

universität freiburg

Databases and Information Systems WS 25/26

Lecture 5: Joins and more SQL

November 18, 2025

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

Overview of this lecture

■ Organizational

- Your experiences with ES4

Coding a simple RDBMS

■ Contents

- Join operation
- Join processing
- More operations
- More SQL

A new kind of operation

How to compute it

Grouping, ordering, etc.

In particular, subqueries

ES5: Extend your implementation from ES4, in particular by a join operation + improve a query + learn some more SQL

Experiences with ES4 1/2

■ Excerpts from your feedback [somewhat representative]

"I really liked the lecture and the sheet was surprisingly easy"

"Very good exercise + very good frame to work with"

"The code templates are as always top notch!"

"Implementing the operations was easy, but queries took ages"

"Took me 7 hours + taught me that Cartesian products are bad"

"My story with this exercise sheet is one of great frustration, dumbness and amazement in the end ... Exercise 2 very tough"

"Exercise 2 was very hard ... the sheet took me about 9 hours"

"This exercise was extremely frustrating ... it took me nearly 20 hours and I couldn't even finish everything"

Experiences with ES4 2/2

■ What annoys you most about HISinOne

"The load times when clicking through the Studienplaner ... feels like it definitely calculates Cartesian products at every step"

"Definitely that it is so slow ... but on the other hand, this gives us HISinOne hate which is quite entertaining"

"The insane time it takes to load certain pages"

"Everytime I want to do a math lecture, I have to call three people so that they can add the course as an option for CS electives"

"Everything that comes with any old(ish) PHP website"

"In Sep 2024, they took down the grade overview (Notenspiegel) because of mistakes in the graphics ... and they just don't fix it"

"It does not remember my login on my PC, only on my phone"

Join operation 1/5

■ Motivation

- The following pattern is very frequent in a SQL query:
 - two tables T_1 and T_2 in the FROM clause
 - A condition $T_1.col_1 = T_2.col_2$ in the WHERE clause
- For example, in this query from Lecture 4, it occurs **twice**

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

It turns out that it is a good idea to define an operation specifically for this combination ... see next slides

Join operation 2/5

■ Single-column equi-join

- Signature: **join**(T_1, T_2, j_1, j_2) or $T_1 \bowtie_{j_1, j_2} T_2$, where
 - $T_1 = (D_1, C_1, R_1)$ and $T_2 = (D_2, C_2, R_2)$ are two tables
 - $j_1 \in \{1, \dots, |C_1|\}$ and $j_2 \in \{1, \dots, |C_2|\}$ [column indexes]
- Result: a table T with
 - $C = (C_1, C_2)$ and $D = (D_1, D_2)$
 - $R = R_1 \times R_2 = \{ (r_1, r_2) : r_1 \in R_1, r_2 \in R_2, r_1[j_1] = r_2[j_2] \}$

Join operation 3/5

■ Single-column equi-join, example

– SQL:

```
CREATE TABLE profs(  
  prof TEXT, course TEXT);  
CREATE TABLE studis(  
  student TEXT, course TEXT);  
  
SELECT * FROM profs, studis  
  WHERE profs.course = studis.course;
```

– Operation:

$T_1 = \text{profs}, T_2 = \text{studis}$

$\text{join}(T_1, T_2, 2, 2)$

prof	course
Brox	V720
Hutter	V2040
Bast	V1304
Burgard	V2040

student	course
7383436	V1304
3487464	V2040
3344637	V2040
3347343	V2060
5634555	V2040
7537345	S1303

prof	course	student	course
Hutter	V2040	3487464	V2040
Burgard	V2040	3487464	V2040
Hutter	V2040	3344637	V2040
Burgard	V2040	3344637	V2040
Hutter	V2040	5634555	V2040
Burgard	V2040	5634555	V2040
Bast	V1304	7383436	V1304

Join operation 4/5

■ Single-column equi-join, why the name?

- It's called **single-column** because the equality constraint concerns only one column from each of the two tables

We could also define a join operation that specifies multiple columns from each table (same number for each table)

- It's called an **equi-join** because the condition specifies that the values in column j_1 of T_1 and j_2 of T_2 are **equal**

We could also define a join operation with another condition

Note: in a worst case, when we know nothing about the condition in advance, we are back to computing the Cartesian product and evaluating the condition for each row of it

Join operation 5/5

■ Cartesian product vs. Join

– SQL:

```
CREATE TABLE profs(1prof TEXT, 2course TEXT);  
CREATE TABLE studis(3student TEXT, 4course TEXT);
```

```
SELECT * FROM profs, studis  
WHERE profs.course = studis.course;
```

– Sequence 1:

$T_1 = \text{cartesian-product}(\text{profs}, \text{studis})$

$T_2 = \text{select}(T_1, r \rightarrow r[2] = r[4])$

– Sequence 2:

$T_1 = \text{join}(\text{profs}, \text{studis}, 2, 2)$

■ Hash join

- If we know nothing about the table, a simple and reasonably efficient implementation is via a hash join:
 1. Let $x_1, \dots, x_n$ be the rows of the first table X in any order
 2. Let $K = \{x_1[j_1], \dots, x_n[j_1]\}$ be the set of distinct values in the join column j_1 of X
 3. Compute a hash map H , which maps each $k \in K$ to the set of indices of rows of X containing k in its join column
 4. Let $y_1, \dots, y_n$ be the rows of the second table Y in any order
 5. For each row y_j , if $y = y_j[j_2] \in K$, then for each $i \in H[y]$, add row (X_i, Y_j) to the result

The next slide provides an example computation

Join processing 2/8

■ Hash join, example computation

– Operation:

$T_1 = \text{profs}, T_2 = \text{studis}$

$\text{join}(T_1, T_2, 2, 2)$

$K = \{v720, v2040, v1304\}$

Hash map H

$v720 : \{1\}$

$v2040 : \{2, 4\}$

$v1304 : \{3\}$

Iterate over Y

$y_1: v1304 \rightarrow \text{matches } x_3$

$y_2: v2040 \rightarrow \text{matches } x_2, x_4$

$y_3: v2040 \rightarrow \text{matches } x_2, x_4$

...

Y

join column

prof	course	student	course
		7383436	V1304
		3487464	V2040
x_1 Brox	V720	y_1 3344637	V2040
x_2 Hutter	V2040	y_2 3347343	V2060
x_3 Bast	V1304	y_3 5634555	V2040
x_4 Burgard	V2040	y_4 7537345	S1303

prof	course	student	course
Bast	V1304	7383436	V1304
Hutter	V2040	3487464	V2040
Burgard	V2040	3487464	V2040
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.

■ Hash join, running time

*in practice, $C_L < C_I$
so $n_1 < n_2$ is good*

- The hash join takes time $C_I \cdot n_1 + C_L \cdot n_2 + C_C \cdot n \cdot k$, where
 - n_1 and n_2 are the number of rows in X and Y
 - n and k are the number of rows and columns in the result
 - C_I is the average cost for an insertion into hash map H
 - C_L is the average cost for a lookup in hash map H
 - C_C is the average cost for copying a value to the result
- Note:
 - In the worst case, $n = n_1 \cdot n_2$ and this is asymptotically just as expensive as the Cartesian product
 - But typically, $n \ll n_1 \cdot n_2$

■ Merge join

- Assume that we have the values in the join columns of X and Y in sorted order, namely as

$(x_1, i_1), (x_2, i_2), \dots, (x_n, i_n)$ $\mathcal{L}_X$

$(y_1, j_1), (y_2, j_2), \dots, (y_n, j_n)$ $\mathcal{L}_Y$

where the second component is the index of the row in X or Y , where the value from the first component occurs

- Then we can compute the join using our **Zipper algorithm** from Lecture 1 for the intersection of sorted lists

Important detail: whenever a pair from X matches a pair of Y , output **all** pairs from Y with the same value ... see next slide

Join processing 5/8

■ Merge join, example computation

$L_X = (V1304, 3), (V2040, 2),$
 $(V2040, 4), (V720, 1)$

$L_Y = (S1303, 6), (V1304, 1),$
 $(V2040, 2), (V2040, 3),$
 $(V2040, 5), (V2060, 4)$

ZIPPER MERGE $\swarrow$ from L_X $\searrow$ from L_Y

$R = (V1304, 3, 1)$
 $(V2040, 2, 2)$
 $(V2040, 4, 2)$
 $(V2040, 2, 3)$
 $(V2040, 4, 3)$
 $(V2040, 2, 5)$
 $(V2040, 4, 5)$

Join column

prof	course
Brox	V720
Hutter	V2040
Bast	V1304
Burgard	V2040

Join col

student	course
7383436	V1304
3487464	V2040
3344637	V2040
3347343	V2060
5634555	V2040
7537345	S1303

Prof	Course	Student	course
Bast	V1304	7383436	V1304
Hutter	V2040	3487464	V2040
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.

- Merge join, running time [given the already sorted sequences]
 - The merge join takes time $C_M \cdot (n_1 + n_2 + n) + C_C \cdot n \cdot k$, where
 - n_1 and n_2 are the number of rows in X and Y
 - n and k are the number of rows and columns in the result
 - C_M is the average cost for comparing or writing one element with the Zipper algorithm
 - C_C is the average cost for copying a value to the result
 - Note:
 - This is asymptotically the same as for the hash join, but typically $C_M \ll C_I$ and $C_M < C_L$, so that a merge join is faster especially when both n_1 and n_2 are large

■ Hash join or merge join, cost estimate

- For ES5, consider the following simplistic estimates, where n_1 and n_2 are the size estimates of tables X and Y , respectively
 - $\text{cost} \sim n_1 + n_2 + \min(n_1, n_2) \cdot k$
 - $\text{rows} \sim \min(n_1, n_2)$
 - $\text{cols} = k$
- Notes:
 - For the "cost" estimate, we ignore the constants C_I , C_L , C_M , C_C from slides 6 and 9 because they are hard to quantify
 - For the "rows" estimate, we simplistically assume that each row from the table with fewer rows has one match in the table with more rows

■ Particularities of SQLite3

- SQLite3 does not have a hash join

On <https://www.sqlite.org/optoverview.html> it dubiously says:
"An automatic index is about the same thing as a hash join"

- SQLite3 does not perform a merge join either

Instead, it always iterates over the values in one join column and searches them in the other (using an index if it exists or computing a temporary index if it does not exist)

More about **indexing** and SQLite3's peculiarities in **Lecture 6**

- A typical RDBMDs supports both merge and hash joins and it's part of the query planning to figure out which one is better

More join operations 1/5

■ Why more join operations?

- So far, our join operation only included rows in the result, when the respective join column value occurs in **both** tables

This is called an **inner join** (and it's the default)

- We could also include those rows from T_1 in the result, where the value in the join column does not occur in T_2

This is called a **left outer join** ... where "left" refers to T_1

Similarly, we can define a **right outer join** ... referring to T_2

- We could also include rows from both tables in the results, even if their join column values does not match the other table

This is simply called an **outer join** (= both left and right)

- We don't define these operations here, just two examples

More join operations 2/5

■ Left outer join, example SQL query and result

```
CREATE TABLE profs(prof TEXT, course TEXT);
```

```
CREATE TABLE studis(student TEXT, course TEXT);
```

```
SELECT * FROM profs LEFT JOIN studis ON profs.course = studis.course;
```

X

prof	course
Brox	V720
Hutter	V2040
Bast	V1304
Burgard	V2040

Y

student	course
7383436	V1304
3487464	V2040
3344637	V2040
3347343	V2060
5634555	V2040
7537345	S1303
7456444	S1303

X

prof	course	student	course
Brox	V720	NULL	NULL
Hutter	V2040	3487464	V2040
Burgard	V2040	3487464	V2040
Hutter	V2040	3344637	V2040
Burgard	V2040	3344637	V2040
Hutter	V2040	5634555	V2040
Burgard	V2040	5634555	V2040
Bast	V1304	7383436	V1304

More join operations 3/5

■ Right outer join, example SQL query and result

```
CREATE TABLE profs(prof TEXT, course TEXT);  
CREATE TABLE studis(student TEXT, course TEXT);
```

```
SELECT * FROM profs RIGHT JOIN studis ON profs.course = studis.course;
```

✗

prof	course
Brox	V720
Hutter	V2040
Bast	V1304
Burgard	V2040

✓

student	course
7383436	V1304
3487464	V2040
3344637	V2040
3347343	V2060
5634555	V2040
7537345	S1303

□ = does not
matter in ✗

✗

prof	course	student	course
Hutter	V2040	3487464	V2040
Burgard	V2040	3487464	V2040
Hutter	V2040	3344637	V2040
Burgard	V2040	3344637	V2040
Hutter	V2040	5634555	V2040
Burgard	V2040	5634555	V2040
Bast	V1304	7383436	V1304
NULL	NULL	3347343	V2060
NULL	NULL	7537345	S1303

More join operations 4/5

■ Natural join

- A **natural join** between two tables is an equi-join between two tables that considers all columns as join columns that have the same name in both tables
- For example, the following three SQL queries are all equivalent

```
CREATE TABLE profs(prof TEXT, course TEXT);
```

```
CREATE TABLE studis(student TEXT, course TEXT);
```

```
SELECT * FROM profs, studis WHERE profs.course = studis.course;
```

```
SELECT * FROM profs JOIN studis ON profs.course = studis.course;
```

```
SELECT * FROM profs NATURAL JOIN studis;
```

The second is the most modern formulation, the first is more traditional, the third makes the join columns (too) implicit

More join operations 5/5

■ Non-equi join

- The condition on the join columns does not have to be equality
- For example, the following query computes all restaurants that are within 500 meters of a bus stop

```
CREATE TABLE bus_stops(name TEXT, loc POINT);  
CREATE TABLE restaurants(name TEXT, loc POINT);  
  
SELECT * FROM bus_stops JOIN restaurants  
ON ST_Distance(bus_stop.loc, restaurants.loc) < 500;
```

This is a so-called **spatial join**

- The exact formulation of the query above depends on the database; for SQLite3, this requires the **Spatialite** extension

■ Motivation

- Many natural database queries require **grouping** the rows of a table by some common value and then computing something interesting for each group

For example, all movies by the same director and their average IMDb score ... a variant of this query is on ES5

- The next slide provides the formal definition of the corresponding operation, after that we see several examples

The definition looks complicated, but is actually not hard to understand, the examples will clarify this

■ The group-by operation

* the sets $\{g_1, \dots, g_m\}$ and $\{a_1, \dots, a_n\}$ must be disjoint and m must be ≥ 1

– Signature: **group-by**($T, \text{cols}, \text{agg}$), where*

- $T = (C, D, R)$ is an arbitrary table with k columns
- cols is a sequence $(g_1, \dots, g_m)$, where each $g_j \in \{1, \dots, k\}$
- agg is a sequence $((a_1, f_1), \dots, (a_n, f_n))$ of pairs, where each $a_j \in \{1, \dots, k\}$ and each $f_j : \mathcal{P}(D_{a_j}) \rightarrow D'_{a_j}$ is a function

– Result: a table $T' = (C', D', R')$, where

- $C' = (c_{g_1}, \dots, c_{g_m}, c_{a_1}, \dots, c_{a_n})$
- $D' = (D_{g_1}, \dots, D_{g_m}, D'_{a_1}, \dots, D'_{a_n})$
- $R' = \{ (r_{g_1}, \dots, r_{g_m}, r'_{a_1}, \dots, r'_{a_n}) : r \in \text{project}(T, (g_1, \dots, g_m), T) \text{ and } r'_{a_j} = f_j(\{ r''_{a_j} : (r''_1, \dots, r''_k) \in R \wedge r''_{g_j} = r_{g_j} \text{ for all } j \}) \}$

$\mathcal{P}(X) := \text{power set of } X$
 $= \text{set of all multisubsets}$

Grouping 3/7

■ GROUP BY, example

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

```
SELECT year, MAX(score) AS max_score  
FROM movies GROUP BY year;
```

Operation: group-by(movies, (2), (3, max))

title	year	score
Oppenheimer	2023	8.5
Kashmir files	2022	8.6
Western Front	2022	7.8
The Whale	2022	7.7
Dune	2021	8.0
King Richard	2021	7.5

year	max_score
2023	8.5
2021	8.0
2022	8.6

■ GROUP BY with HAVING

- Assume we want to only show those groups, where $\text{max_score} > 8.0$
- This cannot easily be expressed using our syntax so far

Because max_score is not a column of the original table, but created by the GROUP BY

- To formulate conditions on the results of the aggregate functions, we can use the **HAVING** clause

See the example on the next slide and figure out yourself how one would have to extend the formal definition of group-by

Grouping 5/7

■ GROUP BY with HAVING, example

```
CREATE TABLE movies(name TEXT, year INT, score REAL);
```

```
SELECT year, MAX(score) AS max_score
```

```
FROM movies GROUP BY year HAVING max_score > 8.0;
```

Operation: group-by(movies, (2), (3, max), $r \rightarrow r[2] > 8.0$)

name	year	score
Oppenheimer	2023	8.5
Kashmir files	2022	8.6
Western Front	2022	7.8
The Whale	2022	7.7
Dune	2021	8.0
King Richard	2021	7.5

year	max_score
2023	8.5
2022	8.6

2021 : max = 8.0 → ✗
2022 : max = 8.6 → ✓
2023 : max = 8.5 → ✓

■ GROUP BY, implementation and cost estimates

- The straightforward implementation is via a hash map

The implementation of GROUP BY is part of ES5, Exercise 1

- Here are estimates for this operation when the input table has n rows and the output table has k columns

$\text{cost} \sim n \cdot k$, $\text{rows} \sim n/2$, $\text{cols} = k$

- Simplistic assumptions behind these estimates
 - The average size of each group is 2 [random and unrealistic]
 - Insertion into a hash map takes constant time [realistic]
 - Evaluating the aggregation function f_j is cheap and takes time linear in the number of arguments [often realistic]

■ GROUP BY, advanced aggregation

- Here is an example for a typical query that requires a GROUP BY, but cannot be expressed using only GROUP BY

For each year, the movie with the highest score and the corresponding title

- The following query gives us the highest score per year

```
CREATE TABLE movies(name TEXT, year INT, score REAL);  
  
SELECT year, MAX(score) AS max_score  
FROM movies GROUP BY year;
```

- But how do we get the corresponding movie?

GROUP BY only allows per-column aggregation, but here we need the title with a property (highest score) from another column

- There are many, many, many more SQL operations
 - The SQL standard is written down on hundreds of page
See [ISO/IEC 9075:2023 \(ANSI X3.135\)](#) ... it's expensive
 - In this course, we will just provide the most basic important operations, see <https://sqlite.org/lang.html> for more
 - On the following slides, we will briefly explain the following three by example (no more formal operations)
 - ORDER BY
 - LIMIT and OFFSET
 - Subqueries

All three are needed very frequently ... and also for ES5

■ ORDER BY

- According to our definition from Lecture 4, tables are multisets of rows, that is, the rows do not have an inherent order
- However, for the final result, we often want an order
- This can be accomplished via **ORDER BY [expression] [ASC|DESC]** at the end of the SELECT query, for example:

```
SELECT * FROM movies ORDER BY score DESC;
```

- If there also is a GROUP BY, the ORDER BY comes afterwards:

```
SELECT year, MAX(score) AS max_score FROM movies  
GROUP BY year ORDER BY year ASC;
```

■ LIMIT and OFFSET

- Once we have an order, it is natural to ask only for a segment of the sorted list of row
- For example, the movies with the 10 highest scores

```
SELECT title, score FROM movies  
ORDER BY score DESC LIMIT 10
```

- If we want a segment in the middle, we need OFFSET, for example, for the 5 movies after the top 10:

```
SELECT title, score FROM movies  
ORDER BY score DESC LIMIT 5 OFFSET 10
```

- Note that (naturally), LIMIT and OFFSET come after ORDER BY (which, again naturally, comes after GROUP BY)

■ Subqueries

- All operations take one or more tables as an argument
- A SELECT query has a table as a result
- So it's natural that in SQL, wherever you can write a table, you can also write a whole SELECT query; you just have to put it in parentheses, for example:

```
SELECT title FROM movies
JOIN directors ON movies.movie_id = directors.movie_id
JOIN (
    SELECT DISTINCT person_id FROM awards
    WHERE award_name LIKE 'Academy Awards%'
) AS awardees ON directors.person_id = awardees.person_id;
```

Explanation on next slide ... you also need a subquery for ES5

■ Subqueries, explanation of example query

- The subquery gives us a table with a single column, namely the person IDs of all Oscar winners; so we effectively have

```
CREATE TABLE movies(movie_id INT, title TEXT, ...);  
CREATE TABLE directors(movie_id INT, person_id INT);  
CREATE TABLE awardees(person_id ID);
```

- In terms of these tables, our query is then equivalent to

```
SELECT title FROM movies  
NATURAL JOIN directors NATURAL JOIN awardees;
```

which gives us the title of all movies directed by the people in table awardees (that is, Oscar winners)

To understand a SQL query with subqueries, understand the table created by each subquery

References

■ Relevant Wikipedia articles

https://en.wikipedia.org/wiki/Relational_algebra [other joins]

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

https://en.wikipedia.org/wiki/Sort-merge_join

<https://en.wikipedia.org/wiki/SQL> [SQL standard]

■ SQLite3

Explanation why there are no merge joins in SQLite3:

https://www.sqlite.org/optoverview.html#hash_joins

Overview over SELECT query syntax of SQLite3:

https://sqlite.org/lang_select.html