universität freiburg

Databases and Information Systems WS 25/26

Lecture 4: Query planning basics

November 11, 2025

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

Overview of this lecture

- Organizational
  - Your experiences with ES3 — Database Design
  - Please tell your tutor … — which feedback you want
- Contents
  - Basic operations — Projection, Selection, Cartesian product
  - Query planning — Finding the optimal sequence of operations

ES4: Implement your own very simple database engine; provide three different sequences of operations for a given SQL query; measure their running time and check the correctness of the results against SQLite3

Experiences with ES3 1/2

- Excerpts from your feedback [somewhat representative]

  "Loved the lecture, it made me immediately want to start doing the exercise sheet … was relatively easy but still took some time"

  "Fun exercise + nice way to introduce DB design and queries"

  "Very good learning experience and helped me get used to SQL"

  "It took a while to get the data sorted into different tables, but it was a satisfying process … lecture gave a good introduction"

  "The lecture nicely refreshed my knowledge on databases"

  "Since I had some SQL back in school it was pretty easy"

  "Useful exercise even though creating the tables was annoying"

  "Exhausting but very rewarding at the end when it all worked"

Experiences with ES3 2/2

- What struck you about Richard Hipp [the creator of SQLite3]

  "His website ... it looks like SQLite feels"

  "This guy had no idea what he was doing half of the time, it's crazy how good his product was that big companies were just offering him opportunities and he just went with the flow"

  "MCDC test coverage (~ tests have to cause each branch at machine code level to be taken), which took him a year of 12 hour days every single day"

  "Billions of unit tests, which run for 3 days prior to a release on all kinds of machines (old and new, different OSs, architectures etc.)"

  "Richard Hipp has almost zero dependencies for SQLite (except for C-Compiler and libc), that he didnt write himself; he wrote his own parsers, version control systems, etc."

Basic operations 1/14

- From SQL queries to operations
  - In Lecture 3, we have learned by example how SQL queries of the form `SELECT … FROM … WHERE …;` work
  - Today we will learn how an RDBMS actually executes such SQL queries, in a nutshell:

    1. Internally, an RDBMS implements operations on tables

    We will see **three** operations today (more in Lecture 5), and it will be your task in ES4 to implement them

    2. A SQL query is translated to a sequence of these operations

    This is called query planning … for ES4, do this manually

    3. The sequence of operations is then executed

    This is called query processing … you code this yourself in ES4

Basic operations 2/14

- Basic operations
  - The following slides will introduce three basic operations, called projection, selection, and Cartesian product

    We already introduced them informally in the last part of Lecture 3, but today we will define them mathematically

    We use "Cartesian product" and "cross-product" synonymously
  - The input is one or two tables for each operation, and the output is always a single table
  - Before providing the definitions, let us first repeat the definition of a table from Lecture 3 and introduce some basic formalism

Basic operations 3/14

- Recap of the definition of a table ... from Lecture 3, Slide 8
  - We define a table as a three-tuple (C, D, R), where
    - $C = (c_1, \ldots, c_k)$ are the column names, one per column
    - $D = (D_1, \ldots, D_k)$ are the domains, one per column
    - R is a multiset of tuples $(r_1, \ldots, r_k)$, where $r_j \in D_j$
  - Remarks:
    - We have seen many examples of tables in Lecture 3
    - We have also seen what a multiset is ... Lecture 3, slides 9+10
    - We often call a tuple from R a "row", hence the R

Basic operations 4/14

- Basic notation
  - The following notation will make the definitions on the remaining slides simpler and more intuitive to read
  - Let $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_m)$ be two tuples

    Then we denote by (X, Y) the concatenation of the two tuples, namely $(x_1, \ldots, x_n, y_1, \ldots, y_m)$
  - Let $r = (r_1, \ldots, r_k)$ be a tuple / row of a table with k columns

    Then we denote by $r[j]$ the value r_j in the j-th column

    We start the numbering of the columns with 1, not 0

Basic operations 5/14

- Projection
  - Signature: **project**(T, cols, distinct) or $\Pi_{cols,\ distinct}(T)$, where
    - T = (C, D, R) is an arbitrary table with k columns
    - cols is a sequence $(j_1, \ldots, j_m)$ of indices from $\{1, \ldots, k\}$
    - distinct is either true or false
  - Result: a table T' = (C', D', R'), where
    - $C' = (c_{j1}, \ldots, c_{jm})$ and $D' = (D_{j1}, \ldots, D_{jm})$
    - $R' = \{ (r_{j1}, \ldots, r_{jm}) : r = (r_1, \ldots, r_k) \in R \}$
    - If distinct is true, remove all duplicates from R'
      (that is, turn the potential multiset into the corresponding set)

Basic operations 6/14

- Projection, example
  - SQL:

```
CREATE TABLE people(
   id TEXT, first TEXT, last TEXT);

SELECT DISTINCT first, last
FROM T;
```

  - Operation:

```
T= people

project(T, (2, 3), true)
```

1	2	3
id	**first**	**last**
Q36301	Anne	Hathaway
Q237251	Anne	Hathaway
Q106514	River	Phoenix
Q185140	Joaquin	Phoenix

first	**last**
Anne	Hathaway
River	Phoenix
Joaquin	Phoenix

Basic operations 7/14

- Selection
  - Signature: **select**(T, φ) or $\boldsymbol{\sigma}_{\varphi}(T)$, where
    - T = (C, D, R) is a table and $\varphi: D_1 \times \ldots \times D_k \rightarrow \{ \text{true}, \text{false} \}$ is a function that yields true or false for a given row
  - Result: a table T' = (C, D, R'), where
    - $R' = \{ r \in R : \varphi(r) = \text{true} \}$
  - Note:
    - This operation is traditionally called "select", but it's expressed in the WHERE clause, not in the SELECT clause

      see the example on the next slide

Basic operations 8/14

- Selection, example
  - SQL:

```
CREATE TABLE people(
   id TEXT, first TEXT, last TEXT);

SELECT * FROM people
WHERE first = 'Anne';
```

  - Operation:

```
T = people

select(T, r → r[2] = "Anne")
```

id	first	last
Q36301	Anne	Hathaway
Q237251	Anne	Hathaway
Q106514	River	Phoenix
Q185140	Joaquin	Phoenix

id	first	last
Q36301	Anne	Hathaway
Q237251	Anne	Hathaway

Basic operations 9/14

- Basic operation: Cartesian product
  - Signature: **cartesian-product**(T_1, T_2) or $T_1 \times T_2$, where
    - $T_1 = (C_1, D_1, R_1)$ and $T_2 = (C_2, D_2, R_2)$ are two tables
  - Result: a table T with, where
    - $C = (C_1, C_2)$ and $D = (D_1, D_2)$
    - $R = R_1 \times R_2 = \{ (r_1, r_2) : r_1 \in R_1, r_2 \in R_2 \}$
  - Note:
    - $|C| = |C_1| + |C_2|$
    - $|R| = |R_1| \cdot |R_2|$

Basic operations 10/14

- Cartesian product, example
  - SQL:

```
CREATE TABLE gender(
   id INT, what TEXT);
CREATE TABLE handed(
   id INT, what TEXT);

SELECT *
FROM gender, handed;
```

  - Operation:

T_1 = gender
T_2 = handed

cartesian-product(T_1, T_2)

id	what
Q237251	female
Q6581097	male
Q111918716	other

id	what
Q789447	lefty
Q3039938	righty

id	what	id	what
Q237251	female	Q789447	lefty
Q6581097	male	Q3039938	righty
Q111918716	other	Q789447	lefty
Q237251	female	Q3039938	righty
Q6581097	male	Q789447	lefty
Q111918716	other	Q3039938	righty

Basic operations 11/14

- Translation
  - With the three basic operations **project**, **select**, and **cartesian-product**, we can translate all SQL queries of the kind

    ```
    SELECT [comma-separated list of column names]
    FROM [comma-separated list of tables]
    WHERE [Boolean expression on column names];
    ```

    to a sequence of operations that compute the query result
  - On the next two slides, we will see three ways of visualizing a sequence of operations
    - as a sequence of operations
    - as a single expression
    - as a tree

Basic operations 12/14

- Translation to a sequence of operations
  - SQL:

```
CREATE TABLE movies(id, title, year, score);
CREATE TABLE persons(id, name, birth_date);
CREATE TABLE roles(movie_id, person_id);

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

  - Sequence of operations:

T_1 = cartesian-product(movies, persons)
T_2 = cartesian-product(T_1, roles)
T_3 = select(T_2, r → r[1] = r[8] ∧ r[5] = r[9])
T_4 = project(T_3, (2, 6), false)

Basic operations 13/14

- Translation to a single nested expression
  - SQL:

```
CREATE TABLE movies(id, title, year, score);
CREATE TABLE persons(person_id, name, birth_date);
CREATE TABLE roles(movie_id, person_id);

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

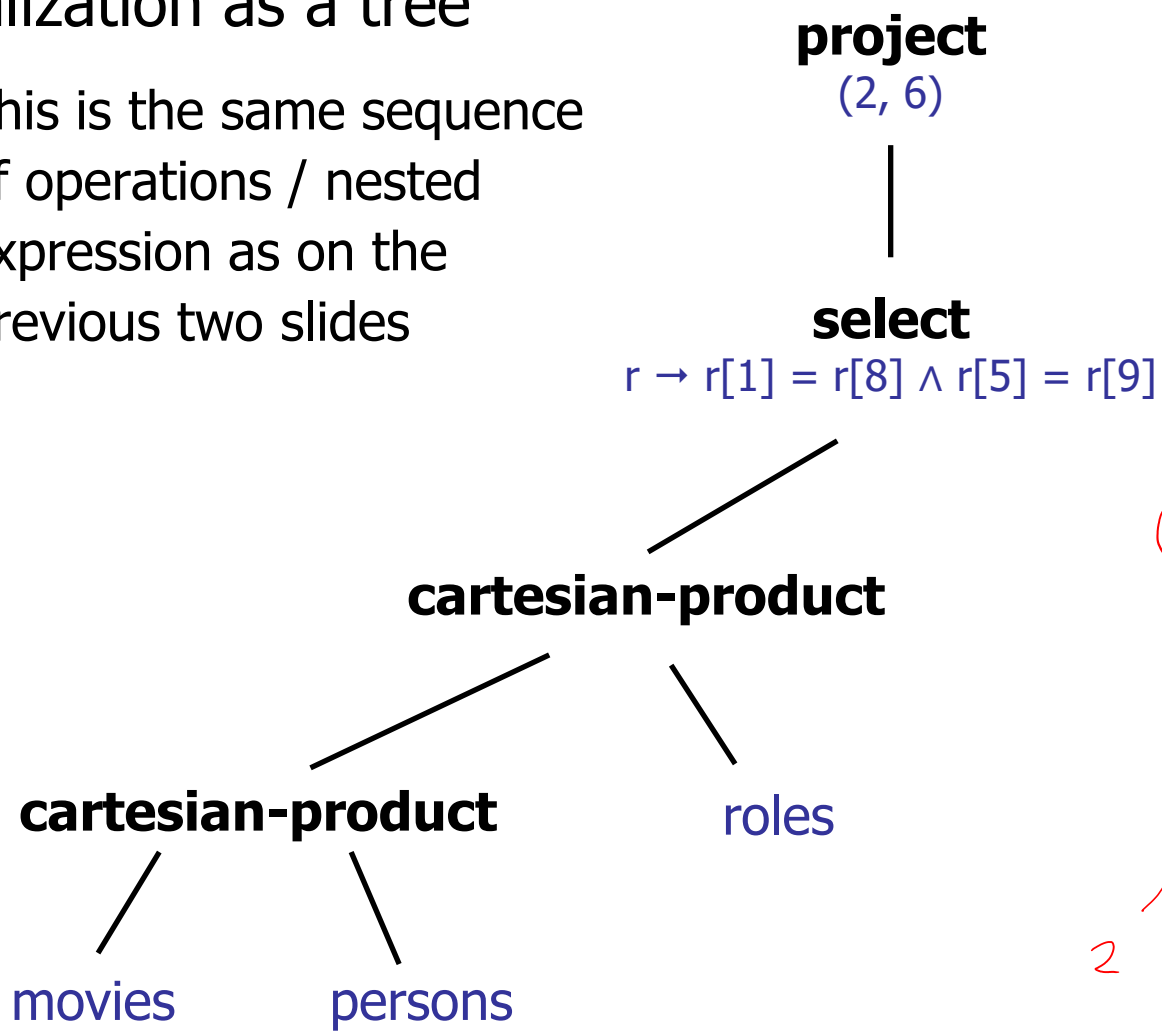
  - Single nested expression:

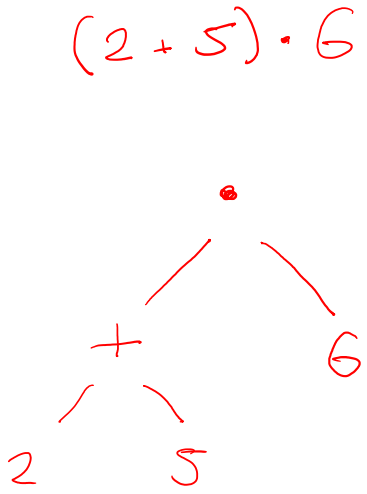
project(select(cartesian-product(cartesian-product(movies, persons), roles), r → r[1] = r[8] ∧ r[5] = r[9]), (2, 6))

Note: this looks scary, but isn't really more difficult than parsing and evaluating an expression like (((3 + 4) · 4) – 3 + 8 · 4) – 30 / 2

Basic operations 14/14

- Visualization as a tree
  - This is the same sequence of operations / nested expression as on the previous two slides





Query planning 1/12

- Motivation
  - A single SQL query can have many different sequences of operations that compute the result for that query

    Regarding the result, these sequences are all equivalent (maybe up to permutation of the columns)

    But the **processing time may be vastly different** for different sequences

    Let us first look at a few examples, and then define the **cost** of a sequence of operations

Query planning 2/12

- Multiple equivalent sequences, example 1
  - SQL:

```
CREATE TABLE movies(id, title, year, score);
CREATE TABLE roles(movie_id, person_id);
CREATE TABLE persons(person_id, name, birth_date);

SELECT * FROM movies, persons, roles;
```

  - Sequence 1:

T_1 = cartesian-product(movies, persons)
T_2 = cartesian-product(T_1, roles)

  - Sequence 2:

T_1 = cartesian-product(movies, roles)
T_2 = cartesian-product(T_1, persons)

Which sequence is better?

(assuming real data for the tables)

Query planning 3/12

- Multiple equivalent sequences, example 2
  - SQL:

    CREATE TABLE movies(id, title, year, score);
    CREATE TABLE roles(movie_id, person_id);

    SELECT * FROM movies, roles WHERE score >= 8.0;

  - Sequence 1:

    T_1 = cartesian-product(movies, roles)
    T_2 = select(T_1, r → r[3] ≥ 8.0)

  - Sequence 2:

    T_1 = select(movies, r → r[3] ≥ 8.0)
    T_2 = cartesian-product(T_1, roles)

Which sequence is better?

(assuming real data for the tables)

Query planning 4/12

- Multiple equivalent sequences, example 3
  - SQL:

    CREATE TABLE movies(id, title, year, score);
    CREATE TABLE roles(movie_id, person_id);

    SELECT DISTINCT title FROM movies, roles WHERE score >= 8.0;

  - Sequence 1:

    T_1 = cartesian-product(movies, roles)
    T_2 = select(T_1, r → r[4] ≥ 8.0)
    T_3 = project(T_2, (2), true)

  - Sequence 2:

    T_1 = select(movies, r → r[4] ≥ 8.0)
    T_2 = project(T_1, (2), true)

Which sequence is better?

(assuming real data for the tables)

Query planning 5/12

- How to tell which sequence of operations is the best?
  - Basic idea:

    1. For each sequence, estimate the cost of each operation
    2. Use this to estimate the total cost for each sequence
    3. Pick the sequence with the lowest cost

  - To give a reasonably good estimate of the cost, we need to know the size of the input tables

    But we only know the input sizes for the operations "at the bottom" of the query execution tree … see slide 18

  - We therefore also need an estimate for the size of the result table (number of rows and columns) for each operation

    In real query planning, one uses even more statistics

Query planning 6/12

- Estimates for **project**(T, cols, distinct)
  - Let T have n rows and k cols, and let k' be the number of columns selected in cols
  - Here are some simple estimates:

    cost ~ n · k', rows ~ n, cols = k'
  - Assumptions and reality check:
    - This assumes that we have to iterate over each row of T and copy the k' columns specified by cols to the result
    - If the tables are stored in "row order" (row by row), these are reasonable estimates
    - If the tables are stored in "column order" (column by column), we could just copy "pointers" to the columns

Query planning 7/12

- Estimates for **select**(T, φ)
  - Let T have n rows and k cols
  - Here are some simple estimates:

    cost ~ n · k, rows ~ n/2, cols = k
  - Assumptions and reality check:
    - This assumes that we have to iterate over each row of T, that φ can be evaluated in constant time, and that (rather arbitrarily) it evaluates to true half of the times
    - We could evaluate φ on a number of sample rows to get an estimate of its cost and how often it evaluates to true
    - If φ is of a particular kind, e.g. score ≥ 8.0, and the table is sorted by score, we do not have to iterate over all rows

Query planning 8/12

- Estimates for **cartesian-product**(T_1, T_2)
  - Let T_i have n_i rows and k_i cols
  - Here are some simple estimates:

    cost ~ $n_1 \cdot n_2 \cdot (k_1 + k_2)$, rows = $n_1 \cdot n_2$, cols = $k_1 + k_2$
  - Assumptions and reality check:
    - These are pretty accurate estimates and there is not much to improve because we have to produce the result, and that has size $n_1 \cdot n_2 \cdot (k_1 + k_2)$
    - Good query plans will therefore **avoid** Cartesian products as best as they can; this will be the topic of Lecture 5, which is about so-called "join" operations

Query planning 9/12

- Cost estimates for Example 1
  - SQL:

```
CREATE TABLE movies(id, title, year, score);
CREATE TABLE persons(person_id, name, birth_date);
CREATE TABLE roles(movie_id, person_id);

SELECT * FROM movies, persons, roles;
```

  - Sequence 1:

T_1 = cartesian-product(movies, persons)
T_2 = cartesian-product(T_1, roles)

  - Sequence 2:

T_1 = cartesian-product(movies, roles)
T_2 = cartesian-product(T_1, persons)

Query planning 10/12

- Cost estimates for Example 2
  - SQL:

```
CREATE TABLE movies(id, title, year, score);
CREATE TABLE roles(movie_id, person_id);

SELECT * FROM movies, roles WHERE score >= 8.0;
```

$n_1 \times 4$
$n_2 \times 2$
$n_1 \ll n_2$

  - Sequence 1: TOTAL = $12 n_1 n_2$

T_1 = cartesian-product(movies, roles)
T_2 = select(T_1, r → r[3] ≥ 8.0)

cost ~ $n_1 n_2 \cdot 6$
rows = $n_1 n_2$
cols = 6

cost ~ $n_1 n_2 6$
rows ~ $n_1 n_2 / 2$
cols = 6

  - Sequence 2: TOTAL = $3 n_1 n_2 + 4 n_1$ BETTER

T_1 = select(movies, r → r[3] ≥ 8.0)
T_2 = cartesian-product(T_1, roles)

cost ~ $n_1 \cdot 4$
rows ~ $n_1 / 2$
cols = 4

cost ~ $n_1/2 \cdot n_2 \cdot 6$
rows ~ $n_1 \cdot n_2 / 2$
cols = 6

Query planning 11/12

- Cost estimates for Example 3
  - SQL:

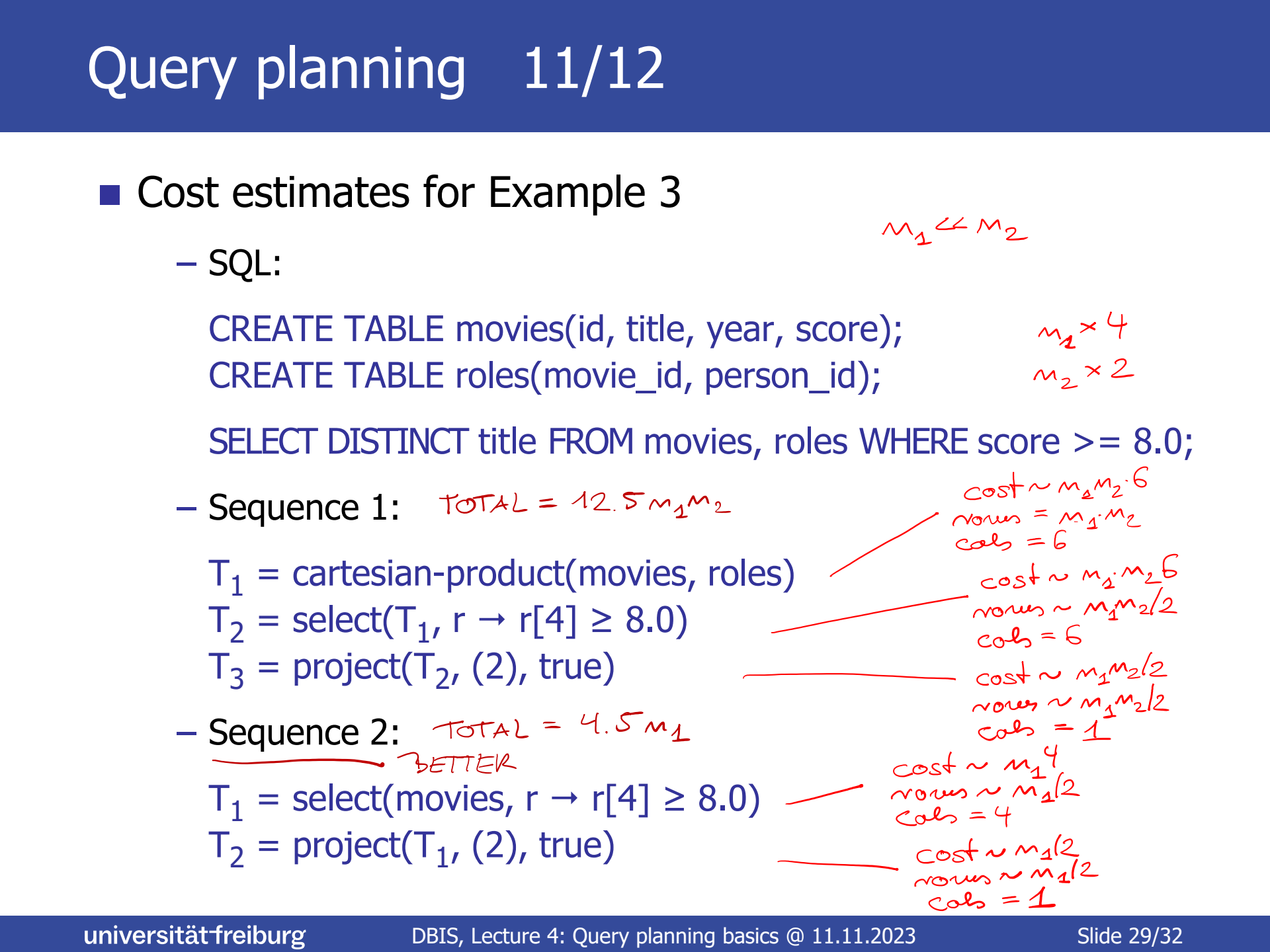
```
CREATE TABLE movies(id, title, year, score);
CREATE TABLE roles(movie_id, person_id);

SELECT DISTINCT title FROM movies, roles WHERE score >= 8.0;
```

  - Sequence 1:

T_1 = cartesian-product(movies, roles)
T_2 = select(T_1, r → r[4] ≥ 8.0)
T_3 = project(T_2, (2), true)

  - Sequence 2:

T_1 = select(movies, r → r[4] ≥ 8.0)
T_2 = project(T_1, (2), true)



Query planning 12/12

- Query plans in SQLite3
  - We can ask SQLite3 for a high-level description of the query plan it's going to use to process a query:

    ```
    EXPLAIN QUERY PLAN
    SELECT DISTINCT title FROM movies, roles WHERE score >= 8.0;
    ```

  - With just EXPLAIN we get a more low level execution plan:

    ```
    EXPLAIN
    SELECT DISTINCT title FROM movies, roles WHERE score >= 8.0;
    ```

    Note: Unlike much of SQL, the output of EXPLAIN is **not** standardized and strongly depends on the internals of the respective system

Exercise Sheet 4

- Please do this exercise sheet
  - The sheet is about implementing what you have learned today
  - Implementing the individual operations will be quite easy
  - Understanding the structure of the code will **not be easy**

    In particular, each operation is a whole **class** (not just a single function) and the objects from that class implicitly form a tree (like the one we have seen on slide 18)

    See the extensive code template we have provided for you
  - The only way to really understand this is by doing the exercise

    If you don't do it now, you will have to do it when preparing for the exam, otherwise you won't be able to solve the resp. tasks

    So take the opportunity and do it now instead of later

References

- Relevant Wikipedia articles

  Basic database operations:
  https://en.wikipedia.org/wiki/Relational_algebra

  Query planning:
  https://en.wikipedia.org/wiki/Query_plan

- SQLite3

  EXPLAIN
  https://www.sqlite.org/lang_explain.html

  EXPLAIN QUERY PLAN
  https://www.sqlite.org/eqp.html