHERE <condition on rows>
```

- Conditions should be simple row-wise checks, do not put RA-expressions in your conditions (unlike in SQL where subqueries are allowed).
  - Boolean syntax from SQL **(AND, OR, NOT …)** or logical symbols **(∧, ∨, ¬ …)**
  - Comparisons like **<, >, = …** on constants and attributes
- Called the **selection operator** because it **selects which rows to keep**.
- NOTE: Selection (σ) does not correspond to the SELECT clause in SQL!

$\pi_{student}(Grades)$	$\sigma_{student=1}(Grades)$
⇕	⇕
SELECT student FROM Grades	SELECT * FROM Grades WHERE student=1

Base relations/Tables

- Base relations like Students in $\pi_{id,name}(Students)$ are part of the algebra
  - In one way they are like constants: The schema of the relations are known
  - In one way they are like variables: The tuples in the relations are unknown
  - Intuitively they are like created tables in SQL, not considering INSERTS
- Base relations in expressions are **simply table names in SQL**

OPERATOR: Cartesian product (×):

- The relational algebra syntax for Cartesian product is **R1 × R2**

In SQL: `SELECT * FROM <SQL for R1>,<SQL for R2>`

We can now join relations:

$\sigma_{<\text{join condition}>}(T1 \times T2)$

Equivalent SQL:

```
SELECT * FROM T1, T2 WHERE <join condition>;
```

Compositional expressions, monolithic queries

Consider this SQL query and an equivalent relational algebra expression:

```
SELECT name, credits FROM Students, Grades
  WHERE idnr = student AND Grade >= 3
```

$\pi_{name,credits}(\sigma_{idnr=student\ AND\ grade\ >=\ 3}(Students \times Grades))$

- The **SQL code is a single query** performing projection, selection and Cartesian product, whereas the **expression does each of those in separate steps**
  - This is a fundamental difference of RA and SQL
  - In RA each subexpression results in a relation, SQL "does everything at once" and gets a single result.

OPERATORS: Other Set operations:

- Just like in SQL, we have the three set operations:
  - Union: R1 U R2
  - Intersection: R1 ∩ R2
  - Difference/subtraction: R1 - R2
- Example:
  Idnr of all students that have not passed any courses:

$$\pi_{idnr}(Student) - \pi_{student}(\sigma_{grade>=3}(Grades))$$

- "Take all idnr from students, and remove all idnr with a passing grade"
- Like in SQL, schemas must be compatible (**same number of attributes**)

Extending Set operations to Bags:

- In sets, each tuple is either in or not in each relation
- In **bags**, **each tuple occurs a number of times in each relation**
- Assuming **x** occurs **n times in R1** and **m times in R2**
  - x occurs **n+m times in R1 U R2**
  - x occurs **min(n,m) times in R1 ∩ R2**
  - x occurs **n-m times in R1 - R2** (minimal of 0 times)
- Translates to UNION ALL, INTERSECT ALL and EXCEPT ALL
- This is the semantics we use for union, intersection and difference in this course

OPERATOR: Grouping (γ):

- The grouping operator γ (gamma) is like a combined SELECT and GROUP BY

Syntax: $\gamma_{<attributes/aggregates>}(R)$

Example: $\gamma_{student,\ AVG(grade) \rightarrow average}(Grades)$

Table: Grades

student	course	grade
S1	TDA357	3
S2	TDA357	3
S1	TDA143	5

student	average
S1	4
S2	3

In SQL:

```
SELECT student, AVG(grade) AS average
    FROM Grades GROUP BY student;
```

- Automatically groups by and projects all attributes in the subscript
- The **arrow indicates naming** (required for all aggregates)
- Result has exactly one attribute for each attribute/aggregate

Example:

- *Select the name of all students that have passed at least 2 courses*

```
Students(idnr, name)
Grades(student, course, grade)
  student -> Students.idnr
```

$$\pi_{name}(\sigma_{passed>=2}(\gamma_{student,\ name,\ COUNT(*)\rightarrow passed}(\sigma_{grade>=3\ AND\ idnr=student}(Students \times Grades))))$$

- **Describing the expression "*bottom up*" (right to left):**
  1. Take the cartesian product of students and grades
  2. Select the rows with passing grades and matching id-numbers
  3. Group what remains by student and calculate the number of passed
  4. Select the rows with at least to passed
  5. Project the name attribute
- **Sanity check! (You are only SELECTING FROM EXISTING ATTRIBUTES)**

$\pi_{name}(\sigma_{passed>=2\ AND\ idnr=student\ AND\ grade>=3}(Students \times \gamma_{student,\ COUNT(*)\rightarrow passed}(Grades)))$

Can not use grade here!

(idnr,name,student,passed)

- Not doing this simple sanity check is probably the most common way to unnecessarily loose points on the exam

Qualified names:

```
Students(idnr, name)
Grades(student, course, grade)
  student -> Students.idnr
```

- If there are name clashes, it makes sense to sanity check with qualified names

$\pi_{name}(\sigma_{Student.idnr=Grades.idnr\ AND\ average>4}(Students \times \gamma_{idnr,\ AVG(grade)\rightarrow average}(Grades)))$

(Grades.idnr, average)

(Students.idnr, Students.name, Grades.idnr, average)

- Note that the attribute average does not have any qualified name

OPERATOR: Renaming (ρ)

- The ρ (rho) operator renames the result of an expression.

Syntax: $\rho_{<new\ schema>}(R)$

Example $\rho_{S(idnr,studentname)}(Students)$

Renames both the relation (for qualified names) and attributes

Students

idnr	name
1	Jonas
2	Emilia
3	Emil

S

idnr	studentname
1	Jonas
2	Emilia
3	Emil

Use $\rho_S(Students)$ to only rename the relation and keep attribute names

- **A renaming example:**

```
SELECT N1.num, N2.num, N1.owner
 FROM Numbers AS N1, Numbers AS N2
 WHERE N1.owner = N2.owner;
```

Here the ρ operator is essential

$\pi_{N1.num,\ N2.num,\ N1.owner}(\sigma(_{N1.owner\ =\ N2.owner}(\rho_{N1}(Numbers) \times \rho_{N2}(Numbers))))$

Sanity check: *(N1.owner, N1.num, N2.owner, N2.num)*

OPERATOR: Join (⋈):

- Like in SQL, there is a special join operator: $R1 \bowtie_{<condition>} R2$
- Purely a **convenience operator**, we can define it using:
  - $R1 \bowtie_c R2 = \sigma_c (R1 \times R2)$

Expression layout:

- When writing relational algebra expressions on paper, it is convenient to start each operator on it's own row.
  - It's often a good idea to start in the middle of the paper with a join, then add operators above it
  - You can easily extend conditions with an extra AND etc…

$$\pi_{name}(\sigma_{passed>=2}(Students \bowtie_{idnr=student} \gamma_{student,\ COUNT(*)\rightarrow passed}(\sigma_{grade>=3}(Grades))))$$

Splitting up expressions (for readability):

- You can break out and name parts of your expressions for readability:

$$R1 = \gamma_{student,\ COUNT(*)\rightarrow passed}(\sigma_{grade>=3}(Grades))$$
$$R2 = (Students \bowtie_{idnr=student} R1$$
$$Result = \pi_{name}\ (\sigma_{passed>=2}(R2))$$

- Especially useful for sanity checking! What attributes do R1 and R2 have?
- **NOTE:** The names (R1 and R2) **cannot be used as qualified names** (i.e. R1.attribute), unless you use ρ.

Expression trees:

The best way to understand an expression in <u>any</u> algebra, is as a syntax tree

Each node in the tree can be computed into a value (or a schema), bottom up

ALL OPERATORS: All basic operators (and some more advanced ones):

- Basic:

- Selection, "Sigma": $\sigma_{<\text{selection condition}>}(R)$
- Projection, "Pi": $\pi_{<\text{attribute list}>}(R)$
- Cartesian product: $R1 \times R2$
- Other set operations: $R1 \cup R2$, $R1 \cap R2$, $R1 - R2$
- Grouping, "Gamma": $\gamma_{<\text{attributes/aggregates}>}(R)$
- Join: $R1 \bowtie_{<\text{condition}>} R2$
- Renaming, "Rho": $\rho_{<\text{Relation name}>(<\text{optional attribute names}>)}(R)$

- Advanced (to match features of SQL):

- NATURAL JOIN: $R1 \bowtie R2$ (Just omit the Join-condition)
- JOIN USING: $R1 \bowtie_{idnr} R2$ (replace Join-condition with attribute)
- Outer joins:
  - Full outer join: $R1 \bowtie^{o}_{<\text{join condition}>} R2$
  - Left/right join: $R1 \bowtie^{oL}_{<\text{join condition}>} R2$ and $R1 \bowtie^{oR}_{<\text{join condition}>} R2$
- DISTINCT: δ (delta), for converting from a bag to a set
  e.g. R1 U R2 is UNION ALL in SQL, δ (R1 U R2) is UNION
- τ (tau), for ORDER BY on an expression. Examples:
  $\tau_{grade}(Grades)$ for SELECT * FROM Grades ORDER BY grade ASC
  $\tau_{-grade}(Grades)$ for SELECT * FROM Grades ORDER BY grade DESC

EXAMPLE: Translating a single query:

- A query with almost everything:

```
SELECT a1, MAX(a2) AS mx
  FROM T1, T2
  WHERE a3=5
  GROUP BY a1,a3
  HAVING COUNT(*) > 10
  ORDER BY a1 ASC;
```

Some things, like HAVING requires new names to be introduced

- A relational algebra expression for it:

$\tau_{a1}(\pi_{a1,mx}(\sigma_{temp>10}(\gamma_{a1,a3,MAX(a2)\rightarrow mx,COUNT(*)\rightarrow temp}(\sigma_{a3=5}(T1 \times T2)))))$

- The sanity check is even more important when "blindly" translating

EXAMPLE: Translating correlated queries:

- Consider a query like

Correlation: subquery refers to outer query

```
SELECT name FROM Students AS S
   WHERE 4<(SELECT AVG(grade) FROM Grades WHERE student=S.idnr);
```

- This is very easy to mistranslate (if you don't sanity check!)
- The correlation needs to be replaced with a join:

$\pi_{name}(\sigma_{4<average}(\gamma_{student,\ AVG(grade)\ \rightarrow\ average}(Grades \bowtie_{idnr=student} Students)))$

EXAMPLE: NOT IN and NOT EXISTS:

- Set subtraction can often (always?) be used to replace NOT IN
- Example: Select students that have no grades

```
SELECT idnr,name FROM Students
  WHERE idnr NOT IN (SELECT student FROM Grades);
```

- In relational algebra (one of many possible solutions):

$R1 = \rho_{NoGrades(s)}(\pi_{idnr}(Students) - \pi_{student}(Grades))$

$Result = \pi_{idnr,name}(Students \bowtie_{s=idnr} R1)$

- Use set subtraction to get the ID of all students without grades, then join back with Students to recover names
  (uses renaming to avoid having two Students.idnr for the join)