Assignment 1

Uploading Data to the Database

Database Tuning

A4
Günter Lukas, StudentID1
Balint Peter, StudentID2
Ottino David, 51841010

March 24, 2025

Experimental Setup

We used Python to implement a program to import the files into the database. The database used is PostgreSQL in its most current version. We use psycopg2/3 as an adapter to interact with the database. For the portability-section we decided on SQLite. Everything is done locally on our private computers.

Straightforward Implementation

Implementation In the naive approach we just establish a connection to the database, loop over the file and create an insert statement for each entry in the input file. Each entry is commited right away by itself. This leads to a high number of small, individual transactions which are written to the disk separately and commited right away. An example statement would look something like this:

```
INSERT INTO Auth (name, pubID) VALUES ("Jurgen Annevelink",
"books/acm/kim95/AnnevelinkACFHK95");
COMMIT;
```

In this approach, it took about 10 seconds to insert 20.000 lines.

Efficient Approaches

Efficient Approach 1: (Batch Input)

Implementation In the first optimized approach we utilize batching. Multiple lines are read from the file, processed and put into a batch. When the batch reaches its batch size, all entries of the batch are inserted as one statement. After all lines are processed, the remaining rows in the batch are also inserted. So if a batch size of 1000 is used, 1 query with 1000 entries is inserted. The statement executed by Postgres would look something like this:

```
INSERT INTO Auth (name, pubID) VALUES ("Jurgen Annevelink",
"books/acm/kim95/AnnevelinkACFHK95"), ("Rafiul Ahad",
"books/acm/kim95/AnnevelinkACFHK95"), (...);
```

After all batches are executed for both tables, all changes are commited at once.

Why is this approach efficient? This approach uses far fewer total statements as the naive approach. This greatly reduces the SQL-overhead of parsing and optimizing each statement separately. It also leads to a more efficient use of transactions since tables are locked less often for the write operation and the data is written in one go to the storage. This also means that fewer network roundtrips are required and therefore network overhead is also reduced.

Important: Cite the references that you used to answer this question, for example, using footnotes or the References section at the end of the report.

[Your answer goes here ...]

Tuning principle The tuning principle that describes this approach best is "Startup costs are high, running costs are low". By inserting multiple entries with one Insert-statement, we create fewer overheads than we would have if we would just execute each row as a separate statement.

Efficient Approach 2: (Copy instead of insert)

Implementation In the second optimized approach a feature of PostgreSQL is used to transfer the data more efficiently. The copy-function, while being less flexible than an Insert-statement, is specifically designed to load rows from a given file into a database table by instructing the database server to read the specified file directly. Also, in this approach WAL-logging is disabled and working memory is increased.

Why is this approach efficient? The copy-functions sole purpose is to read data from files and insert them into database tables (or the other way around), therefore it is highly optimized for this operation. The data is written in large chunks, which reduces overhead and since copy is a single command only one commit per table is issued.

Tuning principle The applied tuning principle is again "Startup costs are high, running costs are low", since copy groups all the inserts into one statement.

Portability

Implementation We decided to use SQLite as our second database system. In order for this to work different code has to be executed in all parts of the program that interact with the database. To differentiate between the two a parameter was implemented that triggers if sqlite is used or not and based on this parameter different code segments are executed. Establishing a connection to the database requires authentication in the case of PostgreSQL while SQLite allows a direct connection. Also the wildcards differentiate between the two:

```
INSERT INTO Auth (name, pubID) VALUES (%s, %s); --Postgres
INSERT INTO Auth (name, pubID) VALUES (?, ?);   --SQLite
```

In the case of batch inputs SQLite also differs from Postgres by providing an executemany-function which can be used with the batch directly. On the other hand there seems to be no equivalent to copy in SQLite so there was no real way of porting the second approach to SQLite.

Did you observe performance differences. If so: Why. If not: Why not? Interestingly, SQLite performed the batch input much faster than Postgres. We assume this is because SQLite has much less overhead in general and is therefore able to perform those transactions faster. Also, since SQLite is a local database while Postgres uses a client-server-model which introduces additional overhead even if the client and the server run on the same machine.

Runtime Experiment

Notes

- We decided to limit the import to 10.000 tuples per table in the naive approach.
- Everything is done locally, the input files are downloaded on our computer and the database as well as the python program also run there.

Approach	Runtime [sec]
Straightforward	10.8350 sec.
Approach Batch Input	51.2956 sec
Apporach Copy	9.2371 sec
Local Approach	...

Time Spent on this Assignment

Balint Peter: **XXX** Günter Lukas: **XXX** Ottino David: **3h**

References

https://www.postgresql.org/docs/current/populate.html
https://www.postgresql.org/docs/current/sql-copy.html