Assignment 1

Uploading Data to the Database

Database Tuning

New Group 8

Frauenschuh Florian, 12109584

Lindner Peter, 12101607

Weilert Alexander, 12119653

March 29, 2024

Experimental Setup

For our experiments we used the following hardware and software:

Component	Specs
Processor	i7-13700H 3.7-5.0 GHz
Memory	32 GiB

Table 1: Hardware: Dell XPS 15 9530

Software	Version
OS	Ubuntu 22.04
Postgres	2.3.4
postgresql	42.7.3
MariaDB	10.6.16
mariadb-java-client	3.3.3
Java	18

Table 2: Software

Both Postgres and MariaDB were hosted on localhost, on which we also executed our experiments. The client was implemented in Java and gained access to our databases using the JDBC drivers listed above. Before executing any tests, we initially created a database for each DBMS by using:

```sql
CREATE DATABASE db_tuning_1;
```

Listing 1: Create database

Additionally, we programmed our client to create the `auth` table on each DBMS if necessary by executing:

```sql
CREATE TABLE IF NOT EXISTS auth (name character varying(49), pubid character
    varying(149));
```

Listing 2: Create `auth` table

Straightforward Implementation

Implementation In the straightforward implementation we read the input file `auth.tsv` line by line and instantly insert each tuple individually:

```
INSERT INTO auth (name, pubid) VALUES(name_val, pubid_val);
```

Listing 3: Insert tuple individually

We note that we used the `PreparedStatement` objects in order to simplify the implementation of our client:

```
...
PreparedStatement insertAuthorStatement = connection.prepareStatement("INSERT
    INTO auth (name, pubid) VALUES(?, ?)");
...
insertAuthorStatement.setString(1, lineParts[0]);
insertAuthorStatement.setString(2, lineParts[1]);
insertAuthorStatement.execute();
...
```

Listing 4: Implementation: Straightforward approach

Hence, we need as many SQL-statements as we have lines, or tuples, in the input file.

Efficient Approaches

Efficient Approach 1: Batch Insert

Implementation In this approach we utilize the batch functionality of the `PreparedStatement` object. Instead of inserting each tuple individually, we add all tuples into a single `INSERT` SQL-statement.

```
...
PreparedStatement insertAuthorStatement = connection.prepareStatement("INSERT
    INTO auth (name, pubid) VALUES(?, ?)");
...
insertAuthorStatement.setString(1, lineParts[0]);
insertAuthorStatement.setString(2, lineParts[1]);
insertAuthorStatement.addBatch();
...
insertAuthorStatement.executeBatch();
...
```

Listing 5: Implementation: Batch approach

To avoid the DBMS to commit during the creation of our batch, we deactivated the auto-commit functionality before our implementation and activated it again afterwards.

```
...
connection.setAutoCommit(false);
...
connection.setAutoCommit(true);
...
```

Listing 6: Implementation: Auto-commit in batch approach

We note that the SQL-statement itself remained the same as in the straightforward approach.

Why is this approach efficient? Batch inserting is an efficient technique to improve performance while inserting multiple tuples into a database. This approach avoids unnecessary overhead that occurs while executing an `INSERT` SQL-statement for each individual tuple. Since now the amount of SQL-statements we have to transfer from the client to the database is much smaller, we also reduced potential overhead due to the round-trips we would have to take otherwise. In addition, the DBMS has much less work in terms of logging in comparison to multiple `INSERT` statements [1].

Efficient Approach 2: Copy/Load Data Infile

Implementation In this approach we make use of the `COPY` and `LOAD DATA INFILE` functionalities of Postgres for the former and MariaDB for the latter. While utilizing these features, the DBMS directly copies the tuples from the file into the database, keeping the required actions of the client to an absolute minimum.

```
...
String sqlCopy = "COPY auth (name, pubid) FROM '" + "/tmp/" + inputFile + "'
    DELIMITER E'\t'";
...
Statement copyStatement = connection.createStatement();
...
copyStatement.execute(sqlCopy);
...
```

Listing 7: Implementation: `COPY` functionality of Postgres

```
...
String loadSql = "LOAD DATA INFILE '" + "/tmp/" + inputFile + "' INTO TABLE
    auth FIELDS TERMINATED BY '\t' " +
             "LINES TERMINATED BY '\n'";
...
Statement loadStatement = connection.createStatement();
...
loadStatement.execute(loadSql);
...
```

Listing 8: Implementation: `LOAD DATA INFILE` functionality of MariaDB

We note that both functionalities are not part of the SQL standard, hence they use DBMS specific syntax. Additionally, each DBMS is required to have direct access to the input file, which we solved by placing the `auth.tsv` file into the `/dev` directory.

Why is this approach efficient? For Postgres the `COPY` functionality is always faster than a batch insert [2]. According to [3], `COPY` enables Postgres to handle its buffers more efficiently than it is able to while using a batch insert. Hence, this results in a faster execution time. In case of MariaDB and `LOAD DATA INFILE`, the performance boost is a result of a more efficient way of reading and caching the input data. For some transactional engines MariaDB even omits the logging functionality in order to increase the performance further [4].

Tuning principle In this approach we apply the “Render on the Server What Is Due on the Server” principle. As already mentioned, we reduced the work for the client to a minimum and shifted the workload towards the DBMS, which has to insert the tuples anyways.

Portability

Implementation To begin with, there are obviously some differences in the connection addresses for each DBMS. For the straightforward approach and the batch insert approach there are no changes needed in the code. The biggest difference in implementation occurs within the `COPY/LOAD DATA INFILE` approach. Here we have to consider the required syntax of each DBMS in order to execute the transaction accordingly. The differences are already portrait during our discussion of the second efficient approach, hence we omit these here.

Did you observe performance differences. If so: Why. If not: Why not? During our experiments we noticed that Postgres has a slight advantage against MariaDB in the straightforward approach. For the batch insert approach, MariaDB required a tad less time than Postgres. The biggest performance difference occurred during the `COPY/LOAD DATA INFILE` approach. Here we noticed that Postgres outperformed MariaDB by quite some bit, as we can see in the table below. This could be due to a more efficient implementation of caching in Postgres, as discussed in [5]. The difference for the straightforward approach follows [6], where Postgres outperforms MariaDB on average as well.

Runtime Experiment

Approach	Runtime [sec]
Straightforward	1077.3
Batch Insert	12.3
`COPY`	1.1

Table 3: Performance Postgres

Approach	Runtime [sec]
Straightforward	1125
Batch Insert	10.5
`LOAD DATA INFILE`	9.6

Table 4: Performance MariaDB

Notes

- As already mentioned, both DBMS were hosted locally on the same machine as the client

Time Spent on this Assignment

Time in hours per person: **8**

References

[1] Antonino Cencio. *SQL Performance Killers: Individual Inserts vs Bulk Inserts.* https://dev.to/antoninocencio/sql-performance-killers-individual-inserts-vs-bulk-inserts-51h4. 22.03.2024. 2020.

[2] Jose Zamora. *Speed up your PostgreSQL bulk inserts with COPY.* https://dev.to/josethz00/speed-up-your-postgresql-bulk-inserts-with-copy-40pk. 22.03.2024. 2020.

[3] pganalyze. *5 Minutes of Postgres: Optimizing bulk loads - COPY vs INSERT.* 22.03.2024. 2020. URL: https://pganalyze.com/blog/5mins-postgres-optimizing-bulk-loads-copy-vs-insert.

[4] MariaDB. *How to Quickly Insert Data into MariaDB.* https://mariadb.com/kb/en/how-to-quickly-insert-data-into-mariadb/. 22.03.2024. 2021.

[5] Kinsta. *MariaDB vs PostgreSQL: A Detailed Comparison.* https://kinsta.com/blog/mariadb-vs-postgresql/. 22.03.2024. 2021.

[6] Cal Mitchell. *PostgreSQL, MariaDB, and HammerDB Deployment: A Performance Analysis.* https://www.albatrossmigrations.com/blog/mariadb-postgresql-performance. 22.03.2024. 2021.