

universität freiburg

Databases and Information Systems WS 25/26

Lecture 3: Database Basics

October 28, 2025

Prof. Dr. Hannah Bast
Department of Computer Science
University of Freiburg

Overview of this lecture

■ Organizational

- Your experiences with ES2
- Exam and exercise sheets
- No lecture next week !!!

Ranking and Evaluation

Don't procrastinate please

Next lecture is on Nov 11

■ Contents

- Tables Relational model, sets vs. multisets
- SQLite3 A very easy-to-use database system
- Design Multiple tables, primary key, foreign key
- SQL Basics SELECT, FROM, WHERE

- **ES3:** Gather some movie data, design suitable tables, load them into SQLite3, translate some questions to SQL queries

Experiences with ES2 1/3

■ Excerpts from your feedback [somewhat representative]

"This sheet was nicely built on the first one"

"Sheet was fun but took me a little longer than the first one"

"I really liked the amount of feedback from the last time"

"I had problems with the worksheet, mostly because I haven't programmed in a too long time"

"I had a lot of difficulties with this sheet ... On a positive note: I have learned a lot about debugging"

"In general, I was a little bit overwhelmed by this weeks lecture, since it transported so much information at once"

"This exercise was really absurdly hard for me ... interesting, but I did not quite understand all the precision formula stuff"

Experiences with ES2 2/3

■ Your favorite quote or movie scene

"The coin toss scene from No Country For Old Man"

"In the mines of Moria from Lord of the Rings [Fellowship of the Ring]"

"Hello, there from Star Wars: Revenge of the Sith"

"Life finds a way from Jurassic Parc"

"The bank robbery scene from the movie Heat, one of the best shooting scenes of all time, made over 30 years ago"

"Ernest Hemingway once wrote: "The world is a fine place and worth fighting for", I agree with the second part" from Se7en

"Every scene with WALL-E gives me a good mood instantly"

■ Results

- Values for b close to zero (but not zero) gave the best results

This corresponds to a very slight (but non-zero) adjustment for documents, where the length deviates from the average

- Small values of k (around 1) gave the best result

A value of $k = 1$ corresponds to tf^* values in the range $[1, 2]$ when $tf > 0$ (that is, when a term occurs in the document at all)

- Baseline results : $MP@3 = 58\%$, $MP@R = 42\%$, $MAP = 42\%$

Your best results : $MP@3 = 76\%$, $MP@R = 45\%$, $MAP = 48\%$

Note: these are typical results for search, getting values near 100% is theoretically possible, but extremely hard in practice

* There was a $MAP = 60\%$ result, but that seems too good to be true

Exam and exercise sheets

- For those of you who do not submit anything
 - For the exam, you eventually need this stuff anyway
 - In particular, writing code (though to a smaller extent than in the sheets) is an important part of the exam

Only practice makes perfect, and it is hard to get enough practice when you start two weeks before the exam

There is always a fraction of students who fail the coding tasks because of lack of practice → **don't be one of them**

- Also, it is only when you work on a concrete task that you realize whether you have actually understood the finer points

Typical symptom: shortly before the exam, there are often all kinds of detail questions about the slides or errors are pointed out → it's better when this happens after the lecture

Tables 1/6

■ A database stores its data in **tables**

- The structure of each table is defined by its **schema**: a fixed number of columns, each with a name and a domain

The **domain** defines the set of possible values, see next slide

- The contents of the table is stored in its rows

While the schema of a table is fixed, the rows and their contents can change over time (and a table can have 0 rows)

title	year	rating
The Shawshank Redemption	1994	9.3
The Dark Knight	2008	9.1
Inception	2010	8.8
Fight Club	1999	8.8

* We discuss **sets** vs. **multisets** on the next two slides

■ The **relational model**

- Formally, we define a table with k columns as follows
 - A tuple $C = (c_1, \dots, c_k)$ of column names, one per column
 - A tuple $D = (D_1, \dots, D_k)$ of domains, one per column
 - A multiset* R of tuples (r_1, \dots, r_k) , where $r_j \in D_j$
- For the example table on the previous slide:

$C = (\text{title}, \text{year}, \text{rating})$

$D_1 = \text{all strings}, D_2 = \text{all integers}, D_3 = \text{all floats}$

$R = \{ (\text{"The Shawshank Redemption"}, 1994, 9.3),$
 $(\text{"The Dark Knight"}, 2008, 9.1),$
 $(\text{"Inception"}, 2010, 8.8),$
 $(\text{"Fight Club"}, 1999, 8.8) \}$

■ Sets vs. Multisets, duplicate elements

- A **set** contains each element at most once, whereas a **multiset** can contain the same element multiple times, for example:

$$S = \{ 1, 3, 7 \} \quad |S| = 3$$

$$M = \{ 1, 7, 3, 7, 7 \} \quad |M| = 5 \text{ and } 7 \text{ is contained } 3 \text{ times}$$

- The original relational model only allowed sets

See the original paper from 1969 in the References on [slide 40](#)

- Most modern database systems allow multisets by default, hence we also allow multisets in our formal definition

Understand that checking for duplicates costs time and space

If you want it anyway, most database systems allow you to specify a **uniqueness constraint** when you create a table

■ Sets vs. Multisets, order of the elements

- Neither sets nor multisets have an **order** of their elements

Even if you inserted the rows into a table in a particular order, the database does not have to remember that order

In practice, a database system can sometimes do its work more efficiently when it is allowed to reorder rows

- If you need an order for your rows, simply create your table with an additional column with a row counter

Most database systems (including SQLite3, [slides 13 – 16](#)) have a way to specify a counter (when creating the table) that **auto-increments** whenever you insert a new row

But only use this when needed, it costs time and space

■ Multiple tables

- A database typically has more than one table

We will see many examples of this in the following and for the exercise sheet you also have multiple tables

- Different tables can (and often do) share some columns

We will see that shared columns are important to meaningfully combine data from different tables

- Deciding in which tables (with which schema) to store given data is a problem called **database design**

Database design is important for both usability and efficiency

Finding the right design can be challenging, see slides [17 – 22](#)

■ Queries

- To query (extract data from) one or several tables from a database, there is a formal query language called **SQL**

SQL = Structured Query Language

- Basic SQL queries are easy to write and understand

For example, to get the title and rating of all movies with rating at least 8.0, we can write:

```
SELECT title, rating FROM movies WHERE rating >= 8.0;
```

- There are many dialects of SQL, but the core functionality is the same for (almost) all database systems

We will learn basic SQL **by example** today, which will be good enough for this lecture and ES3 ... more about SQL in Lecture 4

■ Database system

- A system managing relational tables as described is called a **relational database management system (RDBMS)**
- There are many RDBMSs on the market, both commercial and free, open source and closed source

For example: MySQL, PostgreSQL, Microsoft SQL Server, SQLite, Oracle Database, MariaDB, IBM Db2, SAP HANA, ...

- For the lecture and exercise sheets, we will use [SQLite3](#)

It's a fully functional RDBMS, yet very easy to use (hence "lite")

It is also reasonably efficient, when the data is not very large

On the next slides, we provide a crash course

■ Installation and basic usage

- On Debian/Ubuntu install with

`sudo apt install sqlite3`

- Two modes to start SQLite3:

`sqlite3` will work on an in-memory database

`sqlite3 <name>.db` create database in that file, and if file exists, use database from that file

- Two types of commands ... examples on next slides

`SQL` commands: must end with a semicolon

`SQLite` commands: start with a dot, no semicolon at end

■ Some useful SQLite commands by example

- Specify the column separator used for input and output

`.separator " "` use Ctrl+V TAB for TAB !

- Read table from TSV (tab-separated values) file

`.import movies.tsv movies`

- Execute commands from script file (typical suffix is .sql)

`.read <file with commands>`

- Show execution time of every command

`.timer on`

- Show available command or help on particular command

`.help` `.help import`

■ Some useful SQL commands by example

- Create a table with a given schema

```
CREATE TABLE movies(title TEXT, year INTEGER, rating REAL);
```

Explanation: `movies` is the table name; `title`, `year`, `rating` are the column names and `TEXT`, `INTEGER`, `REAL` their domains

- Launch a SQL query ... more examples on [slides 28 – 36](#) and on ES3

```
SELECT title, year, rating FROM movies WHERE rating >= 8.0;
```

- Delete table / index (without error message if it's not there)

```
DROP TABLE IF EXISTS movies;
```

- See the References on [slide 40](#) for a link to **all** SQL constructs supported by SQLite3

■ Motivating example

- On [slide 7](#), we have seen a table "movies"

Important to note: in that table, each movie has **exactly** one "title", one "year", and one "rating"

- Assume we want to add roles information for each movie, that is which actor played in which movie

Note: Each movie has **many** actors participating in it, and most actors play in more than one movie

But we cannot easily store multiple values in a single cell

How to store such data in tables? This is the central question of database design ... let's look at some ideas for solving this

■ Add "roles" information, Solution 1

- When creating our table "movies", have an additional column "roles", which contains the names of all actors concatenated

This is a **terrible** solution in many ways, in particular: it's hard to extract a particular actor from the "roles" column, it's hard to add additional information (e.g., about the role), ...

title	year	rating	roles
The Shawshank Redemption	1994	9.3	Morgan Freeman, Tim Robbins, Rita Hayworth, ...
The Dark Knight	2008	9.1	Morgan Freeman, Christian Bale, Maggie Gyllenhaal, ...
Inception	2010	8.8	Leonardo DiCaprio, Elliot Page, Marion Cotillard, ...
Fight Club	1999	8.8	Edward Norton, Brad Pitt, Helena Bonham-Carter, ...

■ Add "roles" information, Solution 2

- When creating our table "movies", have an additional column "actor", and add one row for each actor, **repeating** the information in the other columns

Actually not that bad, and some of our queries later will have such results, but we should not **create** tables like that

title	year	rating	actor
The Shawshank Redemption	1994	9.3	Morgan Freeman
The Shawshank Redemption	1994	9.3	Tim Robbins
The Dark Knight	2008	9.1	Morgan Freeman
The Dark Knight	2008	9.1	Christian Bale
The Dark Knight	2008	9.1	Maggie Gyllenhaal
...

■ Add "roles" information, Solution 3

- Have a **new separate table** "roles", where we specify which actor is cast for which movie

CREATE TABLE roles(title TEXT, actor TEXT);

Much better (and if we wanted, we could now also easily add a third column with the role), but there still is a problem!

title	actor
The Shawshank Redemption	Morgan Freeman
The Shawshank Redemption	Tim Robbins
The Dark Knight	Morgan Freeman
The Dark Knight	Christian Bale
The Dark Knight	Maggie Gyllenhaal
...	...

■ Add "roles" information, Solution 3 problem

- A title does not uniquely identify a movie

For example, there are three movies with the title "All Quiet on the Western Front" (1930, 1979, 2022)

- Similarly, a name does not uniquely identify a person

For example, there are two actresses with the name "Anne Hathaway" (one born 1982, and one born 1556)

- In general, "names" are good for human communication, but they are not good as identifiers because they are usually not unique

The URLs of Wikipedia pages, which often have a suffix for disambiguation, are a good example, see the two links above

■ Add "roles" information, Solution 4 ... **that's the one!**

- In our original table "movies", add a column with a unique ID for each table ... for example, the Wikidata ID

```
CREATE TABLE movies(id TEXT, title TEXT,  
                     year INTEGER, score REAL);
```

- Have a table "persons", with a unique ID for each person and while we are at it, let's also have the birth date

```
CREATE TABLE persons(id TEXT, name TEXT,  
                     birth_date DATE);
```

- Have a table "roles" like in Solution 3, but now using pairs of movie ID and person ID (instead of title and name)

```
CREATE TABLE roles(movie_id TEXT, person_id TEXT);
```

- Add "roles" information, Solution 4 continued
 - Here are some example rows for each of the tree tables "movies", "persons", and "roles"

movies

id	title	year	rating
Q172241	The Shawshank Redemption	1994	9.3
Q163872	The Dark Knight	2008	9.1
Q25188	Inception	2010	8.8
Q190050	Fight Club	1999	8.8

persons

id	name	birth_date
Q48377	Morgan Freeman	1937-06-01
Q95048	Tim Robbins	1958-10-16
Q45772	Christian Bale	1974-01-30
Q202381	Maggie Gyllenhaal	1977-11-16

roles

movie_id	person_id
Q172241	Q48377
Q172241	Q95048
Q163872	Q48377
Q163872	Q45772
Q163872	Q202381

■ Primary key and foreign key

- A column introducing a unique ID for each row is called the **primary key** of that table

For example, the column "id" of table "movies" is the primary key for that table, and the column "id" of table "persons" is the primary key of that table

Note: the primary key can also consist of several columns, but we don't need that for now (and you don't need it for ES3)

- When a primary key is used in a table other than the one where it is introduced, it is called a **foreign key** there

For example, in the table "roles", both "movie_id" and "person_id" are foreign keys

■ Primary key and foreign key, use in an RDBMS

- In a typical RDBMS (including SQLite3), you can explicitly specify when a column is a primary key or a foreign key:

```
CREATE TABLE movies(id TEXT PRIMARY KEY,  
                     title TEXT, year INTEGER, score REAL);
```

```
CREATE TABLE persons(id TEXT PRIMARY KEY,  
                      name TEXT, birth_date DATE);
```

```
CREATE TABLE roles(  
    movie_id TEXT REFERENCES movies(id),  
    person_id TEXT REFERENCES persons(id) );
```

- The RDBMS will then make various checks: that a primary key is specified for each row, that it is unique for each row, and that each foreign key indeed exists in the table it references

■ Possible relationships between tables

- There are four kinds of relationships between two tables **A** and **B** sharing keys (directly or via other tables)

1:1 each record in A can be associated with exactly one record in table B

1:N each record in A can be associated with multiple records in table B, but each record in table B can be associated with only one record in table A

N:1 like 1:N but the other way round

N:M each record in table A can be associated with multiple records in table B, and each record in table B can be associated with multiple records in table A

■ Relationship between the tables on slide 23

- First, let's recall the schema of the three tables (abbreviated)

movies(movie_id, title, year, score)

persons(person_id, name, birth_date)

roles(movie_id, person_id)

- Relationships

movies to roles: $1 : N$

persons to roles: $1 : N$

movies to persons (via roles): $N : M$

Note: this is typical, the relation via primary key and foreign key is often 1:N (and cannot be N:M), and the relation via two foreign keys is often N:M

■ Basic structure of a SQL query

- The basic form of a SQL query is as follows:

```
SELECT [comma-separated list of column names]  
FROM [comma-separated list of tables]  
WHERE [Boolean expression, which can use all column names];
```

- On the following slide, we explain the semantics of such SQL queries **by example**, by incrementally constructing the following example query:

```
SELECT movies.title, persons.name  
FROM movies, persons, roles  
WHERE roles.person_id = persons.id  
AND roles.movie_id = movies.id;
```

In Lecture 4, we will define the semantics more rigorously

■ The FROM clause

- Example query (without WHERE and selecting all columns):

`SELECT * FROM movies, persons, roles;`

- Let T_1, \dots, T_k be the list of tables in the FROM clause; then the query above computes the **cross-product** $T_1 \times \dots \times T_k$, which is the set of all (row_1, \dots, row_k) , where $row_1 \in T_1, \dots, row_k \in T_k$

Intuitively: all possible combinations when taking one row from each of the k tables → examples on the next two slides

- Let $\#cols$ be the number of columns of a table and let $\#rows$ be the number of rows of a table, then

$$\#cols(t_1 \times \dots \times t_k) = \#cols(t_1) + \dots + \#cols(t_k) \quad \text{sum}$$

$$\#rows(t_1 \times \dots \times t_k) = \#rows(t_1) \cdot \dots \cdot \#rows(t_k) \quad \text{product}$$

a.k.a. Cartesian product

■ Let's first understand the cross-product for **sets**

- For k arbitrary sets S_1, \dots, S_k the cross-product is defined as:

$$S_1 \times \dots \times S_k := \{ (s_1, \dots, s_k) : s_1 \in S_1, \dots, s_k \in S_k \}$$

- Example with two sets:

$$S_1 = \{ A, B \}, S_2 = \{ 1, 2, 3 \} \quad |S_1| = 2, |S_2| = 3, |S_1 \times S_2| = 6$$
$$S_1 \times S_2 = \{ (A, 1), (B, 1), (A, 2), (B, 2), (A, 3), (B, 3) \}$$

- Example with three sets: $|S_1| = 2, |S_2| = 3, |S_3| = 3, |S_1 \times S_2 \times S_3| = 18$

$$S_1 = \{ A, B \}, S_2 = \{ 1, 2, 3 \}, S_3 = \{ x, y, z \}$$

$$S_1 \times S_2 \times S_3 = \{$$
$$(A, 1, x), (B, 1, x), (A, 2, x), (B, 2, x), (A, 3, x), (B, 3, x),$$
$$(A, 1, y), (B, 1, y), (A, 2, y), (B, 2, y), (A, 3, y), (B, 3, y),$$
$$(A, 1, z), (B, 1, z), (A, 2, z), (B, 2, z), (A, 3, z), (B, 3, z) \}$$

■ Example for the cross-product of three tables

- We take our three tables "movies", "persons", and "roles"


Compared to [slides 23 and following](#), we omit some rows and columns for presentation reasons, otherwise the cross-product does not fit on a single slide → see the next slide

id	title
Q172241	The Shawshank Redemption
Q163872	The Dark Knight

id	name
Q48377	Morgan Freeman
Q202381	Maggie Gyllenhaal

movie_id	person_id
Q172241	Q48377
Q163872	Q48377
Q163872	Q202381

SQL Basics 5/12



id	title	id	name	movie_id	person_id
Q172241	The Shawshank Redemption	Q48377	Morgan Freeman	Q172241	Q48377
Q163872	The Dark Knight	Q48377	Morgan Freeman	Q172241	Q48377
Q172241	The Shawshank Redemption	Q202381	Maggie Gyllenhaal	Q172241	Q48377
Q163872	The Dark Knight	Q202381	Maggie Gyllenhaal	Q172241	Q48377
Q172241	The Shawshank Redemption	Q48377	Morgan Freeman	Q163872	Q48377
Q163872	The Dark Knight	Q48377	Morgan Freeman	Q163872	Q48377
Q172241	The Shawshank Redemption	Q202381	Maggie Gyllenhaal	Q163872	Q48377
Q163872	The Dark Knight	Q202381	Maggie Gyllenhaal	Q163872	Q48377
Q172241	The Shawshank Redemption	Q48377	Morgan Freeman	Q163872	Q202381
Q163872	The Dark Knight	Q48377	Morgan Freeman	Q163872	Q202381
Q172241	The Shawshank Redemption	Q202381	Maggie Gyllenhaal	Q163872	Q202381
Q163872	The Dark Knight	Q202381	Maggie Gyllenhaal	Q163872	Q202381

■ The WHERE clause

- Example query

```
SELECT *  
FROM persons, movies, roles  
WHERE roles.person_id = persons.id  
AND roles.movie_id = movies.id;
```

- Since different tables can use the same column name, columns are disambiguated as follows:

<table name>.<column name>

- A query like the above computes a table with all rows from the cross-product (determined by the FROM clause), for which the expression in the WHERE clause evaluates to **true**

SQL Basics 7/12

<i>movies</i>		<i>person</i>		<i>roles</i>	
id	title	id	name	movie_id	person_id
Q172241	The Shawshank Redemption	Q48377	Morgan Freeman	Q172241	Q48377 ✓
Q163872	The Dark Knight	Q48377	Morgan Freeman	Q172241	Q48377 ✗
Q172241	The Shawshank Redemption	Q202381	Maggie Gyllenhaal	Q172241	Q48377 ✗
Q163872	The Dark Knight	Q202381	Maggie Gyllenhaal	Q172241	Q48377 ✗
Q172241	The Shawshank Redemption	Q48377	Morgan Freeman	Q163872	Q48377 ✗
Q163872	The Dark Knight	Q48377	Morgan Freeman	Q163872	Q48377 ✓
Q172241	The Shawshank Redemption	Q202381	Maggie Gyllenhaal	Q163872	Q48377 ✗
Q163872	The Dark Knight	Q202381	Maggie Gyllenhaal	Q163872	Q48377 ✗
Q172241	The Shawshank Redemption	Q48377	Morgan Freeman	Q163872	Q202381 ✗
Q163872	The Dark Knight	Q48377	Morgan Freeman	Q163872	Q202381 ✗
Q172241	The Shawshank Redemption	Q202381	Maggie Gyllenhaal	Q163872	Q202381 ✗
Q163872	The Dark Knight	Q202381	Maggie Gyllenhaal	Q163872	Q202381 ✓

■ The WHERE clause, continued

- Below is the result of our example query

```
SELECT * FROM persons, movies, roles  
WHERE roles.person_id = persons.id  
AND roles.movie_id = movies.id;
```

Understand why this gives us the roles for each movie!

id	title	id	name	movie_id	person_id
Q172241	The Shawshank Redemption	Q48377	Morgan Freeman	Q172241	Q48377
Q163872	The Dark Knight	Q48377	Morgan Freeman	Q163872	Q48377
Q163872	The Dark Knight	Q202381	Maggie Gyllenhaal	Q163872	Q202381

■ The SELECT clause

- Example query:

```
SELECT movies.title, persons.name  
FROM persons, movies, roles  
WHERE roles.person_id = persons.id  
AND roles.movie_id = movies.id;
```

- A query like the above computes a table
 - with all rows from the cross-product (according to the FROM),
 - for which the expression from the WHERE clause is **true**,
 - with exactly the columns specified in the SELECT clause

■ The SELECT clause, continued

- The first table below is the table from [slide 35](#) (with SELECT *), with the columns specified in the SELECT clause highlighted
- Below that is the table with only those columns → this is the final result of our query

id	title	id	name	movie_id	person_id
Q172241	The Shawshank Redemption	Q48377	Morgan Freeman	Q172241	Q48377
Q163872	The Dark Knight	Q48377	Morgan Freeman	Q163872	Q48377
Q163872	The Dark Knight	Q202381	Maggie Gyllenhaal	Q163872	Q202381

title	name
The Shawshank Redemption	Morgan Freeman
The Dark Knight	Morgan Freeman
The Dark Knight	Maggie Gyllenhaal

■ Explanation vs. what an RDBMS actually does

- The previous slides gave a correct explanation of the result of a SQL query of the form

`SELECT ... FROM ... WHERE ...;`

- But that is **not** how a RDBMS actually computes the result

Iterating over the cross-product of many tables and checking the WHERE condition for each row would be too expensive

Note: when there is no WHERE clause, the RDBMS indeed has to compute the full cross-product

- In Lecture 4, we will see a much more efficient way for the typical case where some or all of the terms in the WHERE clause are of the form `some_column = some_other_column`

■ Point of ES3

- For ES3, you do not have to care about how the RDBMS computes the result

The point of ES3 is to understand the basic concepts of this lecture, in particular: to practice database design, get to know SQLite3, and get used to how basic SQL queries work

References

- Original paper on the relational model

E. F. Codd: Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks, [IBM Research Report RF599](#), 1969

- Relevant Wikipedia articles

https://en.wikipedia.org/wiki/Relational_model

https://en.wikipedia.org/wiki/Relational_algebra

https://en.wikipedia.org/wiki/Database_design

<https://en.wikipedia.org/wiki/SQL>

https://en.wikipedia.org/wiki/List_of_relational_database_management_systems

- SQLite3

<https://sqlite.org>

Website

<https://www.sqlite.org/lang.html>

SQL supported