

universität freiburg

Databases and Information Systems WS 25/26

Lecture 6: Indexing, Query Optimization, ACID

November 25, 2025

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

Overview of this lecture

■ Organizational

- Your experiences with ES5
- Joins and more SQL

■ Contents

- Index B+ tree
 - Query optimization
 - ACID
- For faster joins
- Computing the optimal query plan
- Dynamic updates of the data

ES6: Run the query optimization by hand on a given example
+ integrate query optimization into your own DB system

Experiences with ES5 1/2

■ Excerpts from your feedback [somewhat representative]

"Again, the lecture and the sheet were really nice"

"A great exercise that helped to deeply understand the concepts"

"Very fun sheet, as always ... but the manual query planning makes my head hurt"

"I enjoyed the lecture but found the exercise sheet harder than usual, mainly to understand grouping with multiple columns"

"I really struggled with the execute method for GroupBy"

"The sheet was very hard and I did not have enough time"

"I liked that instead of switching between slides and editing mode to fix a wrong font style, the assistant just takes note of it"

Experiences with ES5 2/2

- What does it say about humanity that the most used RDBMS in the world has neither a hash nor merge join
 - "That they avoid Algorithms and Data Structures at all cost"
 - "People just start out with something that works for now ... then it becomes too much of a hassle to replace it"
 - "As long as things work, humans don't feel the need to change it"
 - "People care about practicability"
 - "The obvious way is not the best in all situations"
 - "1. SQLite is used more for [small data and] individual purposes
2. Individuality leads to a lot of problems we are facing right now such as climate change and poverty
3. Once the most used database engine in the world has a hash join and a merge join, we know that humanity is saved!"

■ Motivation

- To accelerate the computation of a join, we can precompute a so-called **index** on the join columns of the respective tables
- For example, here are the SQL commands to compute an index on the `movie_id` column of the tables `movies` and `persons`

```
CREATE INDEX movies_index ON movies(movie_id);  
CREATE INDEX persons_index ON persons(movie_id);
```

- Let's do this with SQLite3 and measure the effect on a query like `SELECT * FROM movies NATURAL JOIN roles;`

Observation 1: faster with indexes than without

Observation 2: not faster with both indexes instead of just one

■ Definition of an **index B+ tree**

- Input: a table and a column in that table with sortable values

This is easy to generalize to multiple columns

- We define the index B+ tree for this input as a **search tree** with a given degree **d** with the following properties
 - Each leaf contains up to **d** pairs (value, row index)
 - There is a pointer from one leaf to the next, and the leaf pairs are in sorted order when scanned from left to right
 - Each of the inner nodes has **$d' \leq d$** ordered children and stores the smallest value for each child 2, ..., d'
- The next three slides will provide two examples for $d = 3$

Index B+ Tree 3/8

- Table **movies** with 9 rows ... build index for **rating**

	title	year	rating
1	Forrest Gump	1994	8.8
2	The Matrix	1999	8.7
3	The Godfather	1972	9.2
4	Mulholland Drive	2001	7.9
5	The Shawshank Redemption	1994	9.3
6	Back to the Future	1985	8.5
7	12 Angry Men	1954	9.0
8	Dr. Strangelove	1964	8.3
9	The Dark Knight	2008	9.1

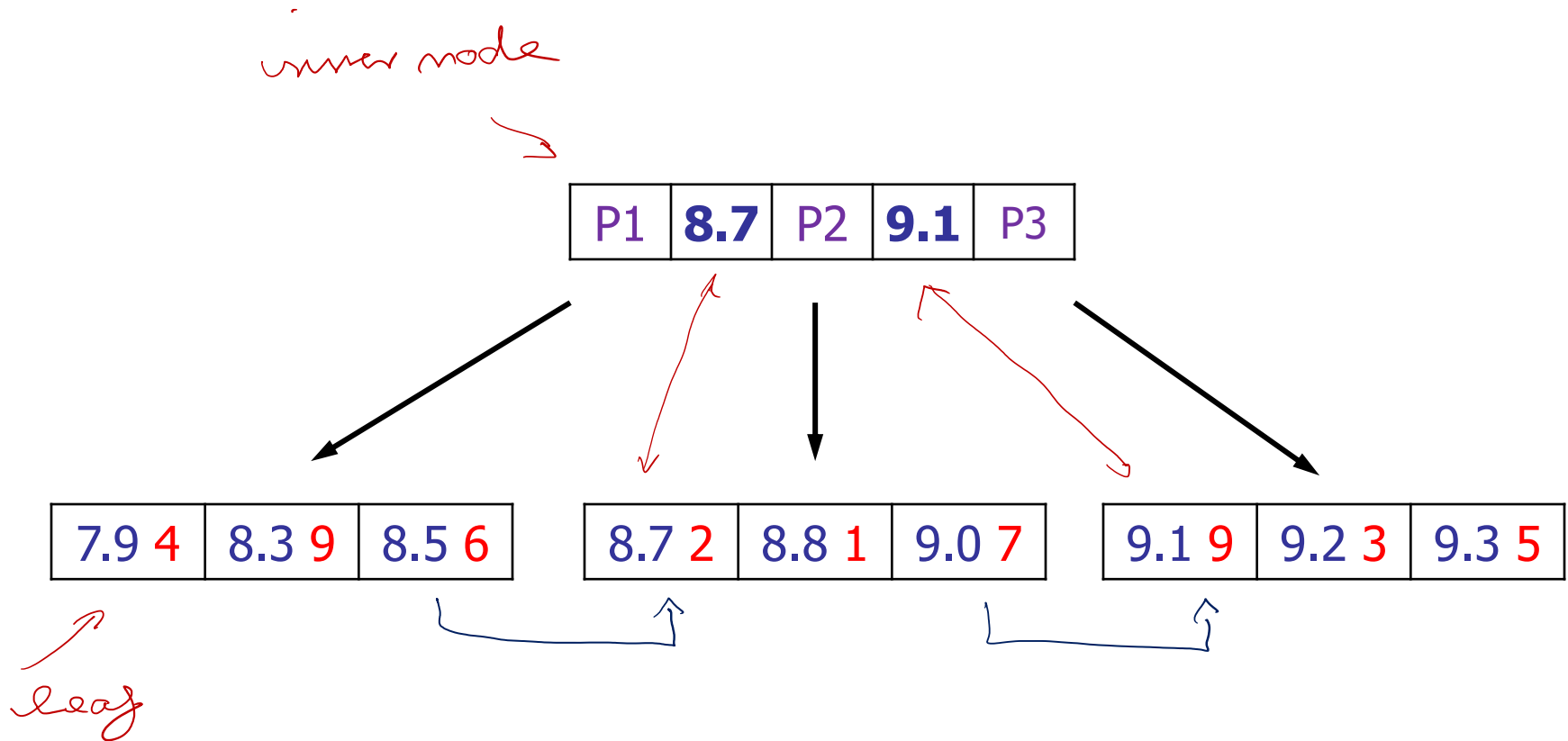
Pairs of (value, row index) in sorted order (by value).

(7.9 4), (8.3 8), (8.5 6), (8.7 2), (8.8 1), (9.0 7), (9.1 9), (9.2 3), (9.3 5)

Index B+ Tree 4/8

■ Index B+ Tree for column **rating** of table **movies**

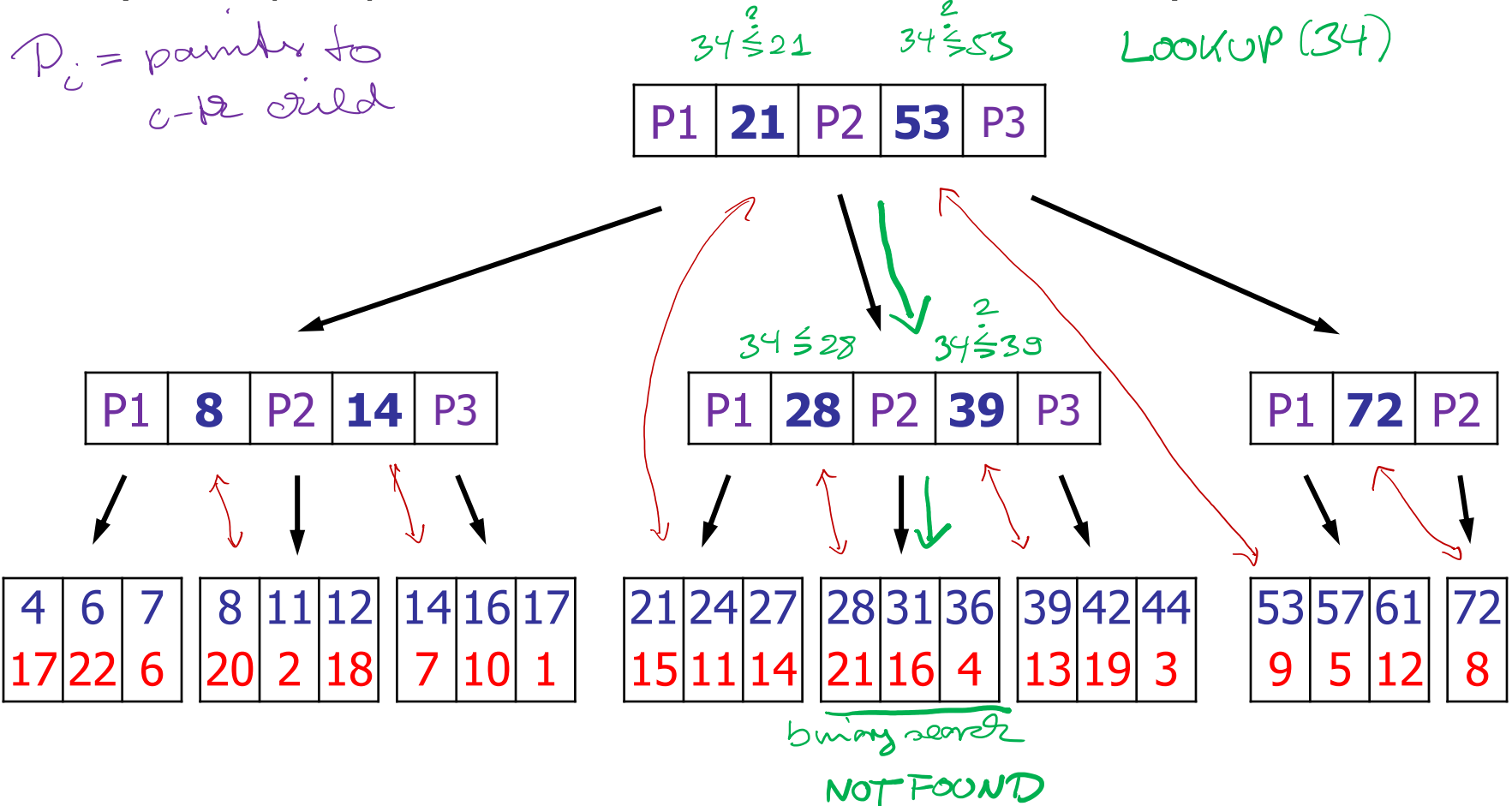
$d=3$



Index B+ Tree 5/8

- Index B+ Tree with 8 leaves and 22 values
(for simplicity, we assume that all values are distinct)

$d = 3$



Index B+ Tree 6/8

■ Lookup in a B+ tree ... with degree d and n values stored

- Lookup of a value takes $O(\log n)$ time

*binary search in
d elements*

Time proportional to $\log_2 d$ time in each inner node or leaf

The depth of the tree is bounded by $\lceil \log_d n \rceil$

$$\log_d n \cdot \log_2 d = \frac{\log n}{\log d} \cdot \frac{\log d}{\log 2} = \log_2 n$$

- When one IO-operation can read a complete inner node or leaf at a time, then lookup of a value takes $\log_d n$ IO-operations

See [Algorithms and Data Structures, Vorlesung 7](#)

- For joining, SQLite3 uses B+ trees as follows:
 - iterate over one join column
 - lookup the values in the B+ tree of the other join column

Index B+ Tree 7/8

■ Iterating over the leaves of a B+ tree

- An index B+ tree trivially provides all pairs (value, row index) in sorted order
- If we have an index B+ tree for both join columns, this is **exactly** the input we need for an efficient merge join

See [Lecture 5, Slide 13](#)

- SQLite3 **never** uses a merge join, even when it has a B+ tree for both join columns

That is an interesting design decision

■ Variations and dynamic updates

- A **clustered B+ tree** stores the full table rows in the leaves (instead of just the row index, like an index B+ tree does)

That way, we can produce the full result of the join directly from the B+ tree, but costs more space and is less IO-friendly

- A **B tree** (as opposed to B+ tree) stores full table rows or pairs (value, row index) also in the inner nodes, not only the leaves

If we want one IO-operation per inner node, that reduces the number of children and hence increases the depth of the tree

Also, some data is now stored twice (leaf and inner node)

- In practice, updates (of tables and indexes) are important

We briefly come back to that at the end of this lecture

Computing the optimal query plan 1/9

- Recap from the last two lectures (and exercise sheets)
 - In Lectures 4 and 5, we have learned what a query plan is
A sequence of operations that, starting from the input tables, compute the final result table
 - We have also learned how to estimate the cost of a query plan
For each operation, estimate its cost and the size of its result from the size estimate for the input tables of the operation
 - So far, we have written a few **individual** query plans in code ... and compared their cost (in particular, in ES 4 and 5)
 - In the following we want to find **the best query plan from all possible query plans** automatically

Computing the optimal query plan 2/9

■ Assumptions and terminology

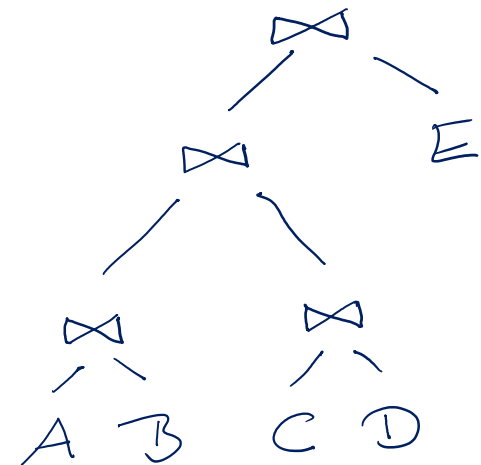
- For simplicity, we only consider binary join operations

As we will see, that problem alone is already hard enough

- We call our tables by capital letters, e.g., **A B C D E**
- We write a query plan as a **parenthesized expression**, e.g.

((AB)(CD))E → this means that we first join A with B and C with D, then we join the two results, and then we join that result with E

On the right, we see the equivalent tree (parenthesized expressions correspond to trees; see Lecture 4 Slide 18)

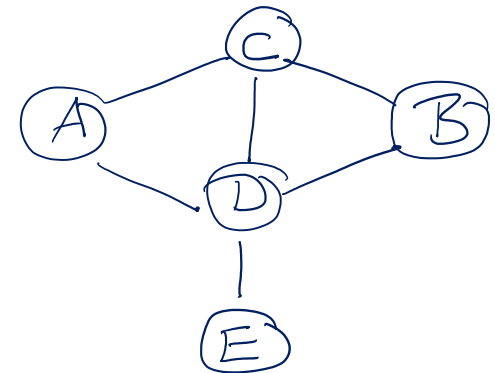


Computing the optimal query plan 3/9

■ Join graph

- We need to know which of the tables can and should be joined by the given SQL query
- This can be represented using a **join graph**, where each node is a table and there is an edge between two tables if the two tables can and should be joined ... for example:

A: C, D SELECT title, name, role, award_name
B: C, D FROM movies as **A**
C: A, B, D NATURAL JOIN persons as **B**
D: A, B, C, E NATURAL JOIN roles as **C**
E: D NATURAL JOIN awards as **D**
 NATURAL JOIN award_names as **E**



Computing the optimal query plan 4/9

■ Cost function

- We take a variant of the cost function from Lecture 5 (the cost is now the **total cost**, that is, the cost of the join **plus** the cost for the two input tables)

$$\text{size}(A \bowtie B) = \min\{ \text{size}(A), \text{size}(B) \}$$

$$\text{cost}(A \bowtie B) = \text{cost}(A) + \text{cost}(B) + \text{size}(A \bowtie B)$$

- For example:

Table A: cost = 0, size = 100

Table B: cost = 0, size = 200

Table C: cost = 0, size = 150

$$\text{size}((AB)C) = \min\{100, 150\} = 100$$

$$\text{cost}((AB)C) = 100 + 0 + 100 = 200$$

$$\begin{aligned} \text{size}(AB) &= \min\{100, 200\} = 100 \\ \text{cost}(AB) &= 0 + 0 + 100 = 100 \end{aligned}$$

$$\begin{aligned} \text{cost}(A(BC)) &= ? \\ \text{cost}((AC)B) &= ? \end{aligned}$$

Computing the optimal query plan 5/9

■ Formula for the optimal cost

e.g. $S = \{A, B, C\}$
 $S_1 = \{A, B\}, S_2 = \{C\}$
 $S_1 = \{B\}, S_2 = \{A, C\} \dots$

- We can relatively easily define the cost for the optimal query plan for a set S of input tables as follows

$$\text{cost}^*(S) = \min \{ \text{cost}^*(S_1) + \text{cost}^*(S_2) + \text{cost}^*(S_1 \bowtie S_2) \}$$

where the minimum is over all sets S_1 and S_2 with $S_1 \cap S_2 = \emptyset$ and $S_1 \cup S_2 = S$ and $S_1 \neq \emptyset$ and $S_2 \neq \emptyset$

- Note that the recursion ends because S_1 and S_2 are strict subsets of S and for sets of size 1 (a single table) cost = 0
- A straightforward program would compute this cost using a recursive function following the formula above

How well do you think that will work?

■ Interlude: Fibonacci numbers

- The Fibonacci numbers can be defined recursively as follows

$$F(0) = 0, F(1) = 1, F(n) = F(n - 1) + F(n - 2)$$

- Let us compute the first few of them by hand

0, 1, 1, 2, 3, 5, 8, ...

- Let us write an recursive function for this

Observation: **F(30)** is still computed reasonably fast, but **F(50)** already seems to take forever

- Reason: our code has running time **$O(2^n)$**
- But it's trivial to implement the computation in linear time

Computing the optimal query plan 7/9

■ Computing $\text{cost}^*(S)$ iteratively in rounds

- Round 1: we start with "singleton" query plans, one for each of the n input tables
- Round 2: we compute the cost of all possible joins of two of the input tables, based on the costs from round 1

...

- Round k : we compute the cost of all possible joins of k of the input tables, based on the results from the previous rounds

For $k_1 = 1, \dots, k//2$ and $k_2 = k - k_1$, we compute the cost of all joinable combinations of a joinable subset of size k_1 and a joinable subsets of size k_2 (for each of which we already know the optimal cost from a previous round) and pick the best one

Computing the optimal query plan 8/9

■ Example computation

- Let's consider three tables **A**, **B**, and **C** with the following sizes
 $\text{size}(\text{A}) = 100, \text{size}(\text{B}) = 200, \text{size}(\text{C}) = 300$
- Let's assume that each pair of tables can be joined with each other (that is, the query graph is a so-called **clique**)
- Then we can compute the optimal join in three rounds:

Computing the optimal query plan 9/9

■ Implementation details

- In round k , do not iterate over all subsets of size k , but only consider the combinations of query plans from previous round that result in a subset of size k

This is already part of the algorithm on the previous slide

- To determine whether two query plans are joinable, we check
 - whether the two sets of tables S_1 and S_2 are disjoint
 - whether there is an edge between a table from S_1 and a table from S_2 in the query graph

In our code, we do this via a nested loop, which has running time $O(S_1 \cdot S_2)$... in practice one would use **bit masks**

■ Dynamically changing tables

- So far, we have read our table once in the beginning and then asked all kinds of queries on it

This is indeed a frequent use case: for example, there is a new dump for Wikidata available only once per month, and we can simply create a new database for each new dump

- However, there are also many applications, where the values in the table change all the time, typical examples are:

bank data (who has how much money on their accounts)

product data (which products are there with which price)

HISinOne (who has which grade for which exam)

■ Changing the **rows** in a table

- Here are three (self-explanatory) example SQL queries for inserting, changing, or deleting rows in a given table

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

```
INSERT INTO movies VALUES('Forrest Gump', 1994, 8.8);
```

```
UPDATE movies SET rating = 8.9 WHERE title = 'Inception';
```

```
DELETE FROM movies WHERE year < 2000;
```

title	year	score
The Shawshank Redemption	1994	9.3
The Dark Knight	2008	9.0
Inception	2010	8.8
Fight Club	1999	8.8

- Changing the **columns** in a table (don't do it)
 - Here are four (self-explanatory) example SQL queries for adding, deleting, renaming, or changing the datatype of a column

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

```
ALTER TABLE movies ADD COLUMN genre;
```

```
ALTER TABLE movies DROP COLUMN genre;
```

```
ALTER TABLE movies RENAME COLUMN score TO rating;
```

```
ALTER TABLE movies ALTER COLUMN rating TYPE INT;
```

One should be very careful with these because users of a database often rely on a fixed schema of the tables

■ Transactions

- When we change the contents of the tables in our database, we have to be very careful with the possibility of errors

If, in a read-only database, a query fails, all that happens is that we don't get a result and we have to ask a again

If, in a money transfer from account X to account Y, the amount was subtracted from X, but not added to Y, that's a problem

- A sequence of database operations that should be considered as one unit is called a **transaction**
- On the next slide, we formulate the four **ACID** properties that transactions should have

We don't have time to go deeper into this in this course

■ The ACID properties

- A transaction is **ACID** if the following four properties hold:

Atomicity: Each transaction (sequence of operations) either fails 100% or succeeds 100%, there is nothing in-between

Consistency: The database always fulfills a set of pre-defined properties, if a transaction invalidates them, it must fail

Isolation: If a sequence of transactions is executed concurrently, the result is the same as if they were executed sequentially

Durability: The changes due to successful transactions never get lost, even if the database crashes at an arbitrary point

- See <https://en.wikipedia.org/wiki/ACID> for an example for each

This is not klausur  **but we did want to mention it**

References

- Relevant Wikipedia articles

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

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

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

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

- SQLite3

Explanation why there are no merge joins in SQLite3:

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