Databases Under The Hood

An Introduction For The Curious User

Annett Ungethüm, 16.12.2021
Introduction CDCS
CDCS Structure

**UHH Project Coordinator**
Prof. Dr. Matthias Rarey
UHH Computer Science
Spokesperson CDCS

**DESY Project Coordinator**
Prof. Dr. Nina Rohringer
DESY / UHH Physics

**TUHH Project Coordinator**
Prof. Dr. Sabine Le Borne
TUHH Mathematics

---

**Accelerator Physics**
Head: Prof. Fey (TUHH) / Prof. Schlarb (DESY)

**Systems Biology**
Head: Prof. Grünewald (UHH)/ Prof. Baumbach (UHH)

**Astro & Particle Physics**
Head: Prof. Schleper (DESY)/ Dr. Gaede (UHH)

**Computational Core Unit**
Head: Prof. Rarey (UHH)/ Prof. Knopp (TUHH)

**Photon Science**
Head: Dr. Barty (DESY)/ Dr. White (DESY)

---

18.11.2021

Introduction CDCS
As a DASHH student you can get a transponder to the CDCS hot desk office space (room 1064)
Ask our secretary Miriam Döring: miriam.doering@uni-hamburg.de
Data Science Thursdays: Database Timeline

Topic of the month: Databases

You might want to send us your questions in advance to get more sophisticated answers

16 Dec 2021
Why and how should I use a database and why it is different from an excel sheet

23 Dec 2021
Getting help with your first queries (upon request)

06 Jan 2021
Relational DBs, document stores, key-value stores: There's a system for every use-case

13 Jan 2021
Get your research data into a database!
Data science works with data, it’s even in the name

Databases are made for managing and analyzing data
## Why use a DB?

<table>
<thead>
<tr>
<th></th>
<th>Database</th>
<th>CSV, Excel sheets</th>
<th>Other file formats: Hdf5, Binary,…</th>
<th>Libraries: dplyr, pandas,…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast loading and parsing</td>
<td>✓</td>
<td>×</td>
<td>Possible but depends on your tools and knowledge</td>
<td>✓</td>
</tr>
<tr>
<td>Automatic parallellization</td>
<td>✓ <em>most DB systems</em></td>
<td>✓ <em>most DB systems</em></td>
<td>Depends on your tools</td>
<td>×</td>
</tr>
<tr>
<td>Cares for data validity, e.g. consistent transactions</td>
<td>✓</td>
<td>×</td>
<td>Only if you combine it with a database, e.g. hdf5 + Hadoop</td>
<td>×</td>
</tr>
<tr>
<td>Optimizes your queries</td>
<td>✓ <em>most DB systems</em></td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Optimized join of data</td>
<td>✓ <em>most DB systems</em></td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Offers a turing complete query language</td>
<td>✓</td>
<td>✓</td>
<td>Depending on your tools</td>
<td>✓</td>
</tr>
</tbody>
</table>

**By „most” I mean all systems but SQLite. We will get to that point later.**

**You can implement all of this yourself…However, generations of PhD students in systems architecture will tell you to run if your supervisor ever asks you to do this.**

**These are basically query execution engines, but they are slow engines.**
It’s simple!

Let’s assume we have to tables: Table_A and Table_B.

<table>
<thead>
<tr>
<th>Column_1</th>
<th>Column_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Column_a</th>
<th>Column_b</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

Let’s further assume we want all entries where Column_2 equals Column_a.

<table>
<thead>
<tr>
<th>Column_1</th>
<th>Column_2</th>
<th>Column_a</th>
<th>Column_b</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

**R**

```r
inner_join(Table_A, Table_B, by = c("Column_2" = "Column_a"))
```

**Python**

```python
pd.merge (Table_A, Table_B, left_on=‘Column_2’, right_on=‘Column_a’, how=‘inner’)
```

**SQL**

```sql
SELECT * FROM Table_A, Table_B WHERE Table_A.Column_2=Table_B.Column_a;
```

→ It won’t win a prize for literature, but it’s close to a spoken language.
Join tables with Excel

**Quick way to join tables by matching columns**

*Requires an add-on*

If you’re not comfortable with Excel formulas yet, or do you have time to figure out the arcane quirks Power Query, our Merge Tables Wizard could be your time saver. Below I’ll show three most popular add-on cases.

**Example 1.** Combine two tables by multiple columns

If you find the array formula for columns match hard to remember, rely on our add-in to do the job quickly and perfectly.

For this example, we will be using the already familiar tables and join them based on 2 columns Seller and Product. Please note that the Lookup table has 2 more columns than the main table.

*With the Merge Tables Wizard added to your Excel ribbon, here’s what you need to do:*

1. Select any cell within your main table and click the Merge Two Tables button on the Ablebits Data tab:

2. Specify the column pairs to match, seller and Product in our case, and click Next:

3. Select the lookup table and click Next:

4. Specify the lookup table columns to join and click Next:

5. Optionally, choose the columns to update with the values from the lookup table. Since there is no need to update the Order ID column, let us leave it unselected, and simply click Next:

6. Select the columns to add to your main table and click Next:

7. In this step, you tell the wizard how exactly you want the tables to be merged. All the options have descriptive labels, so won’t go into long explanations. If you are unsure about a certain option, click the question mark next to it, and a small tooltip will show you how the tables are going to be combined.

The default options work just fine in most cases, so we click Finish without changing anything:

*source: ablebits.com*
It’s an all in one solution!

- Optimized data storage and reader
  - Does not kill your file system with thousands of small files
  - Loads only what is necessary, i.e. not always the whole file
  - Indexes your data (more or less automatically)
- Comes with a standardized query language (SQL)
- Optimized operators (e.g. join, merge, and aggregation are operators)
- Query optimizer (the thing that schedules your operators)
- Additional features often included: compression, encoding, user management, out-of-memory execution (in case your files are really big),...
Let's start simple!
The typical broke student problem:
Which meals are cheaper than 5 €?

**MensaMeals**

<table>
<thead>
<tr>
<th>Meal</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pizza</td>
<td>6,50</td>
</tr>
<tr>
<td>Pasta</td>
<td>4,90</td>
</tr>
<tr>
<td>Pie</td>
<td>1,20</td>
</tr>
<tr>
<td>Potato Salad</td>
<td>5,80</td>
</tr>
<tr>
<td>Pannfisch</td>
<td>7,90</td>
</tr>
</tbody>
</table>
Which meals are cheaper than 5€?

To answer this question, we need:

**Data**

<table>
<thead>
<tr>
<th>Meal</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pizza</td>
<td>6.50</td>
</tr>
<tr>
<td>Pasta</td>
<td>4.90</td>
</tr>
<tr>
<td>Pie</td>
<td>1.20</td>
</tr>
<tr>
<td>Potato Salad</td>
<td>5.80</td>
</tr>
<tr>
<td>Pannfisch</td>
<td>7.90</td>
</tr>
</tbody>
</table>

**Query**

Select all meals where the price is lower than 5€.

**A plan**

- Go through all entries sequentially
- Check the price of each entry
- Save an entry if the price is <5

**System**

Something that implements everything.
Data Query Database System

- Go through all entries sequentially
- Check the price of each entry
- Save an entry if the price is <5

Select all meals where the price is lower than 5

<table>
<thead>
<tr>
<th>Meal</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pizza</td>
<td>6.50</td>
</tr>
<tr>
<td>Pasta</td>
<td>4.90</td>
</tr>
<tr>
<td>Pie</td>
<td>1.20</td>
</tr>
<tr>
<td>Potato Salad</td>
<td>5.80</td>
</tr>
<tr>
<td>Pannfisch</td>
<td>7.90</td>
</tr>
</tbody>
</table>

Something that implements everything
Database System
Get data into a virgin database

1. Create an empty table

   ```sql
   CREATE TABLE MensaMeals (Meal TEXT, Prize REAL);
   ```

2. Insert rows

   ```sql
   INSERT INTO MensaMeals VALUES ('Pasta', 4.90);
   ```

Show contents of your table:

   ```sql
   SELECT * FROM MensaMeals;
   ```
Get data into a virgin database

Database Systems offer import functions

- Supported formats differ (csv, parquet, db, ...)
- Syntax differs
- Auto-detection of data types and delimiters may or may not work

**Examples**

- **PostgreSQL**
  ```sql
  COPY MensaMeals FROM 'home/itsme/MensaMeals.csv' DELIMITER ',' CSV HEADER;
  
  CREATE TABLE MensaMeals AS SELECT * FROM 'MensaMeals.csv';
  
  COPY INTO MensaMeals FROM 'home/itsme/MensaMeals.csv';
  ```

- **DuckDB**
  ```sql
  COPY MensaMeals FROM 'home/itsme/MensaMeals.csv' DELIMITER ',' CSV HEADER;
  ```

- **MonetDB**
  ```sql
  COPY MensaMeals FROM 'home/itsme/MensaMeals.csv' DELIMITER ',' CSV HEADER;
  ```
Let’s keep it simple!

...And do some theory while you are still listening.
This is a relation, defined as \( \text{MensaMeals}(\text{Meal}, \text{Price}) \)

*That's why table based DBs are called relational Databases*

<table>
<thead>
<tr>
<th>Meal</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pizza</td>
<td>6.50</td>
</tr>
<tr>
<td>Pasta</td>
<td>4.90</td>
</tr>
<tr>
<td>Pie</td>
<td>1.20</td>
</tr>
<tr>
<td>Potato Salad</td>
<td>5.80</td>
</tr>
<tr>
<td>Pannfisch</td>
<td>7.90</td>
</tr>
</tbody>
</table>

Meal and Price are the attributes of the relation MensaMeals

This is a tuple, which belongs to the relation MensaMeals

Relation and table often used as synonyms but
- A relation can be defined without tuples, i.e. without being a ‘real’ table
- A table is only an illustration of your data
Relations are usually **illustrated** as tables. This tells us nothing about the storage layout (cf. a matrix that can be stored differently → row- or column-major).

2 main layouts to store your table

---

**Row-Store (tuple-wise)**

<table>
<thead>
<tr>
<th>Meal</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pizza</td>
<td>6,50</td>
</tr>
<tr>
<td>Pasta</td>
<td>4,90</td>
</tr>
<tr>
<td>Pie</td>
<td>1,20</td>
</tr>
<tr>
<td>Potato Salad</td>
<td>5,80</td>
</tr>
<tr>
<td>Pannfisch</td>
<td>7,90</td>
</tr>
</tbody>
</table>

*Memory address →*

**Column-Store (attribute-wise)**

<table>
<thead>
<tr>
<th>Pizza</th>
<th>Pasta</th>
<th>Pie</th>
<th>Potato Salad</th>
<th>Pannfisch</th>
</tr>
</thead>
<tbody>
<tr>
<td>6,50</td>
<td>4,90</td>
<td>1,20</td>
<td>5,80</td>
<td>7,90</td>
</tr>
</tbody>
</table>

*Memory address →*

This is what your traditional relational SQL database does.

This is what all (not so traditional) column-oriented databases do.
Why should you care?

Memory access is expensive!

Throughput on an Intel Xeon E3-1275

Note the log scale

Your ideal layout depends on your use-case. Different systems use different layouts, so choose wisely!
NoSQL and column-oriented DBs: Frequent misunderstandings

• NoSQL stands for Not only SQL
• Wide-column DBs (NoSQL) and column-stores (SQL) are not the same, but both often referenced as column-oriented
  ➢ We will use it to reference column-stores
• Usually, column-oriented databases can be queried using SQL and allow the definition of relations
  ➢ Convenience of SQL, and performance and flexibility of column-stores
  ➢ Example: Fast and easy addition/deletion of attributes

```
ALTER TABLE MensaMeals ADD Calories INT NULL;
```

<table>
<thead>
<tr>
<th></th>
<th>Meal</th>
<th>Price</th>
<th>Calories</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Remember what we just learned about random memory access
Database System
Queries

• Queries consist of **operators** and can be formally described with **query languages**, e.g. relational algebra (RA), SQL

• **SQL** is a keyboard-friendly query language while **RA** is used for internal representation

**Examples:**

SQL: SELECT * FROM MensaMeals WHERE Price < 5;

Relational Algebra: \( \sigma_{\text{Price} < 5} \) (MensaMeals)

Result:

<table>
<thead>
<tr>
<th>Meal</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pasta</td>
<td>4.90</td>
</tr>
<tr>
<td>Pie</td>
<td>1.20</td>
</tr>
</tbody>
</table>

**MensaMeals**

<table>
<thead>
<tr>
<th>Meal</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pizza</td>
<td>6.50</td>
</tr>
<tr>
<td>Pasta</td>
<td>4.90</td>
</tr>
<tr>
<td>Pie</td>
<td>1.20</td>
</tr>
<tr>
<td>Potato Salad</td>
<td>5.80</td>
</tr>
<tr>
<td>Pannfisch</td>
<td>7.90</td>
</tr>
</tbody>
</table>
**Operator Examples**

**Project Operator**

Show only the names of all meals where the price is lower than 5€.

\[\pi_{\text{Meal}} (\sigma_{\text{Price} < 5} (\text{MensaMeals}))\]

**Meal** | **Price**
--- | ---
Pasta | 4.90
Pie | 1.20

SELECT Meal FROM MensaMeals WHERE Price < 5;

Result:

<table>
<thead>
<tr>
<th>Meal</th>
</tr>
</thead>
</table>
Pasta |
Pie   |

**Join Operator**

Where can I get the meals which cost less than 5€?

\[\pi_{\text{Mensa}} (\sigma_{\text{Price} < 5} (\text{MensaMeals} \bowtie \text{MensaMeals.Meal} = \text{DailyOffers.Meal} \text{DailyOffers})))\]

**DailyOffers**

<table>
<thead>
<tr>
<th>Mensa</th>
<th>Meal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Campus Mensa</td>
<td>Pizza</td>
</tr>
<tr>
<td>Mensa Cafe</td>
<td>Pie</td>
</tr>
<tr>
<td>Garden Mensa</td>
<td>Pasta</td>
</tr>
<tr>
<td>Old Mensa</td>
<td>Potato Salad</td>
</tr>
</tbody>
</table>

SELECT Mensa FROM MensaMeals JOIN DailyOffers ON MensaMeals.Meal = DailyOffers.Meal WHERE Price < 5;

Result:

<table>
<thead>
<tr>
<th>Mensa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mensa Cafe</td>
</tr>
<tr>
<td>Garden Mensa</td>
</tr>
</tbody>
</table>
Why should you care?

➔ With RA you can do everything, you can do with other algebras, e.g. prove that two queries produce the same results

➔ Restructure a query for better performance or reusability of subqueries

➔ Understand the output of the query optimizer (later today)

A comprehensible list of transformations can be found here: https://www.postgresql.org/message-id/attachment/32513/EquivalenceRules.pdf
Reuse Subqueries with views

Reusability of queries and query results

- Queries and Subqueries (Views) can be stored and referenced → nicer queries
- The result of views can be stored → higher performance for frequently used queries and remote data

CREATE MATERIALIZED VIEW CheapFood AS
SELECT Meal FROM MensaMeals WHERE Price < 5;

REFRESH MATERIALIZED VIEW CheapFood;

- Refresh the view after updates in your base data
- Not supported by all database systems
Database System
Database System

Parser

Reads query and translates it into intermediate language

Query

Data

*strongly simplified
Database System

- Optimizes operator sequence
- Chooses physical operators
- Additional optimizations, e.g. compression, intermediate materialization, data placement, …
- Result: Query Execution Plan

*strongly simplified*
Database System

- **Parser**: Runs the query, manages memory, and returns the result.
- **Optimizer**: Parse query
- **Engine**: Manages memory and returns the result
- **Result**: Manages memory and returns the result
- **Data**: Stores data

*strongly simplified*
Database System

- Query
- Parser
- Optimizer
- Result
- Engine
- Data

*strongly simplified*
Optimizer: Query Execution Plan Optimization

SELECT Mensa FROM MensaMeals JOIN DailyOffers ON MensaMeals.Meal=DailyOffers.Meal WHERE Price < 5;

• Database Systems use a relational algebra for internal representation
• Optimizers try to automatically find the most efficient sequence of operators
  ➔ Conventional approach: Reduce data as early and as cheap as possible
  ➔ Tool: Cardinality/Selectivity estimation
• The chosen sequence of operators is the final Query Execution Plan (QEP)

Further Reading

15.12.2021
Optimizer: Physical Operator Selection

- For each **logical operator** (e.g. join), there can be different **physical operators** (e.g. hash-join, nested-loop-join), i.e. the same operator can be implemented in different ways.

- Joins are a bottleneck in most queries → Join optimization is a much-noticed field of research.

- Choice of physical operator depends on exact use case. Examples from PostgreSQL:
  - Nested-Loop: full join, one very small table, condition is not an equality
  - Hash Join: similarity joins, small expected hash table
  - Merge Join: sorted data, large tables

Results for different search terms in google scholar

Further Reading

More on join order optimization: *Query optimization through the looking glass, and what we found running the Join Order Benchmark*, V.Leis et al.

Overview on Popular Join algorithms and an alternative: *New algorithms for join and grouping operations*, G. Graefe
Why should I care?

• A look at the query plan can help you identify the bottleneck of your query
• The Explain keyword is supported by many systems and shows the query plan, the physical operator, sometimes the cost (i.e. the runtime) of the operators, and some more or less useful additional information (e.g. the size of the relations and intermediates)

EXPLAIN (analyze) SELECT Mensa
FROM MensaMeals JOIN DailyOffers
ON MensaMeals.Meal=DailyOffers.Meal
WHERE Price < 5;

• Output can look different depending on DB system,
• Options might be available, e.g. analyze, timing on/off, buffers

Example output for join operator (PostgreSQL):
Hash Join (cost=0.00..5.37 rows=3 width=2) (actual time=0.00..2.222 rows=2 loops=1)
  -> Hash Cond: (MensaMeals.Meal=DailyOffers.Meal)
The mensa example is too small to generate interesting output
→ Switch to the Protein Database (PDB)
→ Create a more complex query

```
SELECT count(*) FROM (SELECT 1 FROM testsequence, authors WHERE Sequence_Count < 3 AND testsequence.PDB_ID = authors.IDCODE GROUP BY authors.author) foo;
```
Some show a graph (duckdb)...

...some show an ugly graph (sqlite)...

...and some show a formatted version of their internal RA representation (e.g. MonetDB, PostgreSQL)

→ This is where you are lost without Relational Algebra

\[ \sigma_{C_5 < 3}(X_{15}) \]

Uses English names instead of Greek letters for operators
Output of EXPLAIN

Additional Projection operator only needed in column-stores

- Find the data of the affected tuples (items with the same idx) in the arrays which store the remaining columns
- In row-stores, tuples are stored together, no lookup needed
B-Trees and Hashes are index structures
- Takes time to build
- Makes lookups faster
Task: Find all red entries

Trivial solution: Scan the whole dataset

Structures for faster searching:

Tree structures

Sorting

Hash-based structures (position defined by hash function)
Index structures

A good database system takes care for you of the index structures

But

• It is not always clear (e.g. if an index would come in handy for future queries)
• Not all systems are good systems → You can create an index yourself with CREATE INDEX

CREATE INDEX countindex ON testsequence (Sequence_Count);

An Index can be nested, e.g.

CREATE INDEX countindex ON testsequence (PDB_ID, Sequence_Count);

• Your query plan may or may not provide useful information on which attributes it is using in which sequence
• Nested indexes only work for the exact sequence they are made for, i.e. PDB_ID, or PDB_ID, Sequence_Count, but not Sequence_Count

15.12.2021
Database System

Parser → Optimizer → Engine

*strongly simplified*
Analytical queries usually read only a small number of columns, but all elements of these columns.

For parallel or pipelined execution, data must be split.

**Row Store**
- Intermediate tuples not stored, but passed directly to next operator
  - Operators can be fused
- Limited applicability of other optimizations, e.g. prefetching, vectorization, compression,…

**Column Store**

**Tuple-at-a-time**

**Vector/Block-at-a-time**
- A part (vector/block) of the column processed at once
  - Operator fusion only for small blocks
- Limited applicability of other optimizations, e.g. prefetching, vectorization, compression,…
  - Trade-off between operator fusion and memory access performance

**Operator-at-a-time**
- Whole operator (all elements of the column) processed at once
  - Intermediates materialized
  - No operator fusion, only coarse-grained parallelization
- High potential for optimization of memory reads
Why should I care?

- Different optimizations work with different processing models
- Your hardware limits your optimization space

Example A: You have a new intel server with the AVX512 instruction set for vectorization (under Linux, lscpu tells you if you have it; no root required)

- A system which implements only tuple-at-a-time is not able to use this instruction set

Example B: You do not have much main memory and writing to it is slow

- Materializing your intermediates becomes a bottleneck and might not work at all with operator-at-a-time or large blocks (block-at-a-time)

Vector registers can hold multiple values, e.g., up to 8 64-bit values with AVX512
The Effect of Optimizers and Engines

**SQLite**
- Ancient and extremely popular
- Row-Store
- Disc-centric
- Tuple-at-a-time processing
- Only 1 Join implementation
- Next to no query optimization

**DuckDB**
- New and completely fameless
- Column-Store
- In-Memory with out-of-memory option
- Vector-at-a-time processing
- Different Join implementations
- Optimizer actually does stuff (join order optimization, eliminate common subqueries,...)

**Load data from csv file (<300kB)**

**Simple query on small data (<300kB)**

**Complex query on slightly more data (~20MB)**

```
SELECT * FROM testsequence WHERE Sequence_Count < 3;
```

```
SELECT count(*) FROM (SELECT 1 FROM testsequence, authors WHERE Sequence_Count < 3 AND testsequence.PDB_ID = authors.IDCODE GROUP BY authors.author) foo;
```

- Reads and writes data to/from disk
- Reads from disk and writes to main memory
- Speed-up went from ~2x to ~78x (simple select vs. new query)
Database System

- Query
- Parser
- Optimizer
- Result
- Engine
- Data

*strongly simplified*
The 2 Flavours of Database Systems

Embedded DBS

Application includes/imports API provides interface Embedded Database System accesses Database

You Python, R, C++,… project e.g. Python module, C++ header,… Usually a library which does not need special permissions to be installed, e.g. a *.so file One or more files containing your data, format depends on your DB system

- No special permissions required
- No bells and whistles (no user management, distributed processing, multiple databases at once,…)
- Runs (almost) everywhere
- Porting to other platforms relatively simple (of course, a container makes it even easier)
The 2 Flavours of Database Systems

**Embedded DBS**

- **Application** includes/imports **API** provides interface to **Embedded Database System** accesses **Database**
  - You Python, R, C++,… project
  - e.g. Python module, C++ header,…
  - Usually a library which does not need special permissions to be installed, e.g. a *.so file
  - One or more files containing your data, format depends on your DB system

**Client-Server DBS**

- **Application/Client** includes/uses **API/Driver** provides interface to **Database Server** accesses **Database**
  - **Application/Client**
  - **API/Driver**
  - **Database Server**
  - **Database**
The 2 Flavours of Database Systems

- Usually requires installation as root
- Shipping the whole package (DB, DBS, Application) is challenging
  → Try a container
- Offers more features than Embedded DBS, e.g. user management, data partitioning, drivers for a standardized interface,…

Client-Server DBS
<table>
<thead>
<tr>
<th>Database</th>
<th>Storage</th>
<th>Column Processing</th>
<th>Physical Join Operators</th>
<th>Free?</th>
<th>Open source?</th>
<th>Embedded/Client-Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>Row store</td>
<td>-</td>
<td>Nested Loop</td>
<td>yes</td>
<td>yes</td>
<td>Embedded</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Row store, Column store</td>
<td>depends on extension</td>
<td>(Indexed) Nested Loop, Hash Join, Merge Join</td>
<td>yes</td>
<td>yes</td>
<td>Client-Server, 3rd party projects for embedding</td>
</tr>
<tr>
<td>MySQL</td>
<td>Row Store</td>
<td>-</td>
<td>Different combinations of Block-based, Indexed, Nested Loop, Hash Join</td>
<td>yes (community version)</td>
<td>yes</td>
<td>Client-Server, Embedded (commercial)</td>
</tr>
<tr>
<td>MonetDB</td>
<td>Column store</td>
<td>Operator-at-a-time</td>
<td>Different combinations &amp; variations of Partitioned, Indexed, Nested Loop, Hash Join</td>
<td>yes</td>
<td>yes</td>
<td>Client-Server</td>
</tr>
<tr>
<td>MariaDB</td>
<td>Column store + hybrid (multiple versions)</td>
<td>Block-at-a-time</td>
<td>Different combinations of Indexed, Block-based, Nested Loop, Hash Join</td>
<td>yes (community version)</td>
<td>yes</td>
<td>Client-Server</td>
</tr>
<tr>
<td>DuckDB</td>
<td>Column store (data blocks)</td>
<td>Block-at-a-time</td>
<td>(Indexed, Block-based) Nested Loop, Merge Join</td>
<td>yes</td>
<td>yes</td>
<td>Embedded</td>
</tr>
</tbody>
</table>

*Information comes from a wild combination of different sources including documentations, papers, and source code. It might be incomplete.*

There is no one fits all. Choose wisely!
<table>
<thead>
<tr>
<th>Database System</th>
<th>Use-Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>It's better than not having a database at all.</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Can do almost everything if you know which extension(s) you need. Might be overkill for what you need.</td>
</tr>
<tr>
<td>MySQL</td>
<td>For frequent transactions (e.g. insert new tuples), some features might not be free.</td>
</tr>
<tr>
<td>MonetDB</td>
<td>Good all-in-one solution for most analytical use-cases. Runs reliably on all sizes of machines, even on a laptop (and on my mobile phone).</td>
</tr>
<tr>
<td>MariaDB</td>
<td>Like MonetDB but with more features, e.g. different storages. Might be a total overkill for your project.</td>
</tr>
<tr>
<td>DuckDB</td>
<td>If you don’t need bells and whistles for your analytics (e.g. no user management, no server, limited variety of APIs).</td>
</tr>
<tr>
<td>Oracle</td>
<td>A database system on steroids, can deal with almost everything including geo-spatial data and data files in the PB range. Costs and arm and a leg, but DESY seems to have the money.</td>
</tr>
</tbody>
</table>

source: https://www.oracle.com/assets/technology-price-list-070617.pdf
## Database Systems

<table>
<thead>
<tr>
<th>Database Products</th>
<th>Named User Plus</th>
<th>Software Update</th>
<th>Processor License</th>
<th>Software Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MySQL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MonetDB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MariaDB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DuckDB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### SQLite
- It's better than not having a database at all.

### PostgreSQL
- Can do almost everything if you know which extension(s) you need. Might be overkill for what you need.

### MySQL
- For frequent transactions (e.g. insert new tuples), some features might not be free.

### MonetDB
- Good all-in-one solution for most analytical use cases. Runs reliably on all sizes of machines, even on a laptop (and on my mobile phone).

### MariaDB
- Like MonetDB but with more features, e.g. different storages. Might be a total overkill for your project.

### DuckDB
- If you don't need bells and whistles for your analytics (e.g. no user management, no server, limited variety of APIs).

### Oracle
- A database system on steroids, can deal with almost everything including geo-spatial data and data files in the PB range. Costs and arm and a leg, but DESY seems to have the money.

Source: [https://www.oracle.com/assets/technology-price-list-070617.pdf](https://www.oracle.com/assets/technology-price-list-070617.pdf)
Each library provides an own interface while SQL is (more or less) the same for each DB system.

Not offered by dplyr, pandas or other R, python, matlab,… libraries.

Storage and access inefficient with plain text files (e.g. csv).

Outside of a DB Engine: Limited choice of operator implementations + not state-of-the-art implementations.

*strongly simplified
What we did not cover

- Database design/Entity-Relationship-Model
- (Specialized) schemas, normalization
- Data compression and encoding
- NoSQL DBs (graph data, key-value stores,...) ➔ January
- Concurrent queries (scheduling, conflicts, anomalies...)
- Application interfaces/DB drivers (e.g. JDBC, ODBC)
  ➔ There are great tutorials, just ask the internet
- User-defined functions ➔ for everything you don‘t want to express with SQL
- Anything with a little more depth
Next Week

We will use DuckDB (because it will likely run on your laptop and is comparatively fast)

Import some data, send a few queries, and export the data

We will look at the query plans and create an index
Annett Ungethüm, 16.12.2021

Databases Under The Hood
An Introduction For The Curious User