My First Database
CCU Seminar Series

1 My First Query

We will use DuckDB for this little exercise. This is an Embedded Database System (no installation required)like SQLite. However, unlike SQLite it is a main memory database, organized in columns instead of rows, provides a decent query optimizer, and shows the query plans in a nicely formatted way.

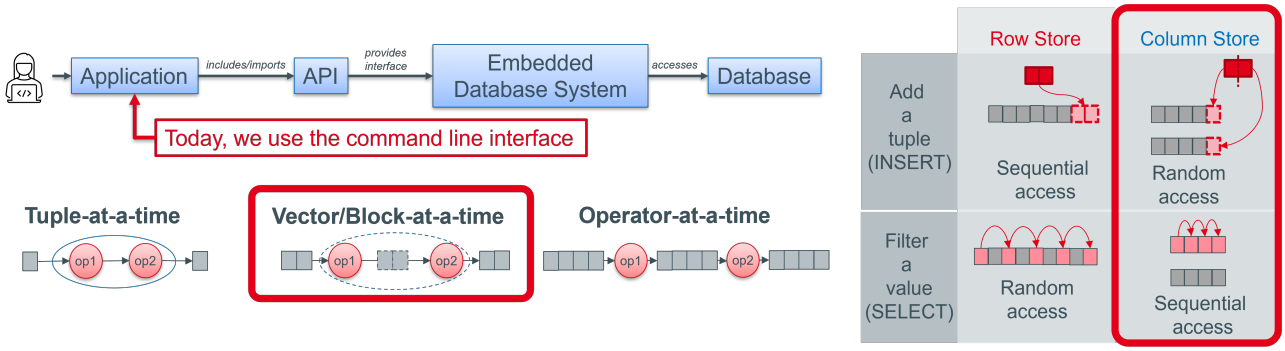

Figure 1: Duckdb is an embedded database system optimized for analytical query processing, i.e. queries which require only read-access but can be very complex.

Let's get started!

1. **Get some test data in csv format.** The examples in this exercise will use some data from the PDB, spefically a table with the compound names (`https://ftp.wwpdb.org/pub/pdb/derived_data/index/compound.idx`) and a table with the newly added sequence entries (`https://www.wwpdb.org/files/new_release_structure_sequence.tsv`). Remember where you save these files, you will need them later.

2. **Download and unzip the binary** of the command line interface for your system. You can find all downloads at `https://duckdb.org/docs/installation/`. Select the latest release, 'CLI', 'Binary', and your operating system to get the according link.

3. **Start the extracted binary.** There might be security warnings on windows and mac because for the system this looks like a random potentially malicious file that you could have downloaded from any shady source. Ignore the warnings.
   Congratulations, you have a database now! It's still empty, though.

4. **Load our data** into two separate tables. We call our tables 'compound' and 'sequence'.

```
CREATE TABLE sequence AS SELECT * FROM read_csv_auto('<path_to_your_file>
    new_release_structure_sequence.tsv');

-- compound.txt has a long header, so we provide the name of the columns and
    their data type to make sure parsing works as intended
CREATE TABLE compound AS SELECT * FROM read_csv_auto('<path_to_your_file>
    compound.idx', columns={'IDCODE': 'VARCHAR', 'COMPOUND': 'VARCHAR'});
```

   Congratulations, you just run a query!
   Note that the *read_csv_auto* function is not standard SQL, every database system has its own way of loading data from a file. But don't worry, usually it's simple.

5. Show some information about the tables we just created. We will see the number of columns (attributes), their name, type, if they are marked as NOT NULL, default values (if they exist), and if they are part

of the primary key (one or more attributes which uniquely identify a tuple). Note: This is again not standard SQL and works differently for every system.

```
PRAGMA table_info('sequence');
PRAGMA table_info('compound');
```

Is anything not as expected? You can drop the table and try it again:

```
DROP TABLE <tablename>;
```

Useful to know: You can leave duckDB with ctrl+d (linux) or ctrl+c (windows).

2 Fun With More Queries

Let's finally do some standard SQL queries. Try to run queries to return the following information:

- Find all sequences where the Sequence_Count is smaller than 5
- Show all new sequences for the ID '7SHK'
- Show all new sequences which contain 'AVGIGAV'
- Show all sequences for the compound named 'Structure of Xenopus laevis CRL2Lrr1 (State 1)' (Hint: You need both tables for this query)

Here are some example queries to help you:

- Select all sequences where the PDB_ID equals 5SB9

```
SELECT * FROM sequence WHERE PDB_ID = '5SB9';
```

- Select all sequences where the PDB_ID equals '5SB9' but show only the first and the last column

```
SELECT PDB_ID, Sequence FROM sequence WHERE PDB_ID ='5SB9';
```

- Select all entries from the sequence table, where the sequence contains 'RECISI'

```
SELECT * FROM sequence WHERE Sequence Like '%RECISI%';
```

- Show the number of new sequences for the ID '5SB9'

```
SELECT count(*) FROM sequence WHERE PDB_ID = '5SB9';
```

- Show the PDB_ID of all sequences with a Sequence_Count greater than 80. The DISTINCT keyword is optional and avoids redundant results.

```
SELECT DISTINCT PDB_ID FROM sequence WHERE Sequence_Count > 80;
```

- Show the name of the compound with the ID '5SB9'

```
SELECT COMPOUND FROM compound WHERE IDCODE = '5SB9';
```

- Show the name of the compounds where the Sequence_Count is greater than 80. You join both tables, i.e. merge all rows from both tables where the ID matches.

```
SELECT DISTINCT COMPOUND FROM compound, sequence WHERE sequence.
    Sequence_Count > 80 AND compound.IDCODE = sequence.PDB_ID;
```

3 Lists and Nested Lists

Let's get some numbers into our database! In this step, we store some random numeric data for some of our PDBIDs.

1. **Create a new table with lists and nested lists.** We call our new table 'metadata'.

```
CREATE TABLE metadata (PDB_ID VARCHAR, numbers FLOAT[], nestednumbers INT
    [][]);
```

2. **Insert some data into our new table**

```
INSERT INTO metadata VALUES ('5SB9', [2.5,1.00,2.3], [[0,1],[2,3],[4,5]]);

INSERT INTO metadata VALUES ('6ZNC', generate_series(0,2), [generate_series
    (0,3),generate_series(2,5),generate_series(5,10)]);

INSERT INTO metadata VALUES ('7AQQ', [0,0.3,1], [generate_series(0,2),
    [2,3], [4,5,6]]);
```

3. **Enjoy looking at your new table**

```
SELECT * FROM metadata;
```

4. Assuming your numbers in the column 'numbers' are 3d vectors in cartesian coordinates, we can **compute their length**.

```
SELECT pdb_id, sqrt(pow(numbers[0],2)+pow(numbers[1],2)+pow(numbers[2],2))
    AS length FROM metadata;
```

5. **Return the number of lists within each nestedlist**

```
SELECT array_length(nestednumbers) FROM metadata;
```

6. **Unnest *nestedlist*** and find out if any of the lists equals [2,3].

```
SELECT * FROM (SELECT pdb_id, unnest (nestednumbers) AS nr FROM metadata)
    foo WHERE foo.nr=[2,3];
```

4 Save Your Data

Save tables as csv

By default, DuckDB saves its state only in main memory. Thus, when you power off your machine or close your application, your data is lost if you didn't save it.

```
COPY sequence to 'my_sequence.tbl';
```

Open the file in a text editor. You see that the default format is different from the one in our original file. You can modify the output with some additional parameters, e.g. for changing the delimiter or including a header line.

```
-- Use a vertical bar as delimiter
COPY sequence to 'my_sequence.tbl' ( DELIMITER '|');
-- Use a tab space as delimiter and include the header line
COPY sequence to 'my_sequence.tbl' ( DELIMITER '        ', HEADER );
```

Save tables as parquet

Plain text files like csv are big compared to binary files. A lot of different formats have been developed to save space. The parquet format is a very common format for storing data on disk. It was originally developed for the Apache Hadoop ecosystem but now it also works with many other systems.

```
COPY sequence to 'my_sequence.parquet' (FORMAT 'parquet');
```

Compare the size of the files, pretty neat, isn't it?

You can load data from parquet files the same way you load from csv files:

```
CREATE TABLE mysequence AS SELECT * FROM 'my_sequence.parquet';
```

Or you can run your queries directly on the file.

```
-- Show the number of records/rows
SELECT count(*) FROM 'my_sequence.parquet';
```

Note that reading from disk can take a while compared to reading from main memory, especially when the file is big.

Persistency In One File

You can create a persistent database by providing a name for the database when starting duckDB. This automatically stores all the data you produce in a single file instead of one file for each table. The suffix .db is automatically added.

```
.\duckdb my_fresh_pdb.db (linux)
```

If the given name already exists, the contained database will be opened. You may also create such a file with sqlite (sqlite3 does not work, yet) or take the one provided via indico (*pdb_test.db*). *pdb_test.db* already contains our two tables. You can load it when starting duckDB:

```
.\duckdb pdb_test.db (linux)
.\duckdb.exe pdb_test.db (windows powershell)
```

An indermediate log file (*.wal) is created after starting duckDB. This file is updated whenever a transaction is finished (create, update, insert, drop, everything that changes your data instead of only reading it). Try to change something in your database and see how the timestamp of your file changes, e.g.

```
CREATE TABLE mynewtable (id INTEGER PRIMARY KEY, comment VARCHAR);
```

Now leave duckDB. Your original *.db file is updated, the intermediate file is deleted.

- Every system has a maximum file size. The exact limit depends on your file system and the configuration of the operating system. Especially flash drives are usually pre-formatted using FAT32 for compatibility reasons, which supports only files smaller than 4GB.
- Of course you can also use this to import your old and rancid sqlite database into duckDB.
- Database files might not be visible if they do not contain any tables.

5 Dissecting Queries

1. **Run a query using the EXPLAIN keyword**, e.g.

```
EXPLAIN SELECT DISTINCT COMPOUND FROM compound, sequence WHERE sequence.
    Sequence_Count > 80 AND compound.IDCODE = sequence.PDB_ID;
```

We get the query plan, i.e. all executed operators in a tree structure.

Note: Windows might show garbage when using EXPLAIN. Here is how to still get some nice output:

- Enable profiling: **PRAGMA** enable_profiling;
- Define where your output will be saved: **PRAGMA** profile_output='queryPlan.txt';

- Run your query without the EXPLAIN statement
- Open *queryPlan.txt* and enjoy your nicely formatted query plan

2. Now we want some **additional information**, i.e. the time our query needs in total and for each operator. To get this, turn on profiling if you haven't done it yet:

```
PRAGMA enable_profiling;
```

3. Now run the query again to **see the timings**. The query is only short running on a small set of data (and multiple rounding errors sum up for the overall timing), but if your hardware is slow enough, you can identify the operator which is the bottleneck in our query. We can speed this operator up by building an index on the used attributes.

6 Indexes

An index can help to speed up you query. It is usually implemented as a tree and can cover one or multiple attributes. *Some* indexes are built automatically by *some* systems. DuckDB automatically builds an index for every individual column, but we can do better.

1. Find the operator with the longest runtime → SEQ_SCAN
2. Find the table(s) and attribute(s) used in this operator → Compound(IDCODE, COMPOUND)
3. Build an index on this/these attribute(s). We will call our index 'cpn'.

```
CREATE INDEX cpn ON compound(compound, idcode);
```

4. Run your query again and look how fast your operator has become (in case it is slow enough to see a difference at all). The speed-up varies heavily depending on your system and any additional processes you are running.

Some indexes are stored if you use duckDB with persistency. Hence, index creation may fail if you try to create a duplicate. Furthermore the runtime difference might not be there because you already had an index.

Further Reading

If you are convinced now that a database might be helpful, there are a lot more features to explore, e.g. sampling, views, a selection of numeric functions,... Most of these features are also available in other database systems, so it's worth knowing that they exist. There are also APIs for C/C++, Java, R, and Python. You can find an extensive documentation at `https://duckdb.org/docs/`. For example code for the APIs, you may also refer to the github of duckDB: `https://github.com/duckdb/duckdb/tree/master/examples`. On top of this, there is a lot to explore beyond relational databases, e.g. document stores, array databases,...