TDA357/DIT621 – Databases

Lecture 1B to 3 – Tables, Relations, SQL, More SQL, Even more SQL

Jonas Duregård

Gold star if you can spot the Stanley Kubrick reference

Relational database

- Simple and familiar data model
- The database is a collection of tables
- Each table has columns and rows
- Example: Tiny database for a school
- Cross referencing: What grade did Bart get in Programmerade System?
  - Answer: 3
- The underlined column names are called *primary keys*, each row must have unique values for these columns

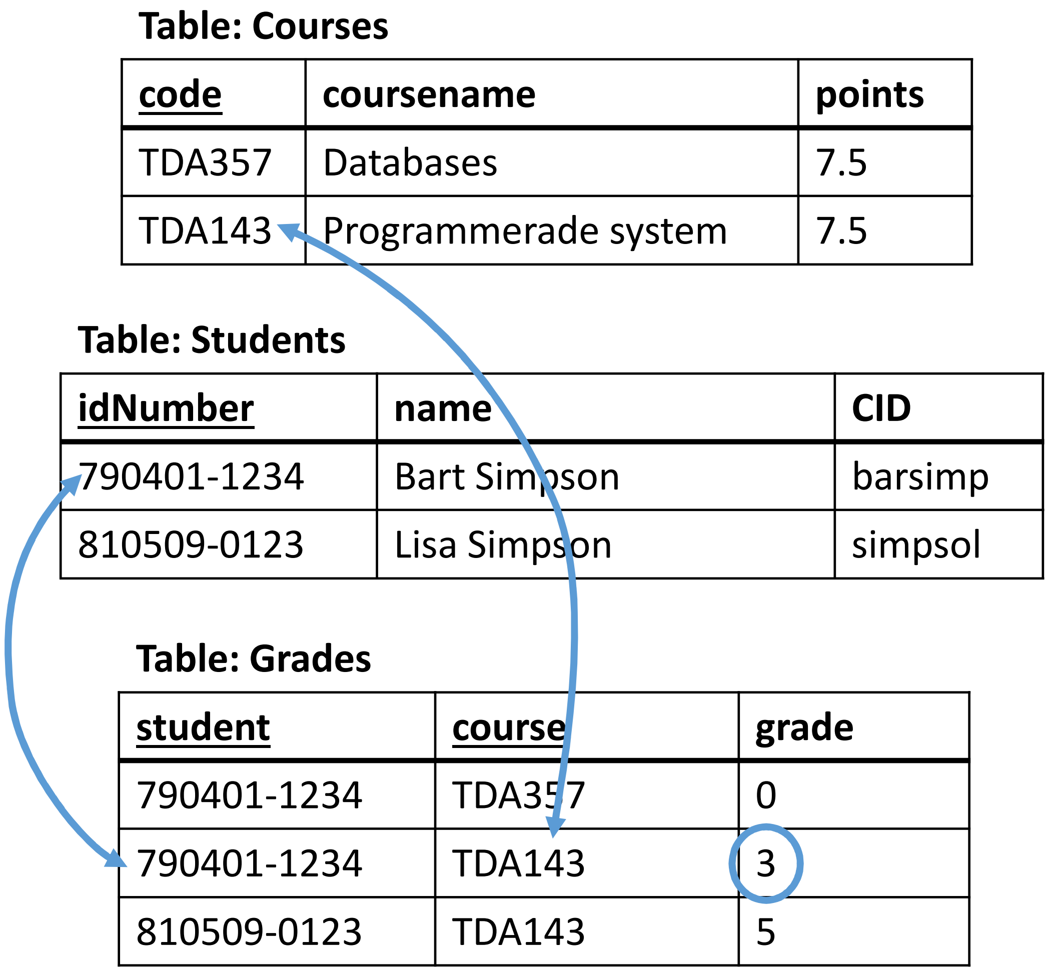


Table: Courses

code	coursename	points
TDA357	Databases	7.5
TDA143	Programmerade system	7.5

Table: Students

idNumber	name	CID
790401-1234	Bart Simpson	barsimp
810509-0123	Lisa Simpson	simpsol

Table: Grades

student	course	grade
790401-1234	TDA357	0
790401-1234	TDA143	3
810509-0123	TDA143	5

The concept of relations

- A mathematical relation is a set of fixed length tuples (a,b,c,...)
- Example: the mathematical operator < (less than) is a relation on pairs of numbers where e.g. the tuple (3,9) is included, but not (9,3)
- A table is basically a relation, with some extra information like column names
- Relations give a simple but powerful theoretical basis for databases

Table

Table: Grades

student	**course**	**grade**
790401-1234	TDA357	0
790401-1234	TDA143	3
810509-0123	TDA143	5

Mathematical relation

{ (790401-1234, TDA357, 0)
, (790401-1234, TDA143, 3)
, (810509-0123, TDA143, 5)
}

Constraints

- A constraints is a limitation on what values you can put in a table
- Some constraints we may have:
  - 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)
- If we have to strong constraints, we can not model all the data we want (e.g. a student can only have a single grade in a single course)
- If we have to weak constraints, we can accidentally model unintended data (e.g. a student having multiple grades in the same course)

SQL

SQL Basics

- SQL ("sequel"), Structured Query Language allows you to do lots of things, including:
  - Create tables
  - Insert or modify values in tables
  - Query tables for data

Easy

Trivial

Kinda' tricky…

"Hello World" in SQL

I'm running these commands in psql, the postgres REPL-interface

Create a table with two text columns

```
postgres=# CREATE TABLE Words (word1 TEXT, word2 TEXT);
CREATE TABLE

postgres=# INSERT INTO Words VALUES ('Hello', 'World!');
INSERT 0 1

postgres=# SELECT * FROM Words;
 word1 | word2
-------+--------
 Hello | World!
(1 rad)
```

Insert a single row in the new table

Query for the whole contents of the table

Look, it's a little table!

Check out the postgres tips on the course webpage to try this yourself

Case convention

- SQL is completely case insensitive (except in text values)
- We will use case in the following way to make code readable:
  - UPPERCASE marks keywords of the SQL language.
  - lowercase marks the name of an attribute.
  - Capitalized marks the name of a table.
- These queries do the same thing, but only the first follows our convention:

```
☺ SELECT attribute FROM Data WHERE attribute2 = something;

😐 select attribute from data where attribute2 = something;

☹ select attRibutE from Data wheRe attribUte2 = soMething;
```

PostgreSQL

- Chalmers postgresql server (check Fire for your credentials):

  ```
  psql –h ate.ita.chalmers.se –U <username>
  ```

- I advice you all to install postgres on your own machines
- Access local postgresql installation:

  ```
  psql –U <username> <dbname>
  ```

- Semicolon and postgres prompt:
  - postgres=> select 1+1
  - postgres-> ;

You always have to end queries with ;
Otherwise they continue on the next input line

Did I tell you to check out the postgres tips on the course webpage?

A good postgres workflow

- Create a file called whatever.sql
- Open the file in your favourite editor and write your SQL code there
  - Start the file with `DROP OWNED BY <username> CASCADE;` to automatically delete everything every time the file is executed
- Run `\i whatever.sql` in psql (run psql in the right directory) to execute the file and enjoy all the error messages
- DO NOT WRITE 500 LINES OF SQL AND THEN RUN IT!
  - This is the #1 rookie mistake
  - Work incrementally: Write one query and re-run the file until it works without errors, then start writing the next one

SQL: CREATE TABLE

- The subset of SQL that deals with creating tables is called the Data Definition Language, SQL DDL
- The basic syntax is:

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

- Where every table element is either:
  - a column
  - a constraint

Types

- Basic table elements (columns) consist of a name and a type.
  - Like **`courseCode CHAR(6)`** or **`salary INT`**
- Most common 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

Types in different databases

- Unfortunately types are poorly standardized between different DBMS
- Which types are available differ a lot
- Technical details of common types also differ

- For PostgreSQL, this link covers the available datatypes: https://www.postgresql.org/docs/10/static/datatype.html

Primary key constraints

- Every table should have a single primary key constraint

```
CREATE TABLE Students (
   idNumber TEXT,
   name TEXT,
   cid CHAR(7),
   PRIMARY KEY (idNumber)
);
```

idNumber is the primary key of Students

```
CREATE TABLE Grades (
  student TEXT,
  course CHAR(6),
  grade INT,
  PRIMARY KEY (student, course)
)
```

student and course together form the key of grades (called a *compound key*)

Reference constraints

- What is the problem here?
  - A student that does not exist has a grade
  - More technical: one of the values in the student column of Grades does not exist in the idNumber column of Students

Table: Students

idNumber	name	CID
790401-1234	Bart Simpson	barsimp
810509-0123	Lisa Simpson	simpsol

Table: Grades

student	course	grade
790401-1234	TDA143	0
790401-1234	TDA357	3
424242-4242	TDA143	4

This row should not be allowed!

FOREIGN KEYs, reference constraints in SQL

```
INSERT INTO Grades VALUES ('424242-4242', 'TDA357', 5);
ERROR: insert or update on table "grades" violates
foreign key constraint "grades_student_fkey"
Key (student)=(424242-4242) not present in "students".
```

```
CREATE TABLE Students (
   idNumber TEXT,
   name TEXT,
   cid CHAR(7),
   PRIMARY KEY (idNumber)
);

CREATE TABLE Grades (
  student TEXT,
  course CHAR(6),
  grade INT,
  PRIMARY KEY (student, course),
  FOREIGN KEY (student) REFERENCES Students(idNumber)
);
```

"student must exist in the idNumber column of Students"

Here, student is both part of the primary key, and a foreign key

Multiple foreign keys

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

```
CREATE TABLE Grades (
  student TEXT,
  course CHAR(6),
  grade INT,
  PRIMARY KEY (student, course),
  FOREIGN KEY (student) REFERENCES Students(idNumber),
  FOREIGN KEY (course) REFERENCES Courses(courseCode)
);
```

Informally: "student must be an actual student, course must be an actual course"

Compound references

A player has a player-number and belongs to a team (within each team, players have unique numbers)

```sql
CREATE TABLE Player (
  pname TEXT,
  team TEXT,
  pnumber INT,
  PRIMARY KEY (team,pnumber)
);
```

A penalty can be given to a player
Constraint: player and team together identify an existing player

```sql
CREATE TABLE Penalties (
  incidentTime TIMESTAMP,
  player INT,
  team TEXT,
  PRIMARY KEY (incidentTime, player, team),
  FOREIGN KEY (player,team) REFERENCES Player(pnumber,team)
);
```

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

```
CREATE TABLE Students (
    idNumber TEXT,
    name TEXT,
    cid CHAR(7),
    UNIQUE (cid),
    PRIMARY KEY (idNumber)
);
```

cid (username) must have unique values for each student

Works (no collisions)

```
INSERT INTO Students VALUES (111111-1111,'Jonas','jonasdu');
INSERT INTO Students VALUES (222222-2222,'Evil Jonas','jonasdu');
```

ERROR: duplicate key value violates unique constraint "students_cid_key"
Key (cid)=(jonasdu) already exists.

Value constraints

- Value constraints are the simplest type of constraints

```
CREATE TABLE Player (
  team TEXT,
  pnumber INT,
  CHECK (pnumber > 0 AND pnumber < 100),
  PRIMARY KEY (team,pnumber)
);
```

```
INSERT INTO Player VALUES ('Team Edward',666);
ERROR: new row for relation "player" violates check constraint
"player_pnumber_check"
Failing row contains (Team Edward, 666).
```

More value constraints

```
CREATE TABLE Grades (
  student TEXT,
  course CHAR(6),
  grade INT,
  CHECK (grade IN (0,3,4,5)),
  PRIMARY KEY (student, course),
  FOREIGN KEY (student) REFERENCES Students(idNumber)
);
```

The value for grade is one of the listed values

```
CREATE TABLE products (
    product_no INTEGER,
    name text,
    price NUMERIC,
    CHECK (price > 0),
    discounted_price NUMERIC,
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

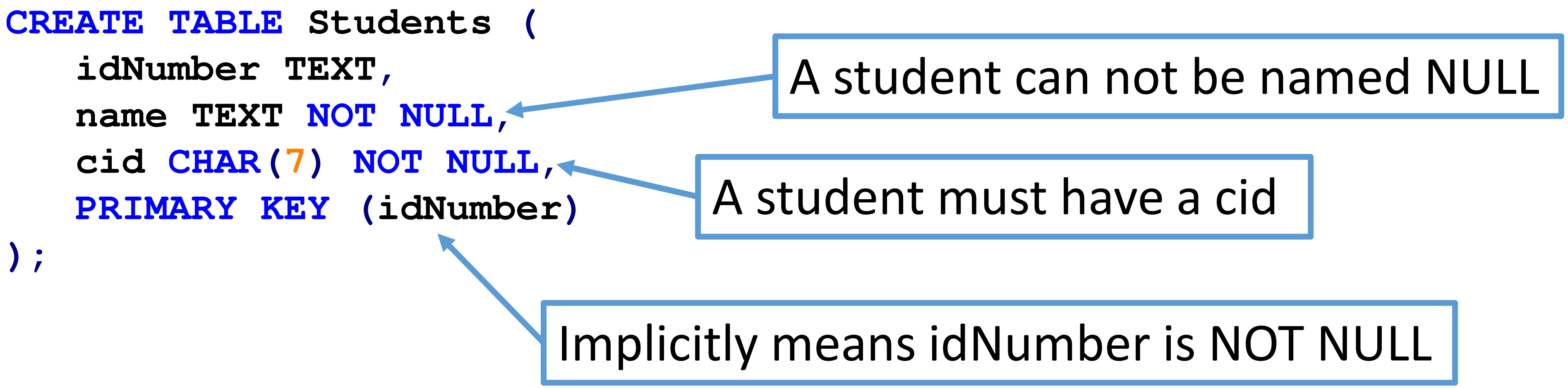
Compares two attributes

Things you can NOT do with value constraints

- Anything that checks outside the values in the row being inserted,
  - E.g. you can not have a check that makes sure the grade in course B is never higher than the grade for the same student in course A

NOT NULL constraint

- The NOT NULL constraint is a special constraint that says that a column can not 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



```
CREATE TABLE Students (
   idNumber TEXT,
   name TEXT NOT NULL,
   cid CHAR(7) NOT NULL,
   PRIMARY KEY (idNumber)
);
```

A student can not be named NULL

A student must have a cid

Implicitly means idNumber is NOT NULL

Naming a constraint

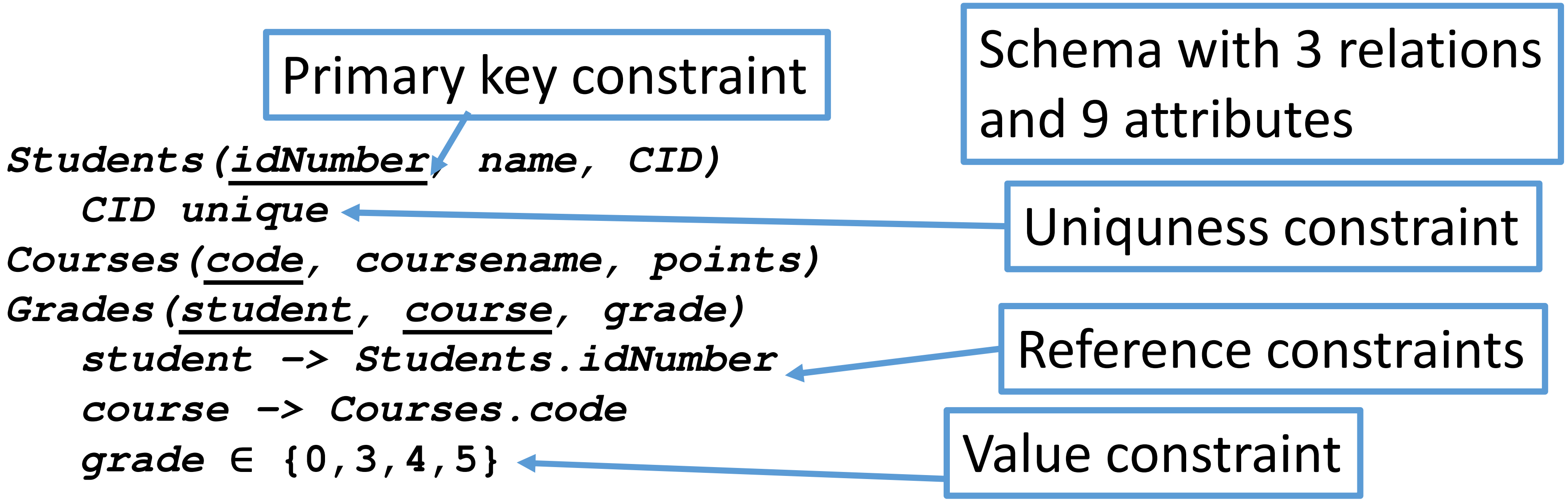
- Not that necessary in Postgres, because it actually does a decent job of automatically naming constraints
- Syntax: CONSTRAINT name_in_one_word <actual constraint>

```
CREATE TABLE Grades (
  student TEXT,
  course CHAR(6),
  grade INT DEFAULT 0,
  CONSTRAINT valid_grade CHECK (grade IN (0,3,4,5)),
  PRIMARY KEY (student, course)
);
```

```
INSERT INTO Grades VALUES ('Jonas','TDA357',6)
ERROR:  new row for relation "grades" violates check constraint "valid_grade"
Failing row contains (Jonas, TDA357, 6).
```

Schemas

- Schemas are compact 'blueprints' of relations
- Like CREATE TABLE, but with less technical detail and informal syntax
- Format is
  `tablename(<attributes, primary key underlined>)`
  `   <list of additional constraints>`



```
Students(idNumber, name, CID)
   CID unique
Courses(code, coursename, points)
Grades(student, course, grade)
   student -> Students.idNumber
   course -> Courses.code
   grade ∈ {0,3,4,5}
```

Primary key constraint

Schema with 3 relations and 9 attributes

Uniquness constraint

Reference constraints

Value constraint

Translating schemas into SQL

- Each relation becomes a table
- Each attribute becomes a column, decide an appropriate type
- Underlined attributes are marked PRIMARY KEY
- References become foreign keys
- Unique/value constraints become UNIQUE/CHECK
- Add NOT NULL everywhere$^{\text{(for now)}}$

```
Grades(student, course, grade)
   student -> Students.idNumber
   course -> Courses.code
   grade ∈ {0,3,4,5}
```

```
CREATE TABLE Grades (
student TEXT,
course CHAR(6),
grade INT NOT NULL,
PRIMARY KEY (student, course),
FOREIGN KEY (student) REFERENCES Students(idNumber),
FOREIGN KEY (course) REFERENCES Courses(courseCode),
CHECK (grade IN (0,3,4,5))
);
```

Types must match referenced columns!

Start of lecture 2

Fire works now!

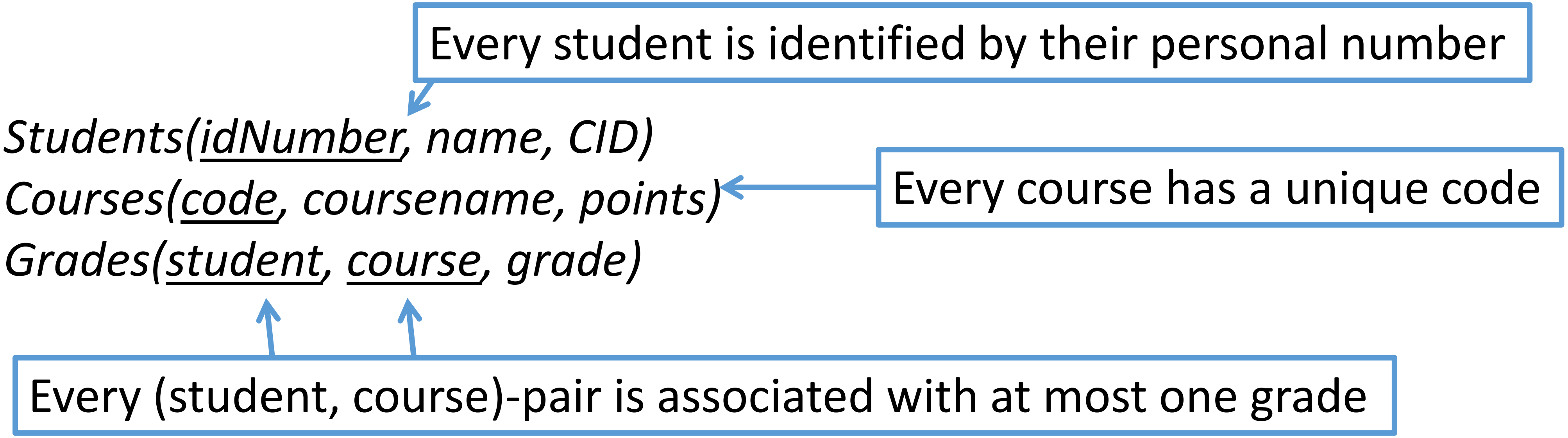
```
CREATE TABLE Grades (
 student TEXT,
 course CHAR(6),
 grade INT NOT NULL,
 PRIMARY KEY (student, course),
 FOREIGN KEY (student)
   REFERENCES Students(idNumber),
 FOREIGN KEY (course)
   REFERENCES Courses(courseCode),
 CHECK (grade IN (0,3,4,5))
);
```

The SQL saga so far:

- We can create tables with CREATE TABLE
  - A list of column/type-pairs and constraints
- Constraints prevent us from putting bad stuff in our database
  - Duplicate values (UNIQE/PRIMARY KEY)
  - Missing data in other tables (FOREIGN KEY a.k.a. references)
  - Generally bad inputs (CHECK a.k.a. value constraints)
- Jonas does not approve of NULL, so use NOT NULL everywhere
- Schemas are compact representations of table structures
  - Can be translated to SQL tables, just add types

The art of selecting primary keys

- Every relation has a set of attributes that together identify values
  - This set of attributes is called the primary key of the relation
  - Every row must have a pairwise unique value for these attributes
- Shown in the schema by underlining the attributes in the primary key
- Note that there is a single primary key, possibly with multiple attributes



Every student is identified by their personal number

Students(idNumber, name, CID)
Courses(code, coursename, points)
Grades(student, course, grade)

Every course has a unique code

Every (student, course)-pair is associated with at most one grade

Primary key problem 1

- What is the problem with this relation?
  Grades(student, course, grade)

Key collision!
(Two rows have the same key values)

student	course	grade
790401-1234	TDA357	0
790401-1234	TDA143	3
810509-0123	TDA143	5

- Consequence: Each student can have at most a single grade in a single course (or no grades at all) ☹

Primary key problem 2

- What is the problem with this relation?
  Grades(student, course, grade)

Key collision!
(Two rows have the same key values)

student	course	grade
790401-1234	TDA357	0
790401-1234	TDA143	3
810509-0123	TDA143	5

- Each course can only give at most one grade to a single student ☹

Primary key problem 3

- What is the problem with this relation?
  Grades(student, course, grade)

Works – no collisions

student	**course**	**grade**
790401-1234	TDA357	0
790401-1234	TDA143	3
810509-0123	TDA143	5

Also works

student	**course**	**grade**
790401-1234	TDA143	0
790401-1234	TDA143	3
790401-1234	TDA143	4

- A student can have multiple different grades in the same course ☹
  - But not several identical grades!
  - Something that should cause a key violation is accepted, we need a stronger constraint

Exactly the primary key we want

- *Grades(student, course, grade)*

Works!

student	**course**	**grade**
790401-1234	TDA357	0
790401-1234	TDA143	3
810509-0123	TDA143	5

student	**course**	**grade**
790401-1234	TDA143	0
790401-1234	TDA143	3
790401-1234	TDA143	4

Collision (which is a good thing!)

Short hands for common operations

- Inlined constraints:
  - PRIMARY KEY can be merged into a column definition:

    ```
    idNumber TEXT PRIMARY KEY,
    ```
  - REFERENCES can be merged into column definitions

    ```
    course CHAR(6) NOT NULL REFERENCES Courses(code)
    ```
- References to primary keys can omit the attribute list:

  ```
  course CHAR(6) NOT NULL REFERENCES Courses
  ```
- None of these short hands work for compound keys/references!

Default values

- You can add default values when creating columns:

```
lastName TEXT NOT NULL DEFAULT 'Doe',,
```

- In inserts you can write DEFAULT instead of any value (or omit it if it's last)
- Example: Make `incident` default to the time when the INSERT is executed

```
incident TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

- Defining a column with the type SERIAL gives it a default value that increases by one for each inserted row (postgres specific)

```
id SERIAL PRIMARY KEY,
```

  - Common way of introducing *synthetic keys*.
- The "default default value" for nullable columns is NULL 😐

Removing tables

- 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

Creating Views

General form:

CREATE VIEW <name> AS <query>;

Example:
CREATE VIEW Cheap AS SELECT name FROM Products WHERE price < 100;

A view is a way of giving a name to a query
Views can be used similarly to a table
(I can write SELECT … FROM Cheap … after creating the view above)

When the underlying table is changed, so is the view.

The first Assignment task is mostly about creating views!

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

Summary SQL DDL

- Mostly about CREATE TABLE (<list of columns+types and constraints>)
- On an abstract level there are three important kind of constraints:
  - Key constraints (PRIMARY KEY and UNIQUE)
  - Reference constraint (FOREIGN KEY)
  - Value constraints (CHECK)
- Combined, these constraints can provide powerful integrity guarantees
- Schemas can be semi-mechanically translated into CREATE TABLE
- The art of constructing sensible schemas will be covered in the design part of the course

SQL: 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

- We have already seen several examples of INSERT
- General form INSERT INTO <table name> VALUES (<expressions>);
- Can fail due to constraints on the table

DELETE

- General form:

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

- Examples:

```
DELETE FROM Students;

DELETE FROM Students WHERE name = 'Jonas';

DELETE FROM Grades
WHERE course = 'TDA357' AND student='111111-1111';
```

Deletes all rows (!)

Can use AND/OR/NOT and a bunch of other stuff (e.g. IN)

Quiz

- Describe in english what this statement does:

```
DELETE FROM Grades
   WHERE grade < 3 AND grade > 5;
```

  - Answer: Nothing (condition is always false)

- Describe in english what this statement does:

```
DELETE FROM Grades
   WHERE grade < 3 OR grade > 5;
```

  - Answer: Delete all recorded grades below 3 and all above 5

UPDATE

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

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

- Can update multiple attributes, e.g.

```
UPDATE Students SET name = 'Jonas', cid='jonasdu'
  WHERE idNumber = '840118-4893';
```

- Condition can be omitted to change all rows:

```
UPDATE Grades SET grade = 0;
```

  - Sets all grades in all courses to 0 ☹
- UPDATE never removes or adds any rows

Quiz

- Do you think an update can trigger errors? How?
    - The updated value may violate uniqueness/value constraints
    - The updated value may be referenced in other tables
    - Types may be incorrect (like giving a TEXT value to an INT)
    - …

Quiz

- What does this statement do?

```
UPDATE Grades SET grade = grade + 1
  WHERE course = 'TDA357' AND grade IN (3,4);
```

  - Answer: Everyone with a 3 or 4 in databases gets a higher grade (yay!)
- What happens if you run it twice?
  - Everyone who had a passing grade will have a 5.

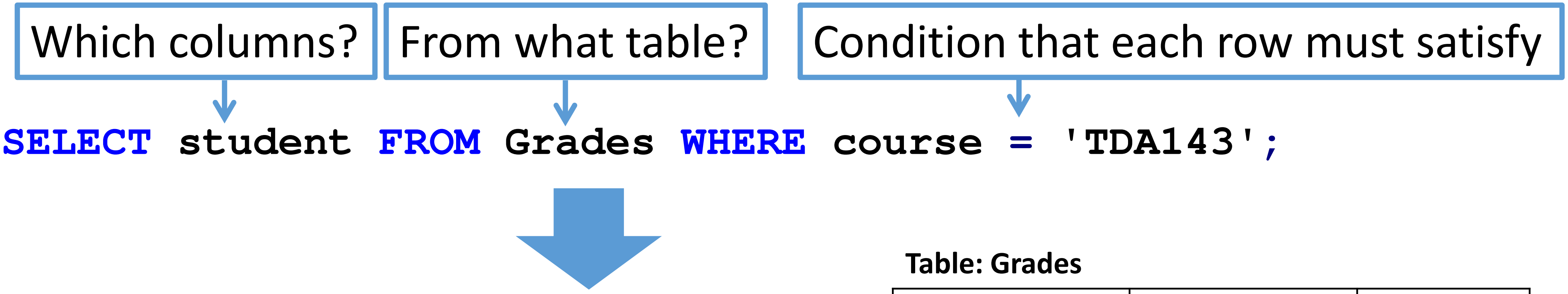
SQL: SELECT (Queries)

- The main part of using SQL is writing queries
- Probably 80% or more of your time on Assignment part 1 is writing queries

- Spoiler alert: The last view takes *a lot* more time than the previous ones

SQL Queries

- The result of each query is a table
- Example: Fetch personal number for each student that has a grade in TDA143:



Which columns?

From what table?

Condition that each row must satisfy

```
SELECT student FROM Grades WHERE course = 'TDA143';
```

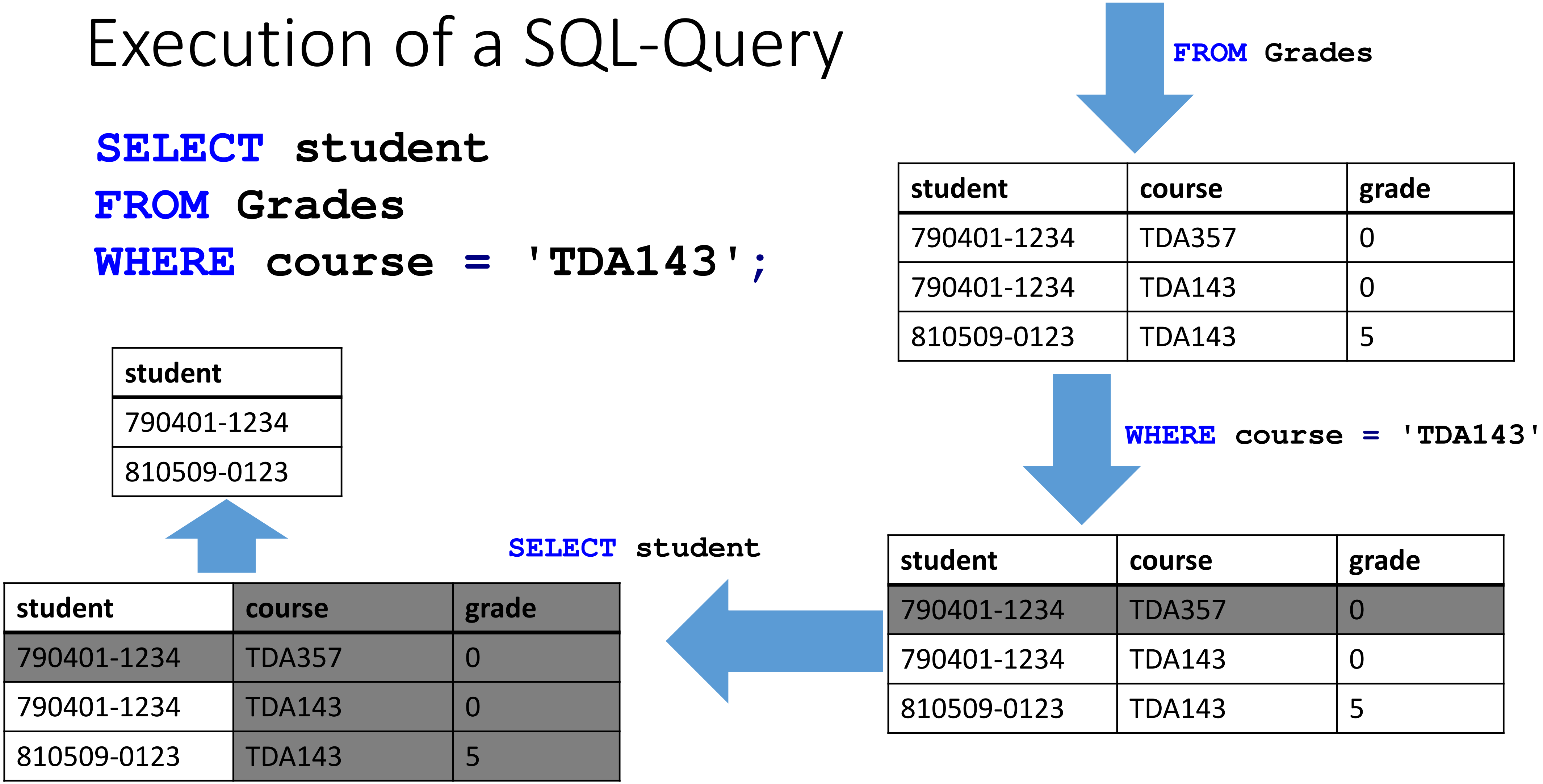
student
790401-1234
810509-0123

Table: Grades

student	course	grade
790401-1234	TDA357	0
790401-1234	TDA143	0
810509-0123	TDA143	5

Execution of a SQL-Query

```sql
SELECT student
FROM Grades
WHERE course = 'TDA143';
```



FROM Grades

student	course	grade
790401-1234	TDA357	0
790401-1234	TDA143	0
810509-0123	TDA143	5

WHERE course = 'TDA143'

student	course	grade
790401-1234	TDA357	0
790401-1234	TDA143	0
810509-0123	TDA143	5

SELECT student

student	course	grade
790401-1234	TDA357	0
790401-1234	TDA143	0
810509-0123	TDA143	5

student
790401-1234
810509-0123

- EXAMPLE: Write a query that selects every student that passed TDA143 (grade 3 or higher) along with the grade they got

```sql
SELECT student, grade
FROM Grades
WHERE course = 'TDA143' AND (grade >= 3);
```

student	grade
810509-0123	5

Table: Grades

student	course	grade
790401-1234	TDA357	3
790401-1234	TDA143	0
810509-0123	TDA143	5

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, if N=|S| and M=|T| then N*M = |S ×T| (Example: S has three elements and T has four, so 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

Cartesian product in SQL

Table: Courses

code	coursename	points
TDA357	Databases	6.5
TDA143	Programmerade system	7.5

Table: Grades

student	course	grade
790401-1234	TDA357	0
790401-1234	TDA143	3
810509-0123	TDA143	5

All columns

From the Cartesian product of courses and grades

```
SELECT *
 FROM Courses, Grades;
```

code	coursename	points	idNumber	course	grade
TDA357	Databases	6.5	790401-1234	TDA357	0
TDA357	Databases	6.5	790401-1234	TDA143	3
TDA357	Databases	6.5	810509-0123	TDA143	5
TDA143	Programmerade system	7.5	790401-1234	TDA357	0
TDA143	Programmerade system	7.5	790401-1234	TDA143	3
TDA143	Programmerade system	7.5	810509-0123	TDA143	5

2*3=6 rows

3+3 = 6 columns

Join-operation

- Suppose we want the name of everyone with a grade in TDA143
- Look at the Cartesian product of Students and Grades (Students × Grades)
- The rows where the personal numbers match are the relevant ones, the rest are nonsense

Table: Students

idNumber	name	CID
790401-1234	Bart Simpson	barsimp
810509-0123	Lisa Simpson	simpsol

Table: Grades

student	course	grade
790401-1234	TDA357	0
790401-1234	TDA143	0
810509-0123	TDA143	5

idNumber	name	CID	student	course	grade
790401-1234	Bart Simpson	barsimp	790401-1234	TDA357	0
790401-1234	Bart Simpson	barsimp	790401-1234	TDA143	0
790401-1234	Bart Simpson	barsimp	810509-0123	TDA143	5
810509-0123	Lisa Simpson	simpsol	790401-1234	TDA357	0
810509-0123	Lisa Simpson	simpsol	790401-1234	TDA143	0
810509-0123	Lisa Simpson	simpsol	810509-0123	TDA143	5

```
SELECT idNumber, name, grade
FROM Students, Grades
WHERE (idNumber=student) AND (course='TDA143');
```

idNumber	name	CID	student	course	grade
790401-1234	Bart Simpson	barsimp	790401-1234	TDA357	0
790401-1234	Bart Simpson	barsimp	790401-1234	TDA143	0
790401-1234	Bart Simpson	barsimp	810509-0123	TDA143	5
810509-0123	Lisa Simpson	simpsol	790401-1234	TDA357	0
810509-0123	Lisa Simpson	simpsol	790401-1234	TDA143	0
810509-0123	Lisa Simpson	simpsol	810509-0123	TDA143	5

```
SELECT idNumber, name, grade
```

idNumber	name	grade
790401-1234	Bart Simpson	0
810509-0123	Lisa Simpson	5

Qualified names

namechange

Table: Students

idNumber	**name**	**CID**
790401-1234	Bart Simpson	barsimp
810509-0123	Lisa Simpson	simpsol

Table: Grades

idNumber	**course**	**grade**
790401-1234	TDA357	0
790401-1234	TDA143	0
810509-0123	TDA143	5

- This doesn't work (ambiguous column name):

```
SELECT idNumber, name, grade
 FROM Students, Grades
 WHERE (idNumber=idNumber);
```

- One (sometimes) solution: Use unique names in tables
- Another solution: Use qualified names to distinguish attributes:

```
SELECT Students.idNumber, name, grade
 FROM Students, Grades
 WHERE (Students.idNumber=Grades.idNumber);
```

both in select/where

Regarding duplicates

- Note that results may contain duplicate rows
  - There are no primary keys in results (only in created tables)

```
SELECT name
FROM Students, Grades
WHERE (Students.idNumber = Grades.student);
```

name
Bart Simpson
Bart Simpson
Lisa Simpson

Each name is repeated for every course taken by a student with that name

Summary: basic SQL-expressions

```
SELECT attribute1, attrubute2 ...
FROM Table1, Table2 ...
WHERE Condition (attribute1 = attrubute2 OR attribute3 = 'text')
```

Compute the result by:

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

JOIN keyword

```
SELECT *
 FROM Students JOIN Grades ON (idNumber=student);
```

is the same as

```
SELECT *
 FROM Students, Grades
 WHERE (idNumber=student);
```

- In general:

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

  is the same as

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

Using

Instead of:

```
SELECT Students.idNumber, name, grade
 FROM Students, Grades
 WHERE (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 occurence of idNumber in the cartesian product! (So I don't need to use qualified names in SELECT)

NATUAL JOIN – the least natural join in the world

- Writing

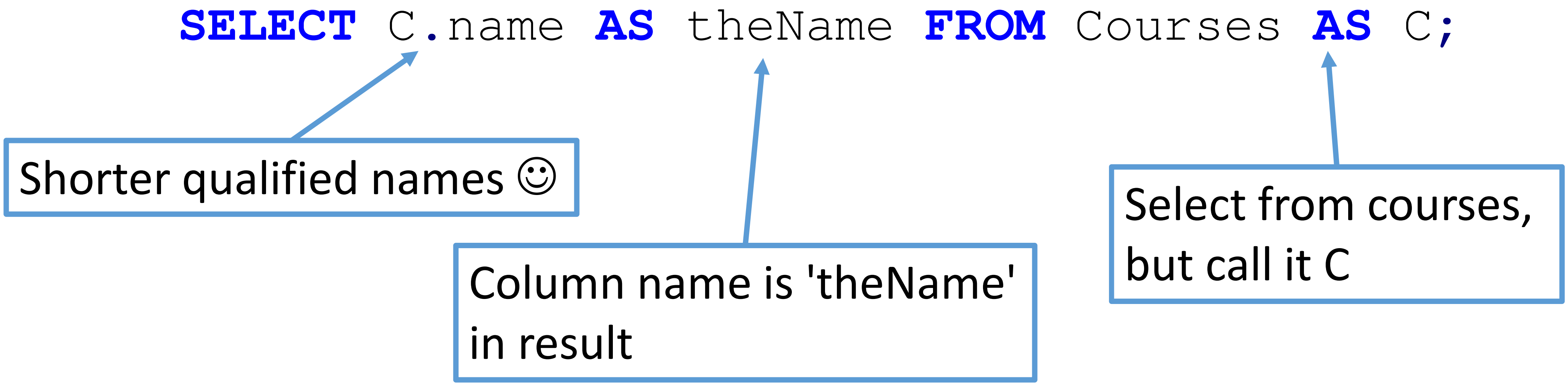
```
SELECT *
 FROM Students NATURAL JOIN Grades;
```

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 two tables (Especially for things like "A NATURAL JOIN B NATURAL JOIN C" …)

Aliasing tables and columns

- 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

- Sometimes useful when selecting constants:

```
SELECT name, 'hello!' AS Message FROM Courses;
```

Selects each course name, along with the text 'hello!' on each row…

Quiz: Self join 1

Table: Numbers

owner	**num**
Bart	11111
Lisa	22222
Bart	33333

- What does this query yield? (how many rows?)

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

- Answer:

N1.num	**N2.num**	**N1.owner**
11111	11111	Bart
11111	33333	Bart
22222	22222	Lisa
33333	11111	Bart
33333	33333	Bart

Quiz: Self join 2

Table: Numbers

owner	**num**
Bart	11111
Lisa	22222
Bart	33333

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

- Answer: 2 rows

N1.num	**N2.num**	**N1.owner**
~~11111~~	~~11111~~	~~Bart~~
11111	33333	Bart
~~22222~~	~~22222~~	~~Lisa~~
33333	11111	Bart
~~33333~~	~~33333~~	~~Bart~~

Quiz: Self join 3

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

Table: Numbers

owner	**num**
Bart	11111
Lisa	22222
Bart	33333

N1.num	**N2.num**	**N1.owner**
11111	33333	Bart

What about this one:

Table: Numbers

owner	**num**
Bart	11111
Bart	22222
Bart	33333

N1.num	**N2.num**	**N1.owner**
11111	22222	Bart
11111	33333	Bart
22222	33333	Bart

Quiz

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

- We have a table of names+phone numbers, and one of names+email
- What do we get from the following expression?

```
SELECT Phones.name,phone,email
 FROM Phones, Emails
 WHERE Phones.name = Emails.name;
```

(Or semi-equivalently `FROM Phones NATURAL JOIN Emails`)

- Do we get all the data from both tables?
  - No! Only the names that have both a phone number and an email appear in the result! (example on next slide)

Table: Phones

name	phone
Bart	11111
Lisa	22222
Maggie	33333

Table: Emails

name	email
Bart	bart
Lisa	lisa
Homer	homer

Inner join

```
SELECT *
 FROM Phones, Emails
 WHERE Phones.name = Emails.name;
```

Result:

Phones.name	phone	Emails.name	email
Bart	11111	Bart	bart
Lisa	22222	Lisa	lisa

Result says nothing about Maggie and Homer ☹

Table: Phones

name	phone
Bart	11111
Lisa	22222
Maggie	33333

Table: Emails

name	email
Bart	bart
Lisa	lisa
Homer	homer

(full) outer joins

- Outer joins are intended to solve exactly this kind of problems: we want everyone who has a phone number OR an email
- 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

```
SELECT * FROM Phones FULL 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)
(null)	(null)	Homer	homer

Regular (inner) join

Extra outer-join rows

Natural join to the rescue

Table: Phones

name	phone
Bart	11111
Lisa	22222
Maggie	33333

Table: Emails

name	email
Bart	bart
Lisa	lisa
Homer	homer

- The weird column-merging stuff that NATURAL JOIN (and USING) does works sort of nicely here:

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

- No nulls in joined columns
- Looks like what I'd expect a combination of the two tables to look like

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

Start of lecture 3

If you're still looking for a lab partner, talk to me during the break

The story so far:

- Use DDL (Data Definition Language) to create tables
- Use DML (Data Manipulation Language) to INSERT/DELETE/MODIFY rows
- Simple queries: SELECT <attributes> FROM <tables> WHERE <condition>;
- Qualified names: Table.column
- x JOIN y ON condition, x JOIN y ON attribute, x NATURAL JOIN y
- FULL OUTER JOIN to get "missing" rows from either table, and NULL values fot the columns from the other table

Table: Phones

name	phone
Bart	11111
Lisa	22222
Maggie	33333

Table: Emails

name	email
Bart	bart
Lisa	lisa
Homer	homer

Left/Right outer join

- Specifying left/right outer join (instead of full) means only missing rows from the left/right operand of JOIN are added

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

- No extra row for homer
- Never any new null values in Phones.x (left side of result)

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

Experiment with outer joins

- Play around with OUTER joins, and get some unexpected results
- For instance, these queries give slightly different results:

```
SELECT Emails.name,phone,email
FROM Phones LEFT OUTER JOIN Emails
                         ON (Phones.name=Emails.name);

SELECT name,phone,email
 FROM Phones LEFT OUTER JOIN Emails USING (name);
```

COALESCE

- 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 …
```

```
SELECT id,COALESCE(totalCredits, 0) AS credits FROM …
```

Use aliasing to give the coalesced values proper names

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

Sets, Bags or Lists?

- 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 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

Table: Grades

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

- By adding DISTINCT after SELECT, duplicate rows will be removed

```
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

- Gives the query set-semantics

Set operations

- There are three set-opertions in SQL: UNION, INTERSECT and EXCEPT
- You use <query1> UNION <query2>
- Union example:

A query containting queries!

```
(SELECT room FROM Lectures)
UNION
(SELECT eroom FROM Exercises);
```

Must have matching column types

UNION

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

- 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

```
(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	11-10 10:00	lecture
HB	11-06 13:00	ER

INTERSECT

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

- Takes the intersection of two queries (all rows that appear in both)

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

Result:

lroom	**ltime**
GD	11-06 8:00

EXCEPT

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

- Takes the difference of two queries (removing the contents of the second from the first)

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

Result:

lroom	ltime
GD	11-06 10:00

UNION ALL

- If we don't care about duplicates, we can use UNION ALL (also INTERSECT ALL and EXCEPT ALL) to keep duplicates (bag-semantics)

```
(SELECT room FROM Lectures)
UNION ALL
(SELECT eroom FROM Exercises);
```

- The query above may give the same room multiple times
- Presumably, it is slightly more efficient
- Ordering is not necessarily preserved

Ordering

Table: Numbers

owner	**num**
Bart	44444
Lisa	22222
Bart	33333
Homer	11111

- 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

```
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

Aggregation

You know it's important because it got a vertically centered headline

Aggregation

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

- Some data we cant quite compute yet:
  - How many courses has Bart passed?
  - What is the average grade in Databases?
  - How many points does Lisa have in total?
  - What is Barts maximum grade?
- These operations are called aggregates
  - Require us to process *groups* of values together
  - Aggregate a set of values into a single value (like the average or sum)

Simple aggregates

- Aggregate functions:
  - COUNT counts rows, AVG computes averages

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

- Always gives a single row
- WHERE applied before aggregation
- Can not mix columns and aggregates (~~SELECT student, AVG(grade)~~)

Quiz

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

- How do I write a query that computes Lisas total points?
- Hint: There is an aggregation function called SUM
- Take one clause at the time, starting with FROM, then WHERE, then SELECT

```
SELECT SUM(points)
 FROM Courses, Grades
 WHERE name = course
   AND grade >= 3
   AND student = 'Lisa';
```

total
25

Intermediate result before aggregation, after WHERE

name	points	student	course	grade
Databases	10	Lisa	Databases	4
Project	15	Lisa	Project	5

Grouping

Table: Grades

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

- I want the average (passing) grade for each student

? →

Table: Courses

student	AVG
Bart	3.0
Lisa	4.5

- To do this, I need to tell SQL to group all the values in Grades by the student attribute (two groups) then for each group select the (unique) student and compute the average of the grades in the group

```
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)

Quiz

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

- For each course, lists its name, points and number of passed students
- Start with FROM, then WHERE, then SELECT and GROUP BY

```
SELECT name, points, COUNT(*) AS passed
  FROM Courses, Grades
  WHERE course = name
    AND grade >= 3
  GROUP BY (name,points);
```

name	points	passed
Databases	10	2
Project	15	1

HAVING

- What if I want to list all students with an average above 4?
- This does not work (the WHERE-clause resolves before the grouping!)

```
SELECT student
 FROM Grades
 WHERE grade >= 3 AND AVG(grade) > 4
 GROUP BY student;
```

- SQL has a special clause for conditions on groups, called HAVING

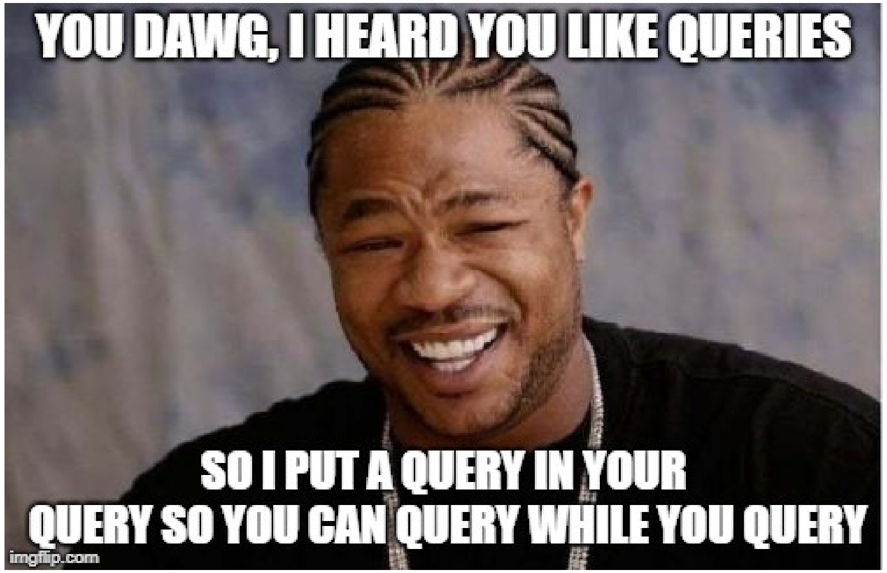
```
SELECT student
 FROM Grades
 WHERE grade >= 3
 GROUP BY student
 HAVING AVG(grade) > 4;
```

Resolved before grouping (to exclude 0)

Resolved during grouping (condition for each group)

Subqueries

Wouldn't our queries be even more awesome if we had queries in them?


YOU DAWG, I HEARD YOU LIKE QUERIES
SO I PUT A QUERY IN YOUR
QUERY SO YOU CAN QUERY WHILE YOU QUERY
imgflip.com

This is possibly the most dreaded and most powerful feature of SQL

Where can you have subqueries?

- Where can you not have subqueries?!
- You can have them in FROM

Cartesian product of a table and a query result

```
SELECT name
 FROM Courses, (SELECT course, COUNT(*) AS graded
                FROM Grades GROUP BY course) AS Q
 WHERE Courses.name = Q.course AND Q.graded > 100;
```

- You can have them in WHERE

Select all above average grades

```
SELECT * FROM Grades
 WHERE grade > (SELECT AVG(grade) FROM Grades);
```

Comparison only works if subquery gives a single row

- You can even have them in SELECT!
- And of course -my personal favorite- you can have them in subqueries

Using subqueries to filter results of set operations

- You can not attach an WHERE-clause directy to a UNION (only SELECT)
- But you can have the UNION in a FROM clause:

```
SELECT time
FROM (SELECT time,room FROM Lectures
      UNION
      SELECT time,room FROM Exercises) AS U
WHERE room='GD';
```

Gets the time of all lectures and exercises in room GD

EXISTS and NOT EXISTS and correlated queries

- A common use of subqueries is something like this:

```
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 (NOT EXIST (<query>) is true if <query> gives zero result rows)
- 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 can not be executed by itself
  - The qualified name is not needed but highly recommended for readability

WITH

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

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

"helper queries"

Final result query

- Compare to how you may create helper methods in java to split up a complicated piece of code
- Note that the whole thing is a single query that gives one result table, but it contains subqueries

Employees(idnr,company,division,salary)

WITH-example

"Find all divisions whose total salary exceeds the average total division salary in its company"

```
WITH
 DivisionTotals AS
  (SELECT company, division, SUM(salary) as total
   FROM Employees GROUP BY company, division),
 CompanyAverage AS
  (SELECT company, AVG(total) as average
   FROM DivisionTotals GROUP BY company)
```

Two helper tables (DivisionTotals and CompanyAverage)

One helper table uses the other!

```
SELECT company, division
  FROM DivisionTotals JOIN CompanyAverage USING (company)
  WHERE total > average;
```

Final result

Using WITH to make one column at a time

```
WITH
  Col1 AS <query>,
  Col2 AS <query>,
  Col3 AS <query>
SELECT <attribute list>
 FROM Col1
      NATURAL LEFT OUTER JOIN Col2
      NATURAL LEFT OUTER JOIN Col3;
```

Each helper table has one or a few columns for the end result, and a common identifier

Chain of LEFT-joins, the leftmost one needs to have all rows we want

- Good way of building complex queries (like the last view in Task 1)
- Each of the column-queries can be executed and tested separately
- You can do this by creating views, but that "pollutes the namespace"

Weird stuff in SQL conditions

What do we get if we have WHERE x=y and x is NULL?

- NULL is not a value so FALSE? NULL is a wildcard value so TRUE?
- The SQL developers couldn't decide, so they added a third value to the boolean type, UNKNOWN, and any comparisons to null give this value
- This can be very confusing. For instance:
  - x=x is not always true (if x is NULL it is UNKNOWN)
  - p OR NOT p is not always true
  - TRUE OR UNKNOWN is TRUE, FALSE OR UNKNOWN is UNKNOWN
  - Truth tables for binary logical operators now have 9 rows instead of 4
- UNKNOWN is counted as FALSE (excluded) in WHERE-clauses
- Use "x IS NULL" to check if attribute x is null (always TRUE/FALSE), or COALESCE

Comments in SQL-files

```
-- This is a single line comment (starts with --)
/* This is a
multiline comment
*/
```

- Writing comments is good for yourself, your lab partner, and graders
- Can also be used to comment out SQL code that currently doesn't work, write TODO:s etc (just clean it up before submitting!)

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 …

Workflow for writing a complicated query

- Start with some data and an understanding of what your query should result in for the test data you have in your tables (add more if needed)
- Write a simple query that shows some of the data you want (e.g. some of the column and most of the rows)
- Wrong number of rows?
  - Sometimes: Modify your WHERE/HAVING conditions
  - Sometimes: Add another table/query to FROM
  - Rarely: Use UNION to add what is missing
- Missing columns?
  - join in another table or subquery, add aggregations …

You now know everything needed for Task 1

- Go forth and solve!

Friday exercises

- Today after lunch there are two exercise sessions
  - The 13-15 one will be in this room, the 15-17 one in Vasa A
- The later one is mainly for the I-students and anyone else who is busy 13-15
- In this weeks sessions you will learn the practicalities of working SQL by working together on a set of problems (find them on the webpage)

This is slide 100!

- It contains nothing useful, I just wanted 100 slides.