Introduction to Informatics for Students from all Faculties

Prof. Dr. Ingo Scholtes
Chair of Machine Learning for Complex Networks
Center for Artificial Intelligence and Data Science (CAIDAS)
Julius-Maximilians-Universität Würzburg
Würzburg, Germany
ingo.scholtes@uni-wuerzburg.de

Lecture 10
Database Systems
January 14, 2025

Motivation

- filesystems allow applications to **store data in files**
- but: structure/meaning of these files is determined by application/user
- how can we **store information** in a structured, interoperable, and consistent way?
  - customer accounts in a bank
  - hotel reservations
  - posts/media in social media platforms

we use **database systems** to organize and store information and to facilitate fast knowledge extraction by multiple users and applications

- efficient database systems are **key technology** in modern information technology

archive with boxes

image credit: Archivo-FSP, CC BY-SA 3.0

Notes:

- **Lecture L10: Database Systems** 14.01.2025
- **Educational objective:** We introduce basic concepts of database systems. We use an example to study relational database design, introduce the query language SQL and motivate database transactions.
  - Introduction to Relational Databases
  - Relational Database Design
  - Structured Query Language (SQL)
  - Database Transactions
- **Exercise sheet 9** 21.01.2025

Notes:

Database Management Systems

- **database (DB)**: collection of data on a given “mini-world”, a part of the world that is of interest for our database

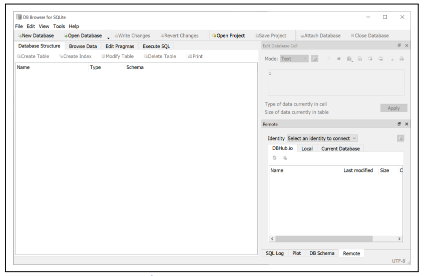
example mini-world

We can consider the **mini-world of movies**, i.e. we want to store information on a movie collection consisting of movies, actors, and directors.

- **database management system (DBMS)**: software that allows to create, manage, and query a DB

key functionality of a DBMS

- allow user to **define databases** (DB)
- allow users to **insert, modify and delete data** into a DB
- support efficient **retrieval of information** from the DB
- guarantee **consistency of data**
- manage **concurrent access** by multiple users
- **user authentication** and access auditing
- support **transactions**



Management interface `DB Browser` for DBMS **SQLite**

Relational Databases

- there are different **types of database management systems**
  - hierarchical or graph-based DBMS
  - object-oriented DBMS
  - relational DBMS
- relational database consists of **relations (= tables)** that consist of **tuples (= table rows)**
- tables have **attributes (= columns)** that must adhere to a given **data type**, e.g. variable-length text, integer or floating point number, date/time, etc.
- how could we store information on a given mini-world in multiple tables?

example table

Movie

Title	Year	Genre
Clockwork Orange	1971	Dystopia
Samsara	2011	Documentary
Woman of Straw	1964	Crime
Highlander	1986	Phantasy

example for a table **Movie** with three attributes **Title, Year, Genre** and four rows, representing four movies

Notes:

- We first briefly introduce some key terms and concepts.
- With the term database we refer to a specific collection of data of interest that we want to store on a so-called “mini-world”. Such a mini-world is one (tiny) part of the world that we want to model with our database, e.g. the customers of a bank, all users and posts of a social media platform, or the movies stored in a movie collection.
- While a database is one specific collection of data, we call the software that allows us to create, manage and query such databases a database management systems (DMBS). A single DMBS can manage many different (maybe thousands of) databases.
- A DBMS is an example of a so-called middleware, i.e. a software that sits between the operating system and other applications that use functions provided by the middleware. As such, the functionality provided by a DBMD goes way beyond what is provided by the filesystem or the OS. It allows us to define databases, insert, modify and delete data, it supports the efficient retrieval of data and guarantees the consistency of data. It also manages concurrent access, allows certain uses to only access some parts of a database and supports transactions (later more on this).
- While we could build some functions based on files, DBMS provide much more convenient functions to efficiently retrieve data from large databases.

Notes:

- There are different types of DBMS that are based on different concepts how the data are organized or stored. For instance we could store data as a graph or network, where nodes store values and links connected those values. Or we could store data in a hierarchical tree structure, where each record is again linked to other records. Or we could store a collection of objects that have clearly defined types.
- In this lecture, we will consider relational databases, which are arguably among the most important DBMS. The key idea is that data is stored in a table format, where each table has multiple columns, and each row represents one tuple (or record). We call such a table also a relation, because each row relates the values of different columns to each other.

Database schema vs. instance

- **relational database schema** describes the "data model" of a database
  - tables
  - attributes of each table
  - data type of each attributes
- **database schema** is usually **created once** when we define a database
- **database instance** refers to data stored in a database at a given point in time
  - collection of all data stored in tables
  - database instance changes whenever contents is updated

simple relational database schema with four attributes

Movie

Title (VARCHAR)	Year (INT)	Genre (VARCHAR)	Length (INT)

instance 1

Movie

Title (VARCHAR)	Year (INT)	Genre (VARCHAR)	Length (INT)
Woman of Straw	1964	Crime	116
Highlander	1986	Phantasy	111

instance 2

Movie

Title (VARCHAR)	Year (INT)	Genre (VARCHAR)	Length (INT)
Clockwork Orange	1971	Dystopia	131
Woman of Straw	1964	Crime	116

Notes:

- We can think of different structures of such table in terms of which columns exist and what are the types of these columns. This is the so-called schema, which originates from Greek $\sigma\chi\eta\mu\alpha$ "schema" translating to "shape".
- The schema of a database is typically created once when the database is created and it is changed very rarely (if at all). For a good schema, it should not be neccessary to change it as new data is stored.
- A database instance refers to the actual tables stored in a database, which are consistent with a schema. So when we update data in a database, the database instance changes but the schema remains the same.

A (badly designed) relational database

consider a database for a **mini-world of movies**, where you want to store information on the title, year, length and genre of movies, as well as on directors and actors.

relational database

Movie

Title	Year	Genre	Length	Director	Actor
Clockwork Orange	1971	Dystopia	131	Stanley Kubrick	Malcolm McDowell
Samsara	2011	Documentary	102	Ron Fricke	NULL
Woman of Straw	1964	Crime	116	Basil Dearden	Gina Lollobrigida
Woman of Straw	1964	Crime	116	Basil Dearden	Sean Connery
Highlander	1986	Phantasy	111	Russell Mulcahy	Sean Connery
Highlander	1986	Phantasy	111	Russell Mulcahy	Christopher Lambert

question

do you see any problems with this relatioal database schema consisting of a single table?

Notes:

- How can we design a database schema for a specific purpose? Let us consider an example of a DB for a mini-world of movies, where we want to store information on movies, actors and directors.
- A simple solution would be to define a single table that holds all of these information.
- Is this a meaningful approach?

Good relational database design?

- during **database design** we determine tables and their schemas
- good database design ensures that attributes are **atomic**, such that we can easily query data (e.g. first- or lastname)
- what if we want to add multiple actors to a single movie?
  - repeated `Actor` attributes?
  - multiple rows with different actors?
- good database design tries to **eliminate redundancy**, i.e. information that is stored multiple times
  - uncontrolled redundancy leads to **data inconsistency**
  - any unavoidable redundancy must be **managed and controlled by DBMS** to avoid inconsistencies
- how can we design a **good database schema**?

atomic attributes

attribute `Director` is non-atomic as it consists of (divisible) First- and Lastname

Movie

Title	Director
Clockwork Orange	Stanley Kubrick

repeated attributes

`Actor1` and `Actor2` are repeated

Movie

Title	Actor1	Actor2

redundancy

`Title` and `Year` are redundant

Movie

Title	Year	Actor
Highlander	1986	Connery
Highlander	1986	Lambert

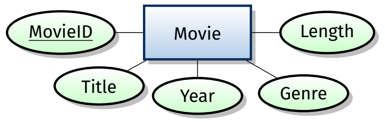
ER model: Entities

- we can use **entity-relationship models** to systematically design database schema
- we use **ER-diagrams** to visualize ER models
- we use **entities** to model specific "things of interest" in our mini-world

example: Entities in movie mini-world

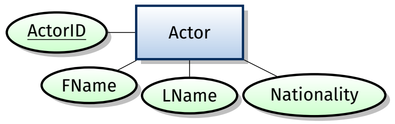
- **MOVIE**s
- **ACTOR**s
- **DIRECTOR**s

- each entity can have multiple **attributes**
- how can we **uniquely identify** a given entity?
- **key attributes** must have unique values across all entities of a given type



entity **Movie** with four attributes

(+ key attribute ID)



entity **Actor** with three attributes

(+ key attribute ID)

Notes:

- A key challenge in database design is to determine a meaningful schema that avoids inconsistencies and redundancies and makes it easy to query the data. Let us consider this in our example.
- A first issue is that we have non-atomic attributes, i.e. we have attributes like Director that could be further divided into a first and a last name. Storing data in non-atomic form makes it difficult to query data, e.g. if you want to search for the lastname.
- A second issue is that we may want to add multiple actors to a single movie. There are two ways to do this if we use a single table.
- We could first repeat the attribute actor multiple times, which is a bad idea because this means that we either have to change the schema when we add an actor or we have to add as many columns as we can have actors (thus defining a maximum), where we have a lot of empty cells for movies that do not have that many actors.
- A second solution would be to repeat a row for each actor, which would introduce redundancy. Redundancy is a very bad idea for databases, not only because it is inefficient to store the same data multiple times, but even more important because we could end up changing one value while keeping its copy unchanged. This would lead to inconsistent data, which we want to avoid at all cost!
- Good relational database design thus avoids redundancy. Those redundancies thst we cannot avoid should be declared to the DBMS, so the DBMS can control that no inconsistencies can ever occur.
- But how can we systematically create a good schema for a database that avoids those issues?

Notes:

- The Entity-Relationship (ER) model helps us to systematically design good database schema (especially relational schema). An ER model consists of two different components: entities and relationships. We visualize an ER model via so-called ER diagrams, where specific symbols represent entities and relationships.
- **Entities** model the types of "things" that we store information on. In our example, these could be movies, actors, and directors. We typically use the singular form as entity name. In an ER diagram, entities are typically drawn as rectangular boxes.
- Each entity can have multiple associated **attributes** that store the actual values for a given entity. Attributes are drawn as ovals connected to the associated entity by a line.
- Some of the attributes of an entity can be **key attributes**, which are assumed to be take unique values among all entities of a given type. As an example, there are multiple different movies that have the same name (e.g. "The Fugitive" which was relased in 1947 and in 1993), so we cannot use the name of a movie to unambiguously refer to a specific movie. We could instead assign a unique number (i.e. an identifier or ID) that we define as key. In the example, we could now assign the 1947 version of "The Fugitive" a unique ID 42, while the 1993 version gets a different ID 125.

ER model: Relationships

- in an ER model, we additionally model **relationships** between entities

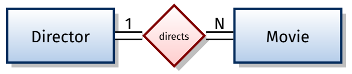
example: relationships in movie mini-world

- DIRECTOR **directs** MOVIE
- ACTOR **plays in** MOVIE

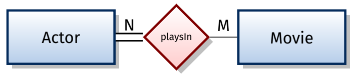
- **cardinality constraint** limit with how many other entities an entity can at most be related
- we can use **participation constraint** to require entity to "participate" in relationship with other entity

example: cardinality and participation constraints

- each movie is directed by **exactly one** director
- each director must direct **at least one** movie
- each movie can have **any number** of actors
- each actor must play in **at least one** movie



One-to-Many relationship **directs** between entities **Director** and **Movie**



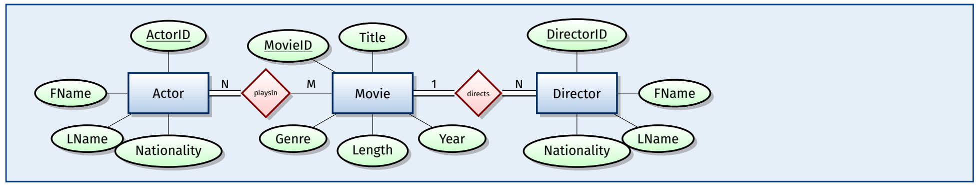
Many-to-Many relationship **playsIn** between entities **Actor** and **Movie**

constraints in ER diagrams

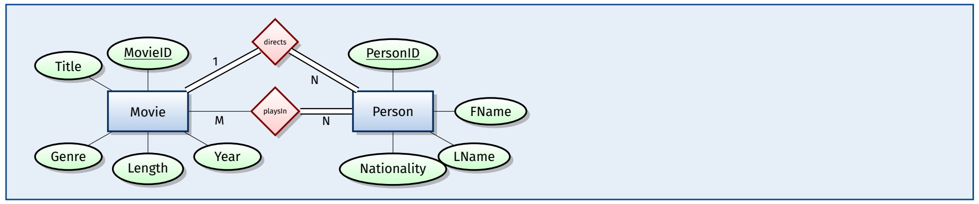
in ER diagrams, cardinality constraints are indicated by numbers (1, N, M), while participation constraints are indicated by a double line. We read cardinality constraints from entity via relation to number (e.g. each "Movie" is "directed by" at most "one" director)

Group exercise

1. Create an Entity-Relationship diagram for a **mini-world of movies**, where you want to store information on the title, year, length and genre of movies, as well as on the names and nationalities of directors and actors.



2. What happens if an actor also directs a movie? Do we need two entities Actor and Director.



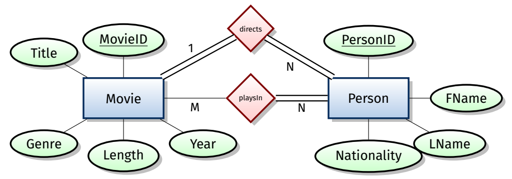
Notes:

- Apart from entities, in the ER model we also model **relationships** between entities. In an ER diagram relationships are typically visualized by a diamond shape, which is connected to the associated entities. In the example above, we see two relationships "directs" and "playsIn" that connect the entities Director and Movie as well as Actor and Movie, respectively.
- A relationship between entities of type A and B can be subject to additional constraints that restrict with how many entities of type B each entity of type A is minimally or maximally related (and vice-versa). Cardinality constraints restrict the maximal number, i.e. we can add a restriction that allows each movie to only have a single director, or that allows one movie to have any number of actors.
- In an ER diagram, we indicate **cardinality constraints** by the small numbers next to a relationship. We read this in the director "Entity, Relationship, Cardinality Constraint", i.e. in the example above each director can direct N (i.e. any number of) movies. In the reverse direction, we read that each movie can be directed by at most director.
- In addition to cardinality constraints, we can also set **participation constraints**, i.e. we specify that each entity must participate in the relationship, i.e. that there must at least be one relation to an entity of the other type. In an ER diagram we indicate this by a **double line** connecting the participating entity and the relationship, i.e. in the example above, each director must direct at least one movie and each movie must have at least one director (together with the cardinality constraint that means each movie must have **exactly one** director). In the example below, each actor must play in at least one movie (and he/she can play in any number of movies), while movies can have any number of actors (including no actors at all, since there is no double line from the entity Movie to the relationship playsIn).

Notes:

Translating ER models to relational schema

- to avoid redundancy, we store **information on different entities in different tables**
  - example: Movie, Person
- to avoid non-atomic attributes, we **store divisible information in multiple attributes**
- **primary key** (underlined attribute) uniquely identifies each row in the table (e.g. via an auto-incremented number)
- we use additional tables to store **relationships between entities**
  - example: directs, playsIn
  - **foreign keys** refer to primary keys in tables Movie and Person
  - combination of (PersonID, MovieID) is primary key in tables directs and playsIn



relational schema implementing ER model

Movie

<u>MovieID</u>	Title	Year	Genre	Length

Person

<u>PersonID</u>	FirstN	LastN	Nationality

directs

<u>PersonID</u>	<u>MovieID</u>

playsIn

<u>PersonID</u>	<u>MovieID</u>

Notes:

- Once we have an ER model of our database, we must translate it to a relational schema. Luckily, this works is a more or less automatic fashion and it does not require any creativity!
- We can simply create one table for each entity and relationship type in or ER model. The attributes of the entities are the columns of our tables. We also use the key attributes as primary keys, which uniquely identify entities.
- For the relationships, we create tables that use foreign keys to refer to the primary keys of entities that participate in the relationship, i.e. the information that a given actor plays in a movie would just be stored by a row that relates the PersonID of the actor to the MovieID of the movie. We could now either introduce a new ID for each of those relations, or we can just use the combination of foreign keys as primary key (since each relation can occur only once).
- Note that for a N:M relation we can just store multiple rows, where each row can relate the same Actor (i.e. a given PersonID) to multiple Movies (i.e. different MovieIDs) and vice-versa.

Example for corresponding database instance

database instance

Movie

<u>MovieID</u>	Title	Year	Genre	Length
1	Clockwork Orange	1971	Dystopia	131
2	Samsara	2011	Documentary	102
3	Woman of Straw	1964	Crime	116
4	Highlander	1986	Phantasy	111

Person

<u>PersonID</u>	FirstN	LastN	Nationality
1	Malcolm	McDowell	England
2	Sean	Connery	Scotland
3	Christopher	Lambert	USA
4	Stanley	Kubrick	England
5	Ron	Fricke	USA
6	Russel	Mulcahy	Australia
7	Basil	Dearden	England

playsIn

<u>PersonID</u>	<u>MovieID</u>
1	1
2	3
2	4
3	4

directs

<u>PersonID</u>	<u>MovieID</u>
4	1
5	2
6	3
7	4

Notes:

Structured Query Language (SQL)

- how can we define, manipulate, and query our database in practice?
- we use **special "programming languages"** to interact with databases
  - data definition language (DDL) to create database schema, define data types, set keys, etc.
  - data manipulation language (DML) to insert, update, and query data
- SQL is **standard language of many DBMS**
  - SQL = Structured Query Language
  - pronounced: "SEQUEL"
  - comprises both DDL and DML
- SQL is a **declarative language**, i.e. we declare **which** data we want rather than **how** DBMS should perform the query

example database query (SQL)

```
SELECT * FROM movie
WHERE YEAR = 1986;
```

SQL-based relational DBMS

- Oracle DBMS
- SAP HANA
- PostgreSQL
- MySQL
- SQLite

SQL DDL: Creating tables

Defining tables

- we can use the `CREATE TABLE` statement to define schema of tables
- `PRIMARY KEY` ensures that attribute value(s) uniquely identify a row
- `FOREIGN KEY` ensures that referenced rows exist in other table

create table Movie with primary key

```
CREATE TABLE Movie (
  MovieID INTEGER,
  Title CHAR(30),
  YEAR INTEGER,
  Genre CHAR(30),
  Length INTEGER,
  PRIMARY KEY (MovieID)
);
```

creates new table *Movie* with following schema

Movie

<u>MovieID</u>	Title	Year	Genre	Length

create table directs with foreign keys

```
CREATE TABLE directs (
  PersonID INTEGER,
  MovieID INTEGER,
  FOREIGN KEY (PersonID)
     REFERENCES Person(PersonID),
  FOREIGN KEY (MovieID)
     REFERENCES Movie(MovieID),
  PRIMARY KEY (PersonID, MovieID)
);
```

creates new table *directs* with following schema

directs

<u>PersonID</u>	<u>MovieID</u>

Notes:

Notes:

SQL DML: Insert, update and delete data

Manipulating data

- we can use `INSERT INTO`, `UPDATE` and `DELETE FROM` to manipulate rows
- `WHERE` can be used to specify which rows should be updated/deleted

example

- add row to table movie

```
INSERT INTO Movie(MovieID,
  YEAR, Title, Genre,
  Length)
  VALUES (1, 2001,
    "The Lord of the Rings",
    "Phantasy", 228);
```

- or equivalently

```
INSERT INTO Movie
  VALUES (1,
    "The Lord of the Rings",
    2001, "Phantasy", 228);
```

example

- change title of movie

```
UPDATE Movie
SET Title =
"The Fellowship of the Ring"
WHERE MovieID = 1;
```

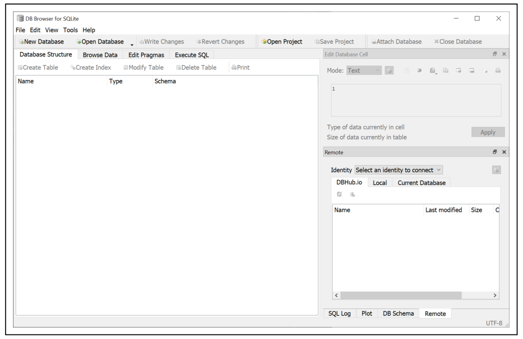
example

- delete all movies released in 2001

```
DELETE FROM Movie
WHERE YEAR = 2001;
```

Practice Session

- we introduce the lightweight SQL-based relational DBMS **SQLite**
- we use the **DB Browser for SQLite** to define the movie database using SQL DDL statements
- we use SQL DML statements to **insert and modify data into our movie DB**



practice session

see directory 10-01 in `gitlab` repository at

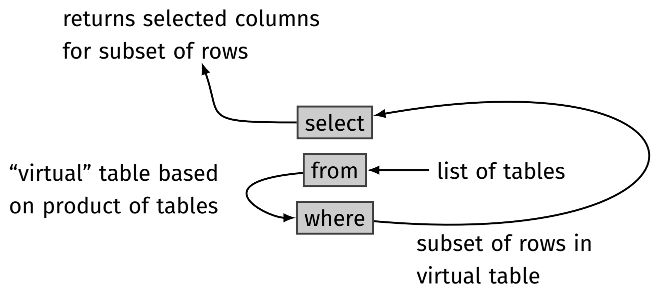
→ https://gitlab2.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_wise_infhaf_notebooks

Notes:

Notes:

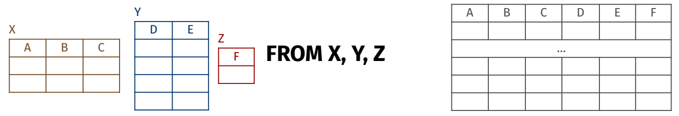
Using SQL to query data

- simple SQL query consists of `SELECT` statement with `FROM` clause and optional `WHERE` clause
- for this lecture we ignore optional `GROUP` and `HAVING` clauses that can be used to calculate aggregates over grouped rows
- simple SQL queries with `SELECT`, `FROM` and `WHERE` are processed as follows:

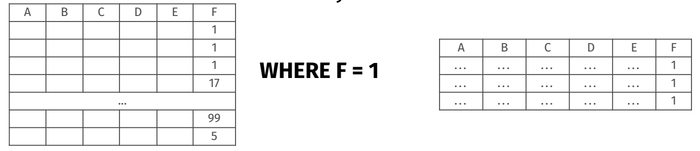

returns selected columns for subset of rows
select
"virtual" table based on product of tables
from
list of tables
where
subset of rows in virtual table

SQL: Evaluation of a simple query

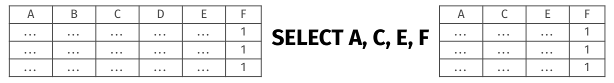
1. compute **product of tables** listed in `FROM` clause



2. eliminate rows that do not satisfy `WHERE` clause



3. return rows for columns listed in `SELECT` clause



Notes:

Notes:

SQL Query Examples

Querying data

- we use SELECT statement to **query a database**
- column list after SELECT can be used to reduce result certain columns
- FROM can be used to define (virtual) table from which data is queried, possibly combining rows from multiple tables
- WHERE can be used to specify condition that selected rows must satisfy

example: query with WHERE clause

- return Title and Year of all Phantasy movies before 2001

```
SELECT Title, YEAR FROM Movie
WHERE Genre = "Phantasy"
AND YEAR < 2001;
```

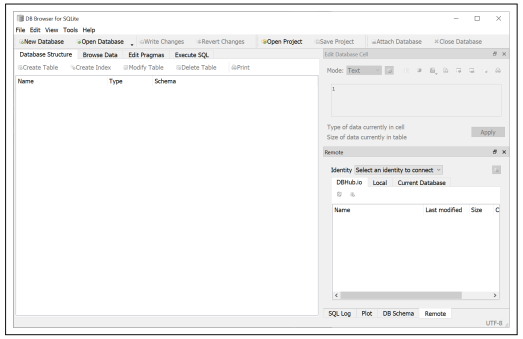
example: query that joins multiple tables

- join Movie, directs, Person tables and select last name of all directors of Phantasy movies

```
SELECT Person.LName FROM
Movie, directs, Person
WHERE Genre = "Phantasy"
AND Movie.MovieID =
  directs.MovieID
AND Person.PersonID =
  directs.PersonID;
```

Practice Session

- we use SQL queries to retrieve data from our database
- we show how we can use the SELECT statement to join data from multiple tables
- we demonstrate how the JOIN keyword simplifies the joining of data from multiple tables



practice session

see directory 10-02 in `gitlab` repository at

→ https://gitlab2.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_wise_infhaf_notebooks

Notes:

Notes:

From database queries to transactions

- we finally consider a relational database that holds data on **customer accounts in a bank**
- wire transfer of EUR 5000 between two accounts can be implemented as **sequence of two SQL queries**
- **what could possibly go wrong?**
  1. concurrent qery of database (after first, but before second `UPDATE` operation)
  2. DBMS could crash after first UPDATE statement

example queries

```
UPDATE account SET
  balance = balance - 5000
  WHERE IBAN = "DE70 32 4134 1232 1231";
UPDATE account SET
  balance = balance + 5000
  WHERE IBAN = "DE70 32 3521 4211 1124";
```

transactions

We call a sequence of database operations (e.g. insert, deletion, modification, retrieval) that form a **logical unit** a **transaction**.

ACID property and SQL Transactions

- for transactions we must guarantee **ACID property**
  - A **A**tomic = indivisible, i.e. either all operations succeed or none
  - C **C**onsistent = integrity of data is not violated
  - I **I**solated = concurrently executed transactions do not have side-effects
  - D **D**urable = result of successful transaction is stored permanently, i.e. it will not be reverted
- in SQL we use `BEGIN` and `COMMIT` to **group queries belonging to transaction**
- any concurrent query executed before `COMMIT` see the DB state *before* the transaction was started
- if any operation within transaction fails, all changes within that transaction will be rolled back

example transaction

```
BEGIN;
UPDATE account SET
  balance = balance - 5000
  WHERE accNr = "DE70 32 4134 1232 1231";
UPDATE account SET
  balance = balance + 5000
  WHERE accNr = "DE70 32 3521 4211 1124";
COMMIT;
```

Notes:

Notes:

In summary ...

- we motivated **database management systems**, which are implemented on top of file systems
- we considered key concepts of **relational database design**
- we introduced the **query language SQL** and showed how we can use it to query information in a relational database
- we motivated the need for **database transactions** and introduced ACID properties

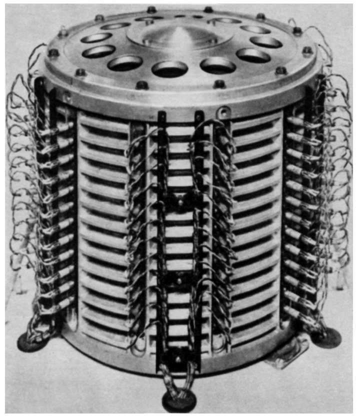

drum memory of ZAM-41 computer (ca. 1961)

image credit: Public Domain, Wikipedia Commons

Self-study questions

1. Explain the components of a relational database.
2. Give an example for a database schema that leads to redundancy and potentially inconsistencies.
3. Give an example for a table schema with a non-atomic attribute.
4. How can we avoid redundancy in a relational database schema?
5. Given a simple ER model for the mini-world of a company, storing information on employees and projects.
6. Explain the difference between a primary key and a foreign key?
7. Explain how we can represent an N:M relationship between entities in a relational table.
8. Give an example for an ER model with a One-to-One (1:1) relationship.
9. Explain the difference between cardinality and participation constraints.
10. How can we merge information on related entities using SQL queries?
11. Explain the challenges that can occur if multiple database queries belong to a transaction.
12. What are the ACID properties?
13. How can we define database transactions in SQL.

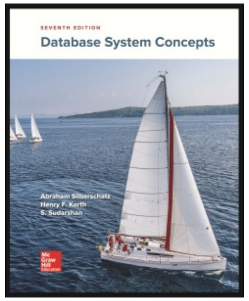
Notes:

Notes:

References and credits

reading list

- Abraham Silberschatz, Henry F. Korth, S. Sudarshan: **Database System Concepts**, 7th Edition, McGraw Hill Education with online material
  → https://www.db-book.com/
- SQL Tutorial
  → https://sqlzoo.net/wiki/SQL_Tutorial
- W3Schools SQL tutorial
  → https://www.w3schools.com/sql/sql_intro.asp
- Online SQL Interpreter
  → https://www.db-book.com/university-lab-dir/sqljs.html



slides reuse material kindly provided by

- **Prof. Dr. Michael Böhlen**
  University of Zürich, Switzerland
- **Prof. Dr. Johann Gamper**
  Free University of Bozen-Bolzano, Italy

Notes: