

EDA216 Database Technology

Lecture 2

Christian.Soderberg@cs.lth.se

January 18, 2017

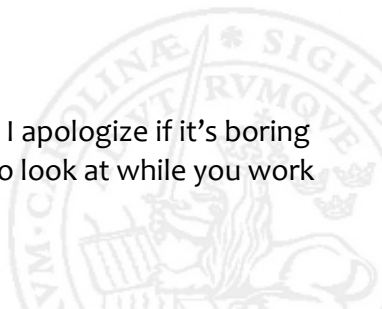


Christian.Soderberg@cs.lth.se

1 / 1

Today's topics

- ▶ Defining tables
- ▶ Inserting values into our tables
- ▶ Updating and deleting values
- ▶ Set operations
- ▶ Views
- ▶ Joins
- ▶ Subqueries
- ▶ There will be lots and lots of examples today, I apologize if it's boring to watch, but I want you to have something to look at while you work on lab 1

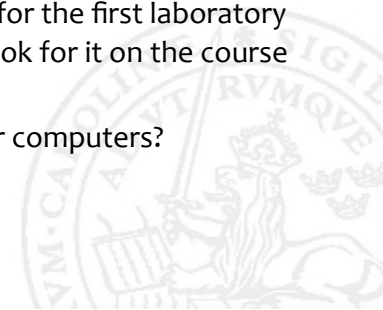


Christian.Soderberg@cs.lth.se

3 / 1

Administration

- ▶ The laboratory exercises have been rescheduled
 - ▶ Lab 4 is no longer with us
 - ▶ Lab 1-3 will take place in weeks 3-5 instead
- ▶ I will update the web site in the next few days
- ▶ This lecture is just supposed to prepare you for the first laboratory exercise, after the lecture I'll put the notebook for it on the course web site
- ▶ Have you managed to install Jupyter on your computers?



Christian.Soderberg@cs.lth.se

2 / 1

Defining Tables

- ▶ We create a table with CREATE TABLE (and, possibly, IF NOT EXIST)
- ▶ We then define the names and types of our attributes
- ▶ Standard types:
 - ▶ CHAR(n)
 - ▶ VARCHAR(n)
 - ▶ BOOLEAN
 - ▶ INT, INTEGER
 - ▶ FLOAT, REAL
 - ▶ DECIMAL(n,d)
 - ▶ DATE, TIME



Christian.Soderberg@cs.lth.se

4 / 1

Defining constraints

We often want to ensure that our data is not corrupt, and we have several tools for that:

- ▶ Key constraints
- ▶ Foreign-key constraints
- ▶ Assertions
- ▶ Triggers



Christian.Soderberg@cs.lth.se

5 / 1

Primary Keys

```
CREATE TABLE students (  
  ssn      CHAR(11) PRIMARY KEY,  
  name     VARCHAR(32),  
  program  INT,  
  year     INT  
)
```

or

```
CREATE TABLE students (  
  ssn      CHAR(11),  
  name     VARCHAR(32),  
  program  INT,  
  year     INT,  
  PRIMARY KEY (ssn)  
)
```

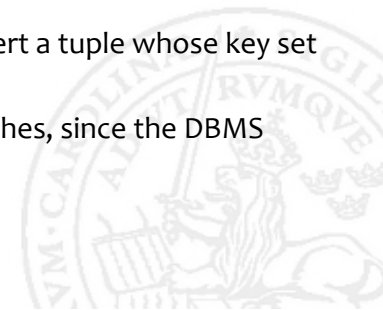


Christian.Soderberg@cs.lth.se

7 / 1

Keys and key constraints

- ▶ A key on relation is an attribute, or a set of attributes, which uniquely identifies each tuple
- ▶ For a table with Swedish citizens, the “personnummer” is such a key
- ▶ For a movie, the title might not be unique, but title and year is normally unique
- ▶ The DBMS would complain if we tried to insert a tuple whose key set is already in the table
- ▶ Searches on keys are faster than other searches, since the DBMS keeps special indexes for keys



Christian.Soderberg@cs.lth.se

6 / 1

Primary Keys

```
CREATE TABLE movies (  
  title    VARCHAR(64),  
  year     INT,  
  studio   INT,  
  PRIMARY KEY (title, year)  
)
```



Christian.Soderberg@cs.lth.se

8 / 1

UNIQUE

We can declare a key using the **UNIQUE** keyword – it's almost the same as a **PRIMARY KEY**, but:

- ▶ There can be more than one **UNIQUE** declaration in each table (but only one primary key)
- ▶ A **UNIQUE** attribute can be allowed to be **NULL**, and **NULL**-values are always considered to be unique

Key Restrictions

- ▶ Two tuples in a table can't have the same key (i.e., the set of attributes which makes up the key can't have the same value)
- ▶ Attributes in the key can't be **NULL**

Invented Keys

- ▶ We can add a simple identifying integer, as a simple key
- ▶ We can declare it using:

```
CREATE TABLE movies (  
    id      INTEGER AUTOINCREMENT,  
    title   VARCHAR(64),  
    year    INT,  
    studio  INT,  
    PRIMARY KEY (id)  
)
```

- ▶ SQLite gives us such a key automatically (**ROWID**)
- ▶ This kind of key has also been called surrogate key, synthetic key, entity identifier, system-generated key, database sequence number, factless key, technical key, or arbitrary unique identifier

Foreign Keys

- ▶ A *foreign key* is an attribute which references a key in another relation
- ▶ We will normally get an error if we try to insert a value with no corresponding foreign key
- ▶ We can turn on and off the checking of foreign keys (especially on bulk inserts, it can be a good idea)

Foreign Keys

```
CREATE TABLE movies (  
  id          INTEGER PRIMARY KEY,  
  title       VARCHAR(64),  
  year        INT,  
  studio_id   INT REFERENCES studios(id),  
  PRIMARY KEY (id)  
)
```

or

```
CREATE TABLE movies (  
  id          INTEGER PRIMARY KEY,  
  title       VARCHAR(64),  
  year        INT,  
  studio_id   INT,  
  PRIMARY KEY (id),  
  FOREIGN KEY (studio_id) REFERENCES studios(id)  
)
```

Christian.Soderberg@cs.lth.se

13 / 1

Modifying tables

- ▶ We can delete a table:

```
DROP TABLE students
```

- ▶ Add a column:

```
ALTER TABLE students ADD stil_id VARCHAR(32)
```

- ▶ Delete a column:

```
ALTER TABLE students DROP stil_id
```

- ▶ Adding and deleting columns is more common than one might think – databases are long-lived and often change many times during their lifetimes

Christian.Soderberg@cs.lth.se

15 / 1

Constraints on Attributes

- ▶ We can make sure an attribute doesn't get the value NULL by declaring it NOT NULL
- ▶ We can also add simple checks, such as

```
CREATE TABLE students (  
  ssn         CHAR(11) PRIMARY KEY,  
  name        VARCHAR(32) NOT NULL,  
  program     INT,  
  year        INT CHECK (year > 2010)  
)
```

Christian.Soderberg@cs.lth.se

14 / 1

Inserting values

- ▶ We use the INSERT keyword to insert values
- ▶ Two common methods, assuming we have:

```
students(ssn, name, program, year)
```

The safest method:

```
INSERT INTO students (ssn, name, program, year)  
VALUES ('910101-1234', 'Oddput Clementin', 42, 2016)
```

Slightly riskier:

```
INSERT INTO students  
VALUES ('910101-1234', 'Oddput Clementin', 42, 2016)
```

Christian.Soderberg@cs.lth.se

16 / 1

Updating values

- ▶ To update a value, we use the UPDATE keyword

```
UPDATE students
SET   name = 'Oddput Orange'
WHERE ssn = '910101-1234'
```

Deleting values

- ▶ To delete a value, we use the DELETE keyword

```
DELETE FROM students
WHERE ssn = '910101-1234'
```

- ▶ If we omit the WHERE clause, the whole table will be emptied (but the table will not be removed, as when we use DROP TABLE)

Indexes

- ▶ As stated above, searching for key values is faster than other searches, since the keys are indexed
- ▶ If we often search for something that isn't a key, we can speed things up by creating an index:

```
CREATE INDEX program_year ON students(program, year)
```

- ▶ We can also drop an index:

```
DROP INDEX program_year ON students
```

- ▶ Adding an index makes queries speedier, but it also makes insertions, deletions and updates slower

Views

- ▶ The tables we create using CREATE TABLE exist physically in the database
- ▶ We can also create views, they are logical tables, and don't exist physically

```
CREATE VIEW freshmen AS
SELECT ssn, name
FROM   students
WHERE  year = 2016
```

- ▶ We use it as if it were a table:

```
SELECT ssn
FROM   freshmen
WHERE  name = 'Oddput'
```

Views

- ▶ Views help us break things into smaller parts
- ▶ They let us reuse our queries
- ▶ Some, but not all, DBMS lets us modify relations via views
- ▶ A *materialized view* (`CREATE MATERIALIZED VIEW`) is stored in the database, it's efficient if we use the view often, but must be recomputed when underlying values changes (not all DBMS's have materialized views)

Joins

- ▶ We saw some examples of *joins* last time, when we used several tables in one query
- ▶ There are several kinds of joins:
 - ▶ `CROSS JOIN` (or just `,`)
 - ▶ `INNER JOIN` (or just `JOIN`)
 - ▶ `NATURAL JOIN`
 - ▶ `LEFT OUTER JOIN` (or just `LEFT JOIN`)
 - ▶ `RIGHT OUTER JOIN` (or just `RIGHT JOIN`)
 - ▶ `FULL OUTER JOIN` (or just `FULL JOIN`)

Set Operations

- ▶ The most common set operations, *in*, *union*, *intersection*, and *difference*, are available in SQL (but not in all DBMS's)
- ▶ The operands involved must be compatible with each other, i.e., they need to have the same attributes and in the same order
- ▶ Example:

```
SELECT title, year
FROM movies
UNION
SELECT title, year
FROM stars_in
```

Example

Assume we have

```
stars(name, address, gender)
execs(name, address, net_worth)
```

Find rich female movie stars who are also executives

```
SELECT name
FROM stars
WHERE gender = 'F'
INTERSECT
SELECT name
FROM execs
WHERE net_worth > 10000000
```

Example

Assume we have

```
stars(name, address, gender)
execs(name, address, net_worth)
```

Find movie stars who are not also executives

```
SELECT name
FROM stars
EXCEPT
SELECT name
FROM execs
```

Christian.Soderberg@cs.lth.se

25 / 1

Simpler solution

```
SELECT name
FROM movies, movie_execs
WHERE title = 'Star Wars' AND
      prod_no = cert_no
```

Christian.Soderberg@cs.lth.se

27 / 1

Subqueries

► Assume we have

```
movies(title, year, length, studio_name, prod_no)
stars_in(title, year, star_name)
movie_execs(name, address, cert_no, net_worth)
```

► Find the producer of Star Wars:

```
SELECT name
FROM movie_execs
WHERE cert_no =
      (SELECT prod_no
       FROM movies
       WHERE title = 'Star Wars')
```

Christian.Soderberg@cs.lth.se

26 / 1

Subqueries

► Assume once again that we have

```
movies(title, year, length, studio_name, prod_no)
stars_in(title, year, star_name)
movie_execs(name, address, cert_no, net_worth)
```

► Find all producers of movies where Harrison Ford stars:

```
SELECT name
FROM movie_execs
WHERE cert_no IN
      (SELECT prod_no
       FROM movies
       WHERE (title, year) IN
              (SELECT title, year
               FROM stars_in
               WHERE star_name = 'Harrison Ford'))
```

Christian.Soderberg@cs.lth.se

28 / 1

Simpler solution

```
SELECT name
FROM movie_execs e, movies m, stars_in s
WHERE cert_no = prod_no AND
      m.title = s.title AND
      m.year = s.year AND
      star_name = 'Harrison Ford'
```

Christian.Soderberg@cs.lth.se

29/1

Example

Find titles that have been used for two or more movies (produced in different years)

- Using a correlated subquery:

```
SELECT title, year
FROM movies old
WHERE year <> ANY
      (SELECT year
       FROM movies
       WHERE title = old.title)
```

- With a join:

```
SELECT old.title, old.year
FROM movies old, movies new
WHERE old.title = new.title AND
      old.year <> new.year
```

Christian.Soderberg@cs.lth.se

31/1

Example

Find the producers who haven't produced any movies (this cannot be written as a join)

```
SELECT name
FROM movie_execs
WHERE cert_no NOT IN
      (SELECT prod_no
       FROM movies)
```

Christian.Soderberg@cs.lth.se

30/1

Aggregation

- Some operators can be applied to a column: SUM, AVG, MIN, MAX, COUNT
- Examples:

```
SELECT AVG(net_worth)
FROM movie_execs
```

```
SELECT COUNT(*)
FROM stars_in
```

— NULL's will count

```
SELECT COUNT(star_name)
FROM stars_in
```

— NULL's will not count

- The aggregation operators may not be used in WHERE clauses — they operate on a whole relation after tuples have been selected with WHERE

Christian.Soderberg@cs.lth.se

32/1

Grouping

- ▶ We can group tuples together with GROUP BY, and then apply an operator to the tuples of the group

- ▶ Examples:

- ▶ Find the length of all movies for each studio:

```
SELECT  studio_name, sum(length)
FROM    movies
GROUP BY studio_name
```

- ▶ Find the total length of all movies produced by each producer:

```
SELECT  name, sum(length)
FROM    movies, movie_execs
WHERE   prod_no = cert_no
GROUP BY name
```

The HAVING keyword

- ▶ We can control which groups should be present in the output by introducing a condition for the group.
- ▶ Example: Find the composers, except Verdi, who have written more than two operas:

```
SELECT  composer, COUNT(*)
FROM    operas
WHERE   composer <> 'Verdi'
GROUP BY composer
HAVING  COUNT(*) > 2
```

- ▶ Observe the order of the selection:
 1. first WHERE (determine which tuples to include),
 2. then GROUP BY,
 3. last HAVING (to determine which groups to include)