OULUN AMMATTIKORKEAKOULU

Emma Tauriainen

DATABASE UTILIZATION IN EMBEDDED SOFTWARE DEVELOPMENT

DATABASE UTILIZATION IN EMBEDDED SOFTWARE DEVELOPMENT

Emma Tauriainen
Bachelor’s Thesis
Spring 2023
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, Option of Device and Product Design

Author: Emma Tauriainen
Title of thesis: Database Utilization in Embedded Software Development
Supervisors: Petri Honkala and Petri Saari (Nordic Semiconductor) and Jukka Jauhiainen (OUAS)
Term and year when the thesis was submitted: Spring 2023
Number of pages: 44 + 4 appendices

The topic of this thesis was database utilization in embedded software development and it was commissioned by Nordic Semiconductor. This work was intended to develop and improve software quality and performance metrics tracking applications by implementing a database with the most suitable features into use.

To find the best choice, five different databases, two SQL-based and three NoSQL-based, were taken into comparison. The databases to be compared were selected based on their popularity and comments from the IT support of the company. When comparing them, their properties were presented and discussed in relation to the requirements of the company. For database transaction model, ACID-model was selected, which left two databases for comparison, PostgreSQL, and MS SQL Server. These two were compared by benchmarking of which MS SQL Server was selected. The database was created to the company's server and database handler script was implemented using Python programming language. The structure of the database handler was planned and implemented to be as generic and scalable as possible. Table structure of the database was designed as efficient and clear as possible as well.

The result of this thesis was a successful entity which consisted of MS SQL Server, its handler script, and an application used to make queries to the database with. For further development a web user interface and interactive visualization web application were discussed.

Keywords: Relational database, SQL, NoSQL, ACID model, benchmarking, MS SQL Server, Python

ACKNOWLEDGEMENT

First, I would like to thank Nordic Semiconductor for the opportunity to complete this thesis. Specifically, I would like to thank Petri Saari and Petri Honkala for being my supervisors from the company. They offered excellent guidance, help, and support during this thesis. I would also like to thank my supervisor Jukka Jauhiainen from OUAS for giving feedback, tips, and perspective on the work.

Additionally, I would like to thank the company's IT support for their comments, tips, and help in defining comparable databases and implementing test databases.

In Oulu 23.2.2023
Emma Tauriainen

CONTENTS

ABBREVIATIONS

ACID	Database model (Atomicity, Consistency, Isolation, Durability)
BASE	Database model (Basically Available, Soft State, Eventually Consistent)
CPU	Central processing unit
FK	Foreign key, field that refers to primary key
GUI	Graphical user interface
IoT	Internet Of Things
JSON	JavaScript Object Notation, an open data interchange format
LAPP	Linux-Apache-PostgreSQL Application server
NoSQL	Non-SQL, No SQL -based database
ORDBMS	An open-source object-relational database management system
PK	Primary key, column in a relational database table, distinctive for each record
RAM	Random-access memory, computer memory’s form, typically used to store working data and machine code
RDBMS	A relational database management system
SQL	Structured Query Language
Tps	Transactions per second
TSDB	Time series database
UI	User Interface

1 INTRODUCTION

Embedded software development focuses on embedded systems which are devices built around a microprocessor where the software is programmed into. Embedded software development and its quality monitoring produce different kinds of metrics that need to be stored utilizing a database. The database is a collection of data to which the data, metrics, are stored usually in electronic form. With the database, the metrics can be monitored and tracked for example to ensure the quality of the software implementation or for other possible further development. These metrics can be, for example, power consumption, the execution time of the system, and free space left in memory. In embedded software development the databases can be embedded in the systems, or they can be external. In this work, the implemented database is external in the company's server.

The commissioner of this work, Nordic Semiconductor, is a fabless semiconductor company from Norway focusing on wireless communication technology that powers the Internet of Things (IoT) (1). The idea behind this thesis was triggered by the need to store and monitor data used in tracking the software quality and performance metrics. When developing software, whether in embedded software development or not, publishing the software does not mean forgetting it. Instead, its outcome, quality, and maintainability must be monitored. With different software quality and performance metrics tracking applications, the development maintains for example cost-effectivity since from these can be seen if the changes affect the software. If they affect, reacting to them in early stage is possible. This work intends to develop software quality and performance metrics tracking applications for which the selected database will eventually be taken into use. In addition, the implementation will be demonstrated with a demo application.

To select the database meeting the company's needs the best, five different databases are compared and the best one of them is selected and taken into use. This work provides one final choice and implementation of the database around which the applications can be built. The result of this work secures the data storage, prevents data losses, or restores the data if needed.

2 DATABASE COMPARISON

Databases were chosen based on their types and upon the usage requirements. These types are, for example, an object-oriented database, a relational database, or personal database where the database is a single file where data is written and read. File-based database, for example, JSON file located in a network drive is not a very safe and efficient choice for a database. If a database is in use for several years, file-based database will become slow due to its increasing file size. Some file systems may cause limitations on file sizes, or a library can be made with some programming language, for example, C or C++, which could have memory limitations, too. However, for example Python is a dynamic programming language so file-based database depends on the computers' memory when handling large amount of data. Using JSON file database in network drive might cause connection problems which could end up corrupting the file.

A proper database would make its applications more reliable and secure since the database most likely has proper back up methods, less data corruption, and its performance is quicker from which the company or any operator benefits. Moreover, it makes configurability easier and increases the usability of the applications that use the database. For example, if applications are changed, the changes can be done to the application-side, and there is no need to modify the database. Accessibility and visibility of the data can also be delimited so that it is not misused or damaged.

In addition, using proper database makes handling the database more straightforward; for example ready-made libraries can be used rather than spend time on making one of its own. If the data consists of small data sets with arbitrary and unrelated data, for example file-based database, solution might be more efficient than a proper database. However, for this work file-based database is not an option due to the needs of the company.

Database selection is a big decision considering its long-term use and data change. Correct selection ensures that the applications that use the database will be available, performant, and scalable over time. Therefore, comparison between different databases is desirable.

2.1 Database options and requirements

The data to be stored and monitored is from tracking the quality and performance of the software metrics in embedded software development; this means that the database must be able to handle a large amount of data. As mentioned in the introduction, the database itself will not be located in any embedded hardware device of the company but rather in their server giving lots of options for its selection.

For comparing different options, five different databases, two SQL-based and three NoSQL-based databases were selected. The options were chosen based on their popularity, availability, and the view of the company's IT support. MySQL (2), one of the most popular databases was left out from the comparison because the company's current version is MySQL community, which is based on mostly non-critical and light usage, and its restore handling is not very reliable. Since a better option, MS SQL Server (3), is available and the company already uses it, it was taken in place of MySQL. If MySQL was used, a commercial version that has better restore handling would have been chosen.

A Structured Query Language (SQL) is a standard programming language that executes queries to a database. It is used with relational databases where the data is stored in tables. The tables are collections of related data entries consisting of columns and rows. (4.) Non-SQL (NoSQL) databases deviate from the traditional relational model. These databases are non-tabular and their type is based on their data model. Document, key-value, wide-column, and graph are the main types for NoSQL databases. (5.)

The database must be able to store a large amount of data and has proper handling of backups. When thinking about data post-processing and monitoring, the possibilities for graphical data presentation, continuous monitoring, and the possibility to add own notes must be considered as well as the possibility for user interface (UI) and configurability. The database must also be able to use with Python programming language (6) and its respective libraries or the database must have drivers that enable the usage.

2.2 Microsoft SQL Server

Microsoft SQL Server (MS SQL Server) is SQL-based, RDBMS, and it was developed by Microsoft in April 1989. Nowadays it works both on Windows and Linux and it is used commercially a lot for example by Atlassian Corporation's tools, work management tool Jira, a team workspace Confluence and version control system repository management solution Bitbucket. Database queries use Microsoft's variant of SQL called Transact-SQL (T-SQL) (7) which adds a set of proprietary programming constructs. It supports different data types including the primitive types such as integer, float, and decimal (8).

For online backup strategies, there are three recovery models: simple, full, and bulk-logged. The most recommended one is full recovery model, which permits no data loss making database recovery possible at any time if backups are up to date at that time. (9.) Simple recovery model is also highly used. It gives a simple backup and ability to do an entire copy of the database or backup with any changes since the last complete backup. This may expose database to failures because the data can be restored only from the point when the backup occurs. (10.) Bulk-logged recovery model permits high performance bulk copy operations. It uses minimal logging for most bulk operations reducing log space but log backup sizes can be substantial because a log backup has as few logged operations as possible. However, the choice of the recovery model is not final as it can be changed at any time. (11.)

One of the features of MS SQL Server is that via a software application SQL Server Management Studio task can be scheduled thus enabling continuous monitoring (12). Also, its configurability is simple by using the application or T-SQL but it is not available for all users. Only the system administrator, database owner, and members of the sysadmin can change the configuration settings for the database. (13.)

MS SQL Server would be reliable choice because it has been around for a long time and has several big clients. It would also give flexibility with data types since future data types are not known. Additionally, this database manages to handle large amount of data and supports Python which would meet some requirements of the company. T-SQL is an extension of SQL and when compared these two T-SQL contains procedural programming and local variables with more options for database handling. Adding own notes and graphical data presentation could be handled with T-SQL. T-SQL also supports join-clauses which might be a good addition if the database is chosen to be a relational database. Therefore, the data structure and its format definition would be straightforward, and data post-processing and follow-up should not cause bigger problems.

MS SQL Server supports partial indexes as well as automated functionality for index management which is an important part in usability of databases. Overall, due to its pivotal features, robust security platform, industry-leading performance, and intelligence across all the data with big data clusters, MS SQL Server is a great option.

2.3 PostgreSQL

PostgreSQL is a free, open-source object-relational database management system (ORDBMS) (14). It emphases extensibility and SQL compliance and runs on all major operating systems. PostgreSQL supports various data types such as primitives, structured, and documents. Users can also create their own data types giving flexibility for future data types. It can also store a large amount of data and it uses primary keys, foreign keys, and exclusion constraints to keep data integrity. PostgreSQL supports index-based table organization and stored functions and procedures as well as multiple programming languages like Python and Perl. (14.) With these features data structure and its format definition would be simple making data post-processing and follow-up possible.

With its capability of extending, PostgreSQL gives multiple options for data handling. One of its use cases is LAPP (Linux, Apache, PostgreSQL, Perl, PHP, and Python), an open-source stack for running dynamic apps and websites. This would be beneficial for future developing, for example developing a web UI. Along with this, it has support for JSON and foreign data wrappers allowing it to link with other data stores. PostgreSQL's system liberates the disk space by looking for empty rows and eliminates unnecessary elements by scanning the table of a data layer. This needs a lot of CPU, which could impact performance, even though it may not be a problem in this implementation. For repetitive tasks, PostgreSQL does not have support for a built-in job scheduler but it supports external tools like Cron or Task Scheduler which are already in use in the company. (12.)

For backup utilities there are built-ins such as pg_dump and pg_dumpall. Also, third-party tools are available such as Amandam NetVault Backup and Bacula (9).

For graphical data presentation PostgreSQL could be used with Grafana which is a multi-platform open-source analytics and an interactive visualization web application. Grafana provides charts, graphs, and alerts and it is used for composing observability. (15.) It would expand the usage and usability vastly in graphical data presentation.

2.4 MongoDB

MongoDB is a source-available document-oriented and a NoSQL database program (16). It uses JSON-like documents meaning that data structure is composed of field and value pairs. The data structure makes storing structured or unstructured data easy for developers and they do not have to think about the data normalization. MongoDB's advantage is that it also provides high performance data persistence. (17.) MongoDB was released in February 2009 making it quite a new database compared to MS SQL Server. Nevertheless, it has become one of the most popular databases, as shown in figure 1 below.

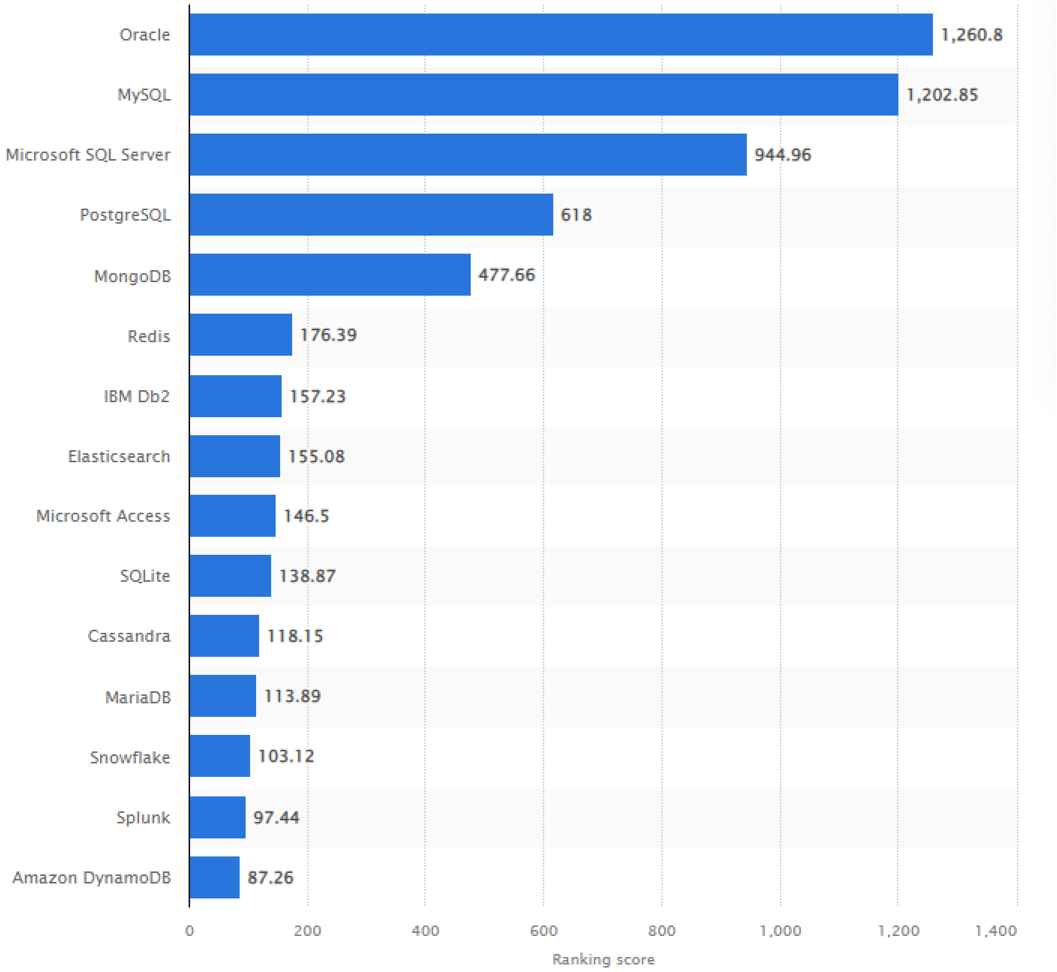


Figure 1. The most popular databases worldwide in August 2022 (18.)

MongoDB works well with real time analytics, content management, and other types of applications. With unstructured or structured data with rapid growth potential, it is an ideal choice. However, in this case the data increases steadily. MongoDB's best features are its performance levels, simplicity, and high speed and higher availability. It stores most of the data in random-access memory (RAM) allowing a quicker performance during execution of queries. The data is stored as documents in compressed Binary JavaScript Object Notation (BSON) files which are binary encoded JSON textual object notations The data can be retrieved directly in JSON format making it easy to read and understand. Since MongoDB is a document-based database, its attributes like replication and gridFS allow an increase in data availability. In addition, using indexing and document accessing is easy. On the other hand, MongoDB does not support transactions because high performance requires the right indexes making data corruption possible. If they are out of order or shoddily implemented, it will operate at slow speed. With some of these downside features it could lead to duplicate or corrupted data which does not make it quite reliable choice. (19, 20.)

MongoDB has Python support making it possible to add your own notes and do data post-processing and reasonable follow-up. MongoDB offers backup methods with Atlas (21), MongoDB Cloud Manager (22) or Ops Manager (23) and mongodump as well as copying underlying data files (24). It also offers a graphical user interface (GUI), MongoDB Compass for querying, aggregating, and analyzing data. MongoDB Compass is free and the source available (25) and supports macOS, Windows, and Linux making MongoDB Compass one option for graphical presentation of data.

2.5 Cassandra

Cassandra is also an open-source, NoSQL database (26). It uses a traditional model with a table structure with columns and rows. In addition, it provides an SQL-like programming language, Cassandra Query Language (CQL) for creating and updating database schema and accessing data. CQL supports features such as user-defined types, functions and aggregates and local secondary indices as well as modifying data in existing tables. However, cross partition transactions, distributed joins and foreign keys or referential integrity are not supported by Cassandra because they require cross partition coordination as they are typically slow. Cassandra supports incremental backups where data can be backed up as it is written making it more robust. Cassandra is a distributed database and is designed to handle massive amounts of data across many commodity servers. This provides high availability with no points of failures. The downside of this is latency issues due to transactions slowing down. Still, selecting Cassandra would offer scope and reliability at least compared to MongoDB. A disadvantage of Cassandra is that like in some NoSQL databases, subqueries and joins are not supported. (26.)

In Cassandra the data post-processing and follow-up differs from RDBMS. For example, returned data’s order in RDBMS can easily be done with Order by query or by default the records will return in the order in which they are written. In Cassandra the sorting is defined by the order specified by the clustering columns which are defined when a table is created. This may not be a good solution for the company’s needs because all use cases of the data usage cannot be known beforehand or the data structure may change in the future. Also, the database structure might get confusing if there are a lot of different tables for different queries. (27.)

Cassandra is configurable and it has a comprehensive documentation about configuration on its website. Cassandra itself does not offer GUI but it is supported by various database designers like DbSchema. DbSchema provides a diagram-oriented database designer and GUI tool for SQL and NoSQL databases, in addition to Cassandra, also for MS SQL Server, PostgreSQL, and MySQL, for instance. (28.)

Since Cassandra supports Python with Cassandra Driver, it is possible to create own graphs from data if a third-party GUI is not wanted to be used. Even though Cassandra is not a very popular database, as shown in the figure 1, it has some of the biggest companies as users such as Facebook, Twitter, and Reddit. (29.)

2.6 InfluxDB

InfluxDB is an open-source time series database (TSDB) with a rich library (30). It is developed by InfluxData for storing and retrieving time series data meaning that a series of data points are indexed in time order. The data can be for example IoT data, real-time analytics, or application metrics. (30.) InfluxDB has its own data scripting language called Flux, which is designed for querying, processing, writing, analyzing, and acting on data. However, Flux can be also used with other sources like SQL Databases (For example MS SQL Server), annotated CSVs and JSON. It would make adding own notes, data post-processing and follow-up possible as well as configuring the database. (31.)

Since InfluxDB is used for storing time series data, it has great support for large amounts of data. It makes data storage one of its strengths and would meet the requirements of the database selection.

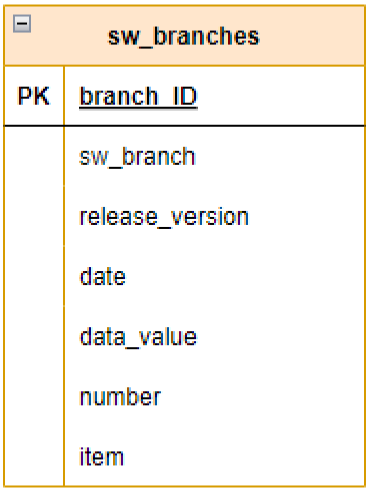
InfluxDB has support for client libraries including Python to integrate InfluxDB to programming languages' scripts and applications. However, many data accessing methods are not currently provided which could cause issues to implementation side. (32.) Data visualization is possible with the InfluxDB UI. It provides multiple visualization types to visualize data such as a gauge view and band visualization. Currently, a graph view is not available yet. (33.) However, InfluxDB is supported by Grafana which initially targeted time series databases (15). InfluxDB backs up data and metadata by copying them to a set of files which are in a specified directory on user's local filesystem (34). This backup method is easy to implement and to access but if a local filesystem is used as a place for backup, user should carefully decide the place for backup to avoid possible corruptions. A safer option would be a cloud backup which, for example, MongoDB provides (25).

3 DATABASE SELECTION

There are two different database transaction models: ACID (Atomicity, Consistency, Isolation and Durability) model, compliant with relational databases, and BASE (Basically Available, Soft State, Eventually Consistent) model, compliant with NoSQL databases. BASE model provides high availability and focuses on flexibility and speed while ACID provides a consistent system ensuring the integrity of the database. In ACID model, Atomicity means that all the operations must be executed or none of them are leave no partially executed transactions. Consistency means that after any transaction the database must remain in a consistent state and no transaction should have an adverse effect on the existing data. Isolation means that transactions cannot affect each other and if there are unfinished executions, they cannot be visible to other transactions. Durability makes sure that even if the system fails or restarts, the database is durable enough to hold all its latest updates. Basically, Available in BASE refers that in NoSQL databases the data availability is spread and replaced across the database cluster's nodes instead of making immediate compulsory consistency of it. Soft State refers to database delegating its responsibility from its consistency to developers. Eventually, Consistent means that even though BASE does not obligate immediate consistency, it does not mean that it never achieves it. However, data reads are possible until the consistency is achieved. Typically, ACID models' code type is simpler than in BASE model. (35.)

After the research work, it was decided that ACID model was selected due to its four properties providing a reliable and robust database. With a BASE model, the developers should be very aware of data of the database. In addition, an often-recommended relational database was chosen for the implementation. If the BASE model had been selected, one of the NoSQL databases would have been the choice. Choosing ACID Model left MS SQL Server and PostgreSQL databases as options even though all the options had their advantages and disadvantages for this implementation. Since between these two options there were not any mutually exclusive features and both could have been suitable choice, the choice was made by comparative analysis, i.e. benchmarking.

At first, test databases including fake data for three years were generated. Both databases ran on virtual computers in the same platforms (VMWare clusters) and table structure was used to make all the data included in one table as seen in figure 2.



⊟	sw_branches
PK	branch_ID
	sw_branch
	release_version
	date
	data_value
	number
	item

FIGURE 2. Table structure of the test databases

After the test databases were made, Python scripts were made for both databases. The scripts measured the execution time of establishing connections to the databases making queries and closing the connections (The scripts can be seen in figures 24 and 25 from appendix 1). The queries were select statements and average value calculations from tables contained more than four thousand rows of dummy data. They returned all data in ascending date order for defined release for 1, 2 and 3 years of dummy data. As seen from table 1 below, a clear difference was already observed between execution times. The execution times in MS SQL Server were only a couple of hundreds of milliseconds while in PostgreSQL they were over two seconds.

TABLE 1: Execution times with different queries for both databases with table style one

Query	Post-greSQL	MS SQL Server
SELECT * FROM testidb WHERE date >= '2021-11-24' ORDER BY date ASC	2.15s	0.076s
SELECT * FROM testidb WHERE date >= '2020-11-24' ORDER BY date ASC	2.25s	0.21s
SELECT * FROM testidb WHERE date >= '2019-11-24' ORDER BY date ASC	2.38s	0.33s
SELECT * FROM testidb WHERE (date >= '2021-11-24') AND (release_version='version_x') ORDER BY date ASC	2.11s	0.078s
SELECT * FROM testidb WHERE (date >= '2020-11-24') AND (release_version='version_x') ORDER BY date ASC	2.15s	0.09s
SELECT * FROM testidb WHERE (date >= '2019-11-24') AND (release_version='version_x') ORDER BY date ASC	2.19s	0.16s
SELECT * FROM testidb WHERE (date >= '2019-11-24') AND (sw_branch='branch_x') ORDER BY date ASC	2.18s	0.14s
SELECT AVG(data_value) AS avg FROM testidb WHERE (date >= '2021-11-24')	2.09s	0.065s
SELECT AVG(data_value) AS avg FROM testidb WHERE (date >= '2020-11-24')	2.10s	0.052s
SELECT AVG(data_value) AS avg FROM testidb WHERE (date >= '2019-11-24')	2.12s	0.056s
SELECT AVG(data_value) AS avg FROM testidb WHERE (date >= '2021-11-24') AND (sw_branch='branch_x')	2.09s	0.053s
SELECT AVG(data_value) AS avg FROM testidb WHERE (date >= '2020-11-24') AND (sw_branch='branch_x')	2.11s	0.056s
SELECT AVG(data_value) AS avg FROM testidb WHERE (date >= '2019-11-24') AND (sw_branch='branch_x')	2.12s	0.066s

In addition to these comparisons with Python, PostgreSQL's performance was evaluated with pgbench-program (See figures 26-29 from appendix 2). It runs benchmark tests on PostgreSQL such as the same sequence of SQL commands in multiple concurrent database sessions and calculates the average transaction rate (36). With a default run, five SELECT, UPDATE, and INSERT commands per transaction were executed resulting in 1012.145 transactions per second (tps) without initial connection time (Figure 26). With 10 clients, 2 threads, 10000 number of transactions per client, tps without initial connection time was 1446.76 (Figure 27). When selecting all data from a time period of three years in date order with ten clients, two threads and 10000 number of transactions, tps was 258.35 (Figure 28). With selection using same parameters but with defined release version, tps was 743.63 (Figure 29). When calculating average value from the same data with same parameter settings, tps was 1722.35 (Figure 30). These tps values describe tps capacity of the PostgreSQL server meaning that this database is also quite efficient. MS SQL Server does not have a similar tps benchmarking method so tps' cannot be compared together.

However, in addition to Python executing tests, elapsed time measurements were executed for MS SQL Server in its UI, MS SQL Server Management Studio (See figures 31-35 from appendix 3). The measurements showed that with the same queries as in other tests, the elapsed time for execution was only a couple of milliseconds depending on the query. For example, the execution time for three years' worth of data was 9 ms (Figure 31), for two years' data it was 6 ms (Figure 32) and for one year's data it was 1 ms (Figure 33). Also, when selecting defined release version's data from three years ago in date order, the execution time was 4 ms (Figure 34). When the average value was calculated, the execution time was only 1 ms (Figure 35). This shows the power of MS SQL Server, at least if it is executed in Management Studio. These results indicated rapid and efficient transactions even with a larger amount of data.

These were the results of a brief benchmarking. Even though they are not directly proportional, it shows that MS SQL Server is faster, at least with the tools used in comparison. Keeping the purpose of this work in mind, MS SQL Server offers an easy-to-use software application (Management Studio) that facilitates database usability. For example, wide knowledge of T-SQL programming language is not necessary because Management Studio provides quick and simple bases for queries. The idea of database handling is also to keep it as simple as possible. The company also recommended MS SQL Server, if possible, due to its familiarity and fault tolerance, so MS SQL Server was chosen as the database.

4 DATABASE IMPLEMENTATION

The first steps of the implementation were to design database table structures as well as a database handling code. First, different table structures, their linking, and their data types had to be examined and designed. The structure of the code had to be designed to be as generic and efficient as possible and include all possible transactions that may be needed with the database. After the design was made, the database could be created and implemented.

4.1 Table structure

For design of the table structure in database, three different styles were considered. The 1st style was a single table containing all the data for all software branches and their respective data which is the same structure as used in benchmarking (See the figure 2).

The 2nd style was a parent table containing child tables for each software branch, where the data was stored. It is described in figure 3 below. Both tables had primary keys (PK) that are unique identifiers for each row in a table. In relational databases when tables are linked together, the child table has a foreign key (FK) that refers to the parent table's primary key. A reference between data is called referential integrity.

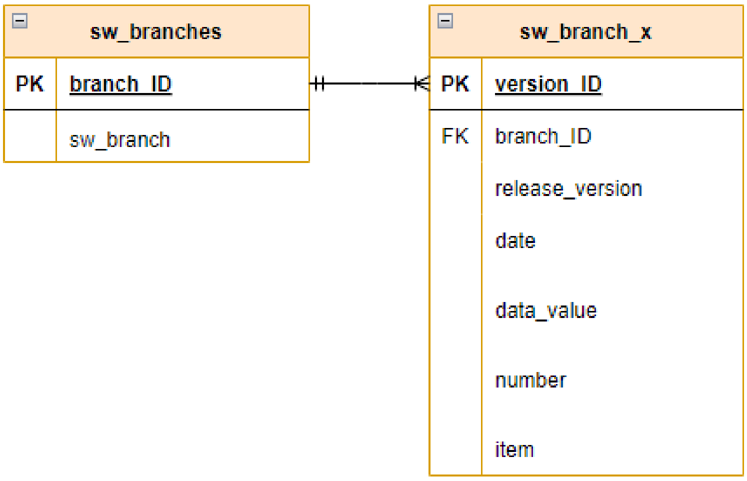


FIGURE 3. Table style two

The 3rd style was to make more distributed tables. The style had the parent table where each software branch had their own child tables. The child tables contained release versions, each of which had its own child tables where the data was stored. The 3rd style is described in figure 4.

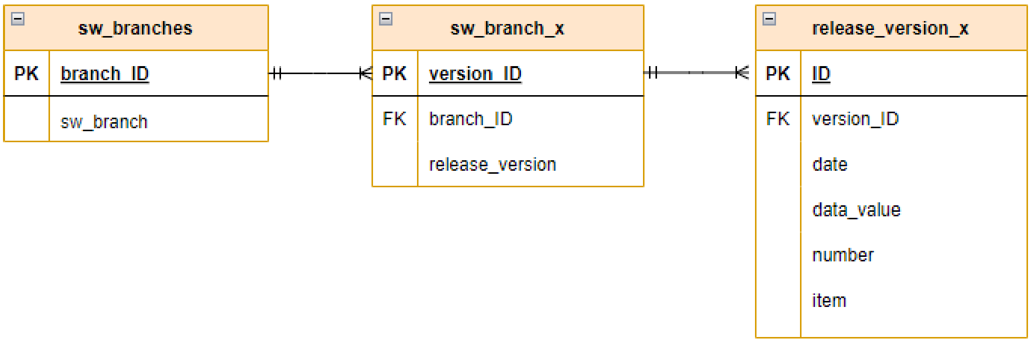


FIGURE 4. Table style three

Since the database is relational, the tables can be modified afterwards. For example adding or deleting columns and rows is possible making the database more flexible for the future. If the database is in use for several years, it probably needs to be modified at some point. There might also be a need to add different use cases to the database, in which case it may be desirable to link their tables together (Figure 5).

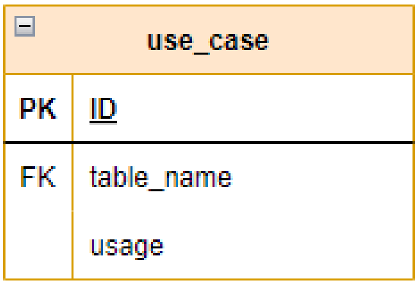


FIGURE 5. All possible tables in database

For readability, maintenance, and simple and continuous monitoring, style two was chosen for table structure because querying is a little faster. When compares to style one, the difference is not significant (See figures 36-39 from the appendix 4). Since new releases are constantly being developed, it is good that they are separated into their own tables. In the future, the data from some early releases might not be needed to be visible. With style three, the structure might have been too complicated. Although MS SQL Server supports complex structures, it was wanted to keep simple and clear.

As mentioned in chapter 2.1, MS SQL Server supports primitive data types, so the tables were initially designed to support these. If necessary, they can be modified afterwards, and columns can be added and removed from the existing tables.

4.2 Code structure

Using Pyodbc module a new class called database class was created using Python programming language and for using MS SQL Server. Pyodbc module is an open-source module and implements the DB API 2.0 specification (37). The class was designed to support basic SQL queries such as insert, select, update, alter and delete. Functionality of the class code was divided into six different sections which is illustrated in the following flowcharts (Figures 6-11). Figure 6 below describes what happens when the class is created. Due to the selected table structure, the existences of the tables are checked. If they do not exist, they are created, and their records are added. If they cannot be created due to, for example, invalid table columns' datatypes, errors are raised, and the connection is closed if it has been created.

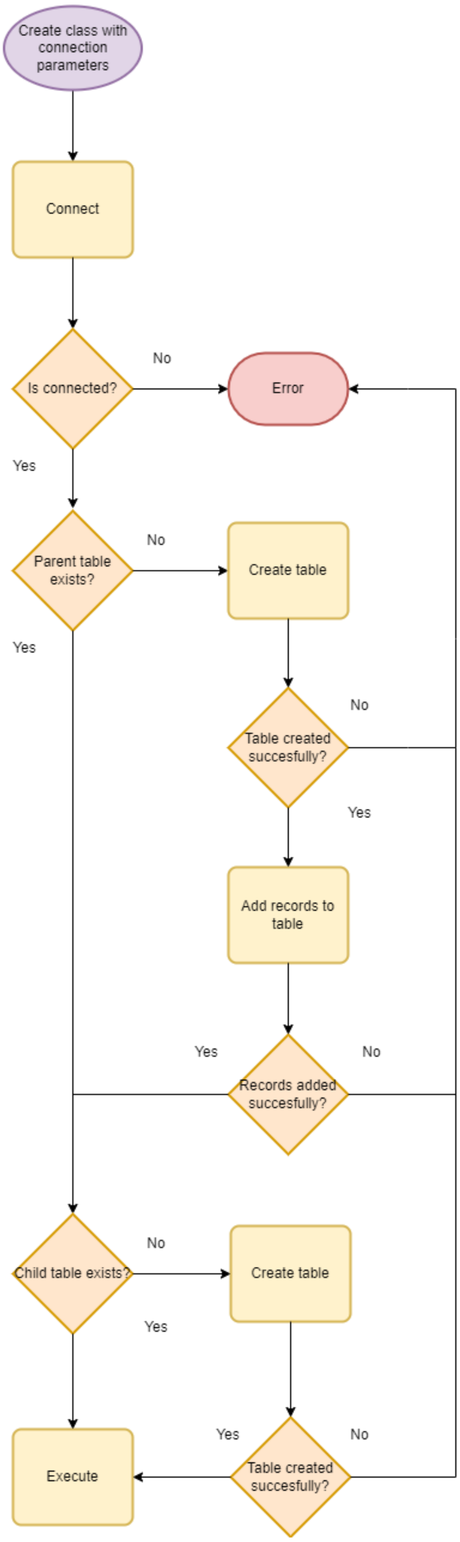


FIGURE 6. Flowchart of connecting the database

Figure 7 below shows the structure of the insert method which inserts data to the database if the parameters given are correct; otherwise, an error is raised. The parameters are the table name and the values that will be inserted. The values must be in the same order as the columns in the table because the method assumes that values are added to all the columns in the table. Errors can be caused by for instance invalid data types or incorrect definitions of table columns.

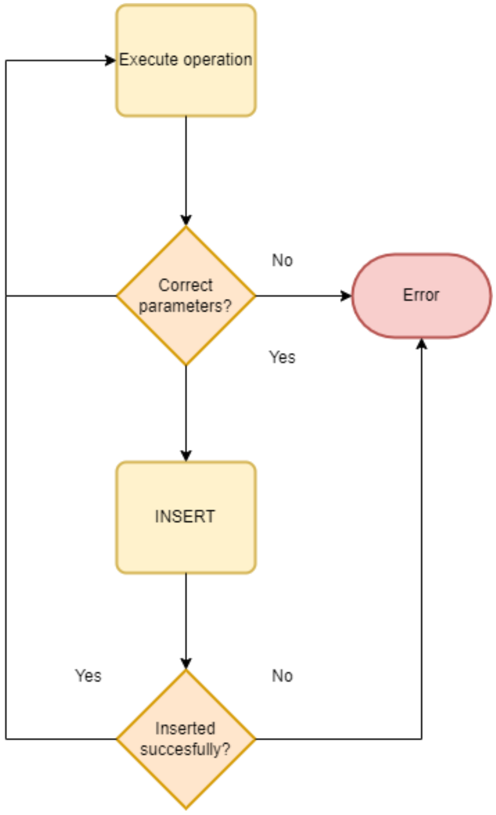


FIGURE 7. Flowchart of the insert method

Figure 8 below describes the selection method. It returns selected data from the database if the parameters are correct; otherwise, it raises an error. Columns and table where the data is returned from, must be defined but the conditions for it are optional. The conditions can be, for example, returning order of the data or definitions for returned values.

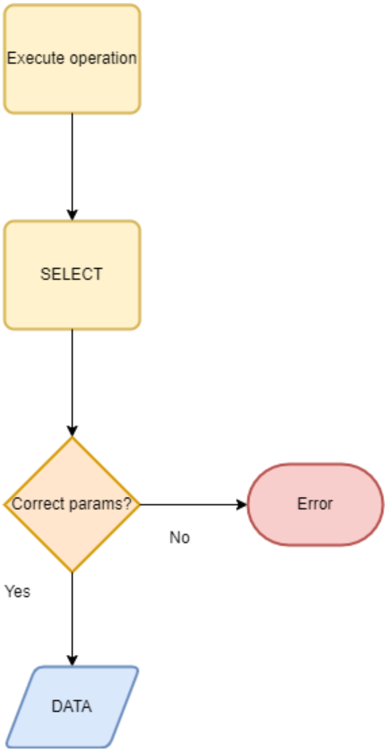


FIGURE 8. Flowchart of the select method

The update method is described in figure 9. It is used when a row in an existing table needs to be modified. For its parameters the method receives the table name, name of the column whose value will be updated, new replaceable value, and definition of the row to be modified. The definition of the row can be its id number for instance. If the parameters are not correct, an error is raised, and execution stops, otherwise the row is updated.

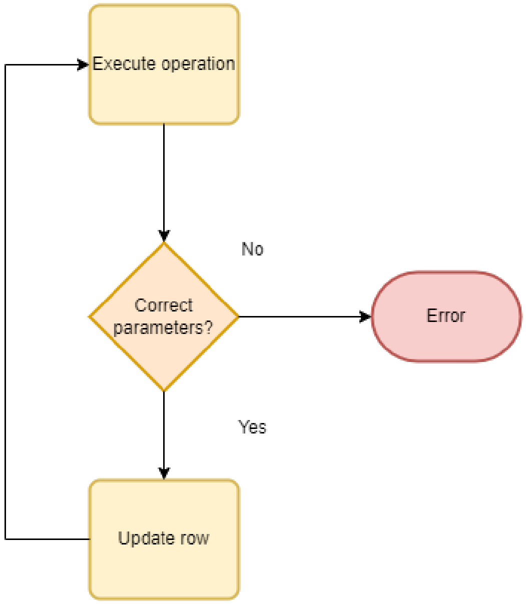


FIGURE 9. Flowchart of the update method

The alter method described in figure 10 below is used when columns from existing table need to be added, deleted, or modified. It is also used when different constraints need to be added or dropped. The user can define what needs to be altered in the table by giving it a statement argument. If table parameters or statement are incorrect, an error is raised. For example, if a new column is added, it must support NULL-values as they are added in the each existing row.

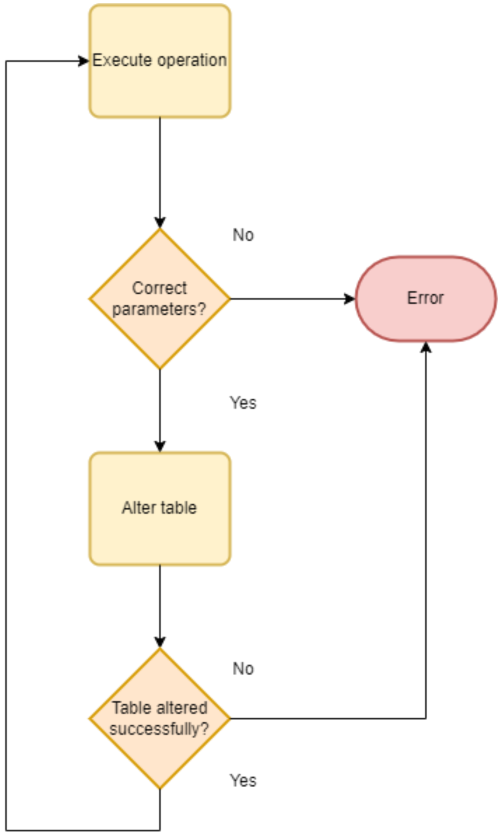


FIGURE 10. Flowchart of the alter method

The delete method described in figure 11 below is used for deleting records from an existing table. The user can decide whether to give optional parameters or not. If none are defined, all the records from the table will be deleted; otherwise, specified records will be deleted. The user decides how those records are selected, for example, by the id number of a row or with defined values of the row.

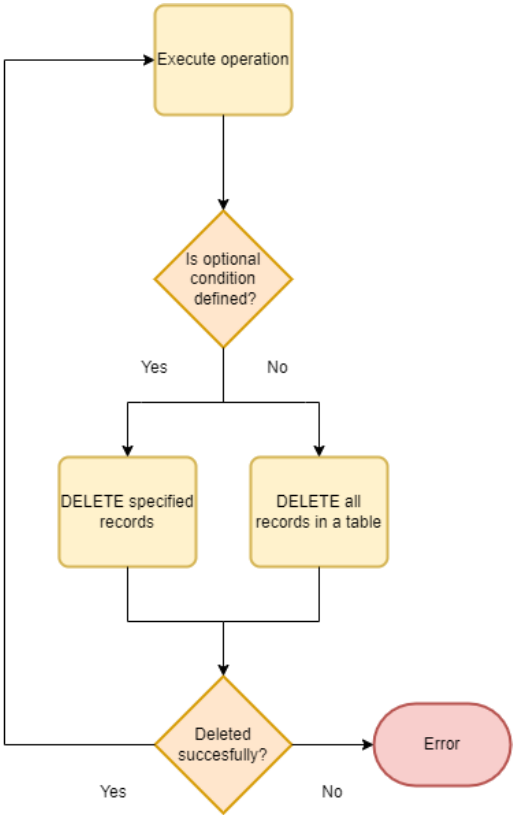


FIGURE 11. Flowchart of the delete method

4.3 Backend implementation

The database was created to the company server by their IT support. Connection to it was done with an earlier mentioned Pyodbc module which provides two objects: Connection and Cursor. Connection object manages connections to the database and executions to the database are made with Cursor object that represents a database cursor. Database connection and object creations can be seen in figure 12 below.

```
self.connection = pyodbc.connect("driver={%s};server=%s;database=%s;uid=
                %s;pwd=%s" % (driver, server, db, user, password))

self.cursor = self.connection.cursor()
```

Figure 12. Database connection and object creations

The database class was written using Python programming language and the class included the methods described in the previous chapter. In addition, the class had private get methods which made sure the connection was established, and they were used in every method. The class had also disconnect method that closed the database connection if it was created. The class required at least the database parameters in order to use its methods. Those parameters are driver of the database, server's name, database's name, username, and password. An example code of the disconnect method and get methods are shown in figure 13 below.

```
def __get_connection(self):
    """Returns self.connection if exists"""

    if self.connection:
        return self.connection
    else:
        raise Exception("Database is not connected!")

def __get_cursor(self):
    """Returns self.cursor if exists"""

    if self.cursor:
        return self.cursor
    else:
        raise Exception("Database is not connected!")

def disconnect(self):
    """Disconnects database connection if exists"""

    if self.connection:
        self.cursor.close()
        self.connection.close()
        print("  Connection is closed")
        self.cursor = None
        self.connection = None
```

Figure 13. Example code of the disconnect method and get methods

Database tables were created in the database class using the table structure style two with necessary relations and primitive data types such as string, date, float, and integer. They could have been created in Management Studio as well. MS SQL Server has support for stored procedures that are prepared SQL code and can be called for executing. However, in this implementation they were not used since database handling could easily be done with one-line queries.

4.4 Application implementation

A demo application was implemented to demonstrate the usage of the database class. The application utilized class methods and parameters of the database could be given as an input with Python's argparse-module. However, the parameters were defined as default values in the application. Argparse-module is used when arguments and sub-commands are given from the command-line (38). The user could also define the usage of the application with arguments that are updating rows to an existing table, deleting rows from one, and altering tables. By default, the database was updated. Table 2 below shows the executing commands.

TABLE 2. Different use cases for demo application

Execution code with args	Use case
python demo_app.py -r	Updates row to an existing table
python demo_app.py -d	Deletes defined row from table
python demo_app.py -a	Adds or deletes column from table
python demo_app.py -u	Updates the database (DEFAULT)

Since the implementation was a demonstration, dummy data was created to the database. It consisted of dates from the last two years, random data values generated by Python's random module, increased values for the number column and NULL values for the item column.

In the beginning of any execution, the application created the database class, checked table existences and did the necessary table creations as described in the figure 5. Without additional arguments, the application updated the database but if one was used, the database was not updated.

To follow the table structure defined at the beginning of the database update, the value of the branch id was selected from the parent table since the branch id was the foreign key of the table. The selecting can be seen in figure 14 below.

```
def update_db(self, branch, table):
    # Get branch_ID
    rows = self.DB.select(table_name=('dbo.' + self.parent_table),
                          columns='branch_ID', conditions =
                          "WHERE sw_branch ='{}'".format(table))

    for row in rows:
        branch_ID = row.branch_ID
```

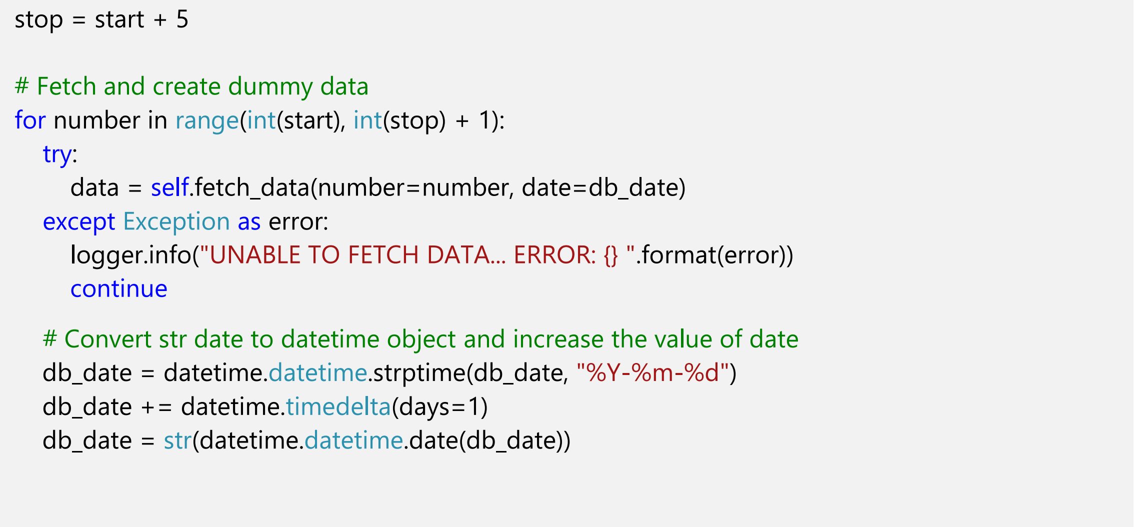
Figure 14. Defining branch id

After that, all values from database's number column and all dates from the database's table were returned. If there were not any records, the value of the number was given as one and the date was set to the current day. Otherwise, both were given their most recent values as shown in figure 15 below.

```
for version in versions:
    db_numbers = []
    db_dates = []

    # Get numbers and dates from db in date order
    rows = self.DB.select(table_name=('dbo.' + table), columns=('date,
                          number'), conditions= "WHERE release_version =
                          '{}' ORDER BY date ".format(version))

    db_numbers = [row.number for row in rows]
    db_dates = [row.date for row in rows]

    # If no records in table
    if not db_numbers:
        start = 1
        db_date = str(datetime.date.today())
    else:
        start = db_numbers[-1]
        db_date = db_dates[-1]
```

Figure 15. Setting values for number and date variables

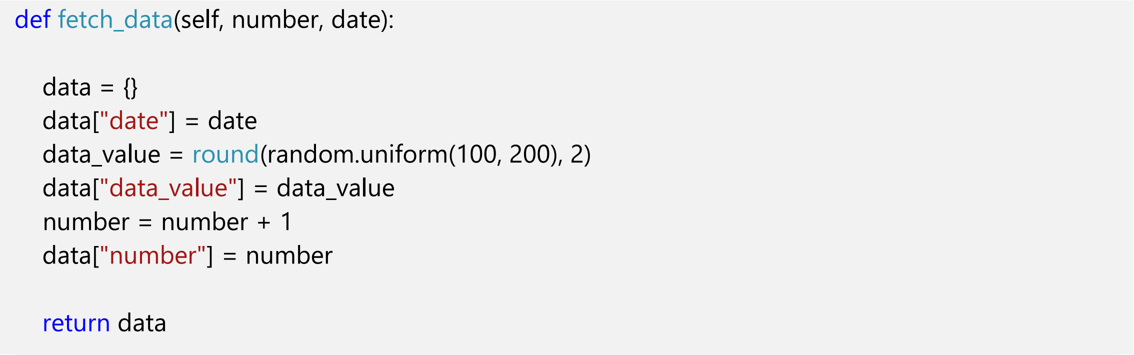
From defined value of the number column to the next five values were looped and data for the database was fetched. Also, dates were increased with Python's datetime module as depicted in figure 16.



```
stop = start + 5

# Fetch and create dummy data
for number in range(int(start), int(stop) + 1):
    try:
        data = self.fetch_data(number=number, date=db_date)
    except Exception as error:
        logger.info("UNABLE TO FETCH DATA... ERROR: {} ".format(error))
        continue

    # Convert str date to datetime object and increase the value of date
    db_date = datetime.datetime.strptime(db_date, "%Y-%m-%d")
    db_date += datetime.timedelta(days=1)
    db_date = str(datetime.datetime.date(db_date))
```

Figure 16. A for loop for fetching data

The fetched data was created from dummy data, too. It had number from the loop and date was the latest date of the database, the current date, or the increased value by the datetime module. The data value was also generated with random module as figure 17 shows.



```
def fetch_data(self, number, date):

    data = {}
    data["date"] = date
    data_value = round(random.uniform(100, 200), 2)
    data["data_value"] = data_value
    number = number + 1
    data["number"] = number

    return data
```

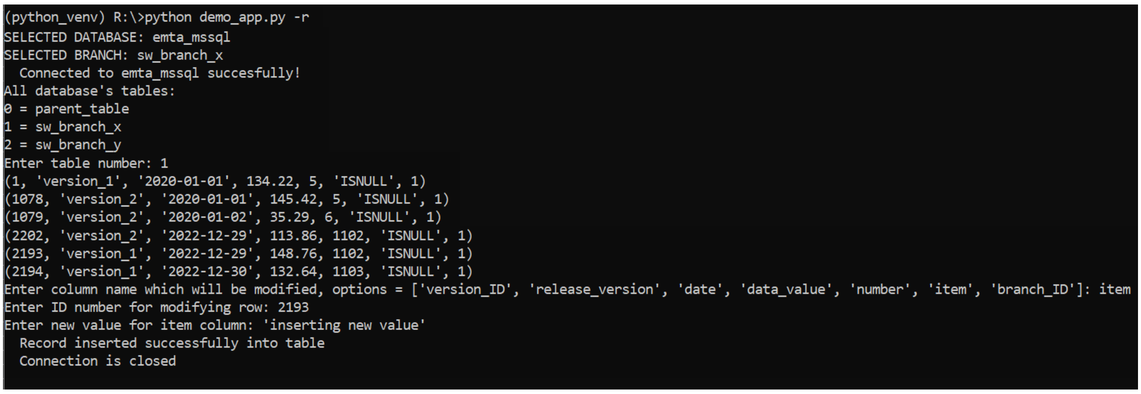
Figure 17. Fetching data function

After the data was fetched, the application checked whether it existed in the database since the insert method does not take a position on that. The existence was figured out by selecting the row with the data. If no records were returned, it meant that the data was not in the database so it was inserted. Whether the data was inserted or not the loop continued to the next round until the end of the loop. This is shown in figure 18 below.

```
date = data.get("date")
data_value = data.get("data_value")
number = data.get("number")
item = 'ISNULL'

data_to_db = (version, date, data_value, number, item, branch_ID)

# Insert data to table if does not exist
conds = "WHERE release_version='{}' AND date='{}' AND data_value={} AND
        number={} AND branch_ID={}".format(version, date, data_value,
        number, branch_ID)

rows = self.DB.select(table_name=('dbo.' + table), columns=columns,
                    conditions=conds)
if not rows:
  self.DB.insert(table_name=table, data=data_to_db)
```

Figure 18. Checking data existence and possible inserting

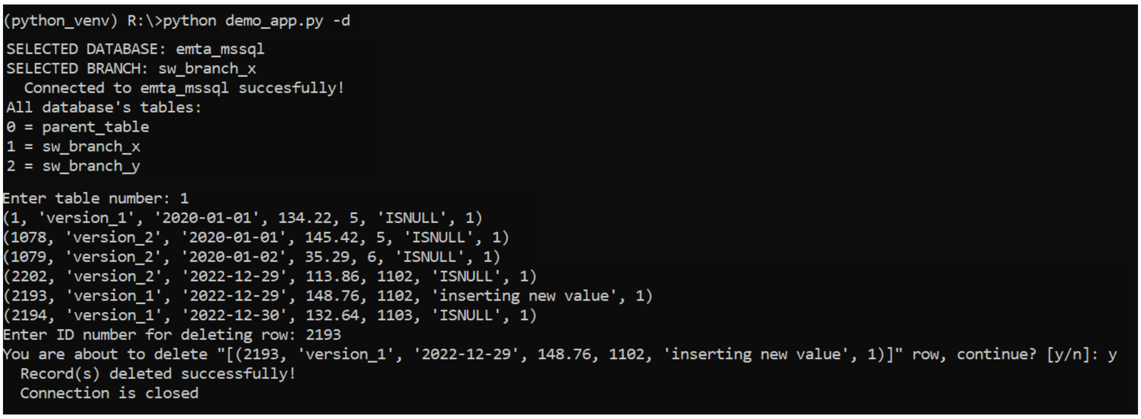
As mentioned earlier, the application executions were done from the command-line. If a row was wanted to be updated to an existing table, the application displayed all tables from the database. The user could select the one to be modified after proper class and table creations. After the selection was taken as an input, the application displayed all data in date order from the table. Based on the data, the user could define which column value was about to be modified as the application displayed columns of the table as well. The correct row was defined by its id number and new value to be replaced was entered to the application by the user. A successful print appeared to the command-line if no errors occurred, all parameters were correct, and the transaction was completed. The execution of this function is shown in figure 19 below.



```
(python_venv) R:\>python demo_app.py -r
SELECTED DATABASE: emta_mssql
SELECTED BRANCH: sw_branch_x
  Connected to emta_mssql succesfully!
All database's tables:
0 = parent_table
1 = sw_branch_x
2 = sw_branch_y
Enter table number: 1
(1, 'version_1', '2020-01-01', 134.22, 5, 'ISNULL', 1)
(1078, 'version_2', '2020-01-01', 145.42, 5, 'ISNULL', 1)
(1079, 'version_2', '2020-01-02', 35.29, 6, 'ISNULL', 1)
(2202, 'version_2', '2022-12-29', 113.86, 1102, 'ISNULL', 1)
(2193, 'version_1', '2022-12-29', 148.76, 1102, 'ISNULL', 1)
(2194, 'version_1', '2022-12-30', 132.64, 1103, 'ISNULL', 1)
Enter column name which will be modified, options = ['version_ID', 'release_version', 'date', 'data_value', 'number', 'item', 'branch_ID']: item
Enter ID number for modifying row: 2193
Enter new value for item column: 'inserting new value'
  Record inserted successfully into table
  Connection is closed
```

Figure 19. Execution of updating row to an existing table

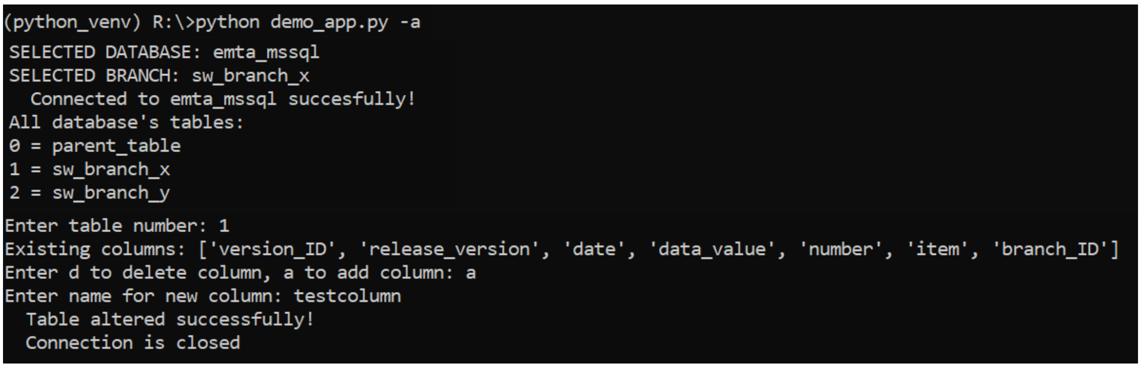
One of the use cases of the application was to delete a row from table. It had similar functionality as updating the row method. At first, all the tables from the database were shown to the user to select the one that would be affected. After the selection, data of the table was displayed in date order from which the user could select the id number of the row that needed to be deleted. Before deleting, the application ensured confirmation from the user about the row's deletion and if the user confirmed it, the row was deleted. Without confirmation, the execution was aborted. Figure 20 shows the user view of this method.



```
(python_venv) R:\>python demo_app.py -d
 SELECTED DATABASE: emta_mssql
 SELECTED BRANCH: sw_branch_x
   Connected to emta_mssql succesfully!
 All database's tables:
 0 = parent_table
 1 = sw_branch_x
 2 = sw_branch_y

Enter table number: 1
(1, 'version_1', '2020-01-01', 134.22, 5, 'ISNULL', 1)
(1078, 'version_2', '2020-01-01', 145.42, 5, 'ISNULL', 1)
(1079, 'version_2', '2020-01-02', 35.29, 6, 'ISNULL', 1)
(2202, 'version_2', '2022-12-29', 113.86, 1102, 'ISNULL', 1)
(2193, 'version_1', '2022-12-29', 148.76, 1102, 'inserting new value', 1)
(2194, 'version_1', '2022-12-30', 132.64, 1103, 'ISNULL', 1)
Enter ID number for deleting row: 2193
You are about to delete "[(2193, 'version_1', '2022-12-29', 148.76, 1102, 'inserting new value', 1)]" row, continue? [y/n]: y
  Record(s) deleted successfully!
  Connection is closed
```

Figure 20. Deleting row from table

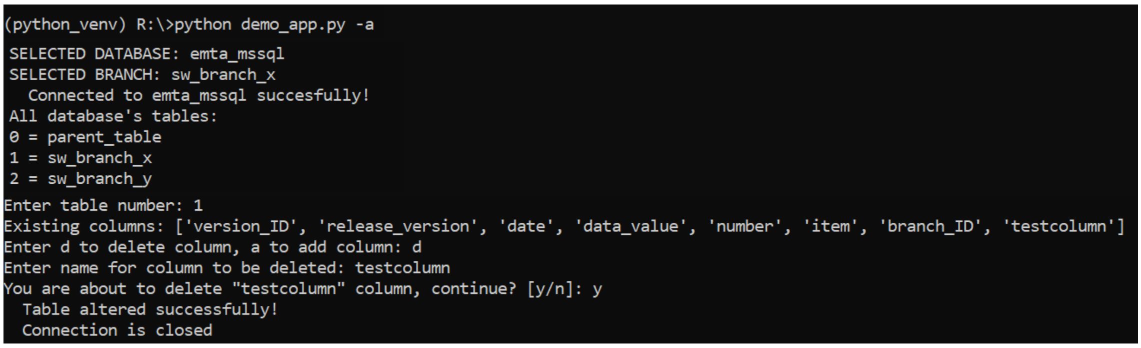
The application offered two different methods for altering tables. Columns could be added or deleted from tables. At first, tables were shown, and the user had to select one. The columns of the selected table were shown, and the user had to choose whether to add or delete a column. If a column was wanted to be added, the user entered his or her name. If no errors occurred, it was altered to the table successfully, as seen in figure 21. In this implementation, new columns' datatype were strings with NULL value support but they can be defined as wished when using the database class.



```
(python_venv) R:\>python demo_app.py -a
 SELECTED DATABASE: emta_mssql
 SELECTED BRANCH: sw_branch_x
   Connected to emta_mssql succesfully!
 All database's tables:
 0 = parent_table
 1 = sw_branch_x
 2 = sw_branch_y
Enter table number: 1
Existing columns: ['version_ID', 'release_version', 'date', 'data_value', 'number', 'item', 'branch_ID']
Enter d to delete column, a to add column: a
Enter name for new column: testcolumn
  Table altered successfully!
  Connection is closed
```

Figure 21. Adding a column to table

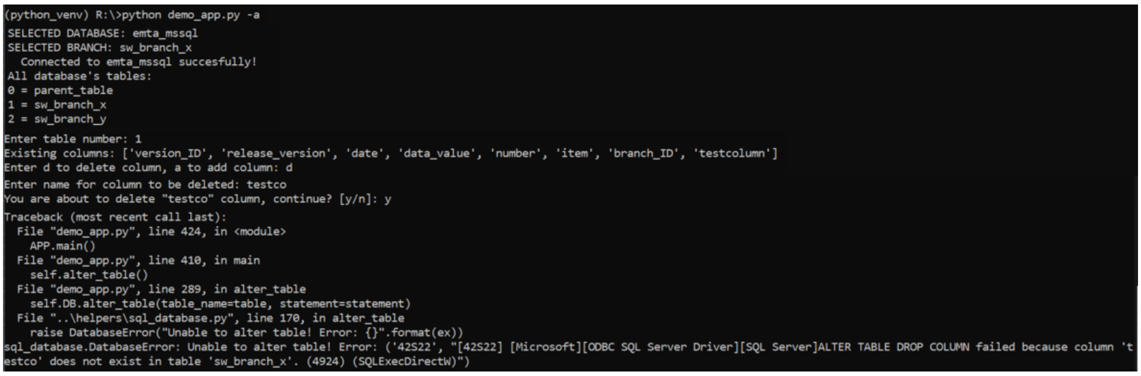
If a column was wanted to be deleted, after the choice of use was made, the application asked for the name of that column and confirmation from the user. If no errors occurred, the column was deleted successfully, as seen in figure 22.



```
(python_venv) R:\>python demo_app.py -a
 SELECTED DATABASE: emta_mssql
 SELECTED BRANCH: sw_branch_x
   Connected to emta_mssql succesfully!
 All database's tables:
 0 = parent_table
 1 = sw_branch_x
 2 = sw_branch_y
Enter table number: 1
Existing columns: ['version_ID', 'release_version', 'date', 'data_value', 'number', 'item', 'branch_ID', 'testcolumn']
Enter d to delete column, a to add column: d
Enter name for column to be deleted: testcolumn
You are about to delete "testcolumn" column, continue? [y/n]: y
  Table altered successfully!
  Connection is closed
```

Figure 22. Deleting a column from table

Figure 23 below shows an example when non existing column is tried to be deleted from the table and an error is raised.



```
(python_venv) R:\>python demo_app.py -a
 SELECTED DATABASE: emta_mssql
 SELECTED BRANCH: sw_branch_x
   Connected to emta_mssql succesfully!
 All database's tables:
 0 = parent_table
 1 = sw_branch_x
 2 = sw_branch_y
Enter table number: 1
Existing columns: ['version_ID', 'release_version', 'date', 'data_value', 'number', 'item', 'branch_ID', 'testcolumn']
Enter d to delete column, a to add column: d
Enter name for column to be deleted: testco
You are about to delete "testco" column, continue? [y/n]: y
Traceback (most recent call last):
  File "demo_app.py", line 424, in <module>
    APP.main()
  File "demo_app.py", line 410, in main
    self.alter_table()
  File "demo_app.py", line 289, in alter_table
    self.DB.alter_table(table_name=table, statement=statement)
  File "..\helpers\sql_database.py", line 170, in alter_table
    raise DatabaseError("Unable to alter table! Error: {}".format(ex))
sql_database.DatabaseError: Unable to alter table! Error: ('42S22', "[42S22] [Microsoft][ODBC SQL Server Driver][SQL Server]ALTER TABLE DROP COLUMN failed because column 't
estco' does not exist in table 'sw_branch_x'. (4924) (SQLExecDirectW)")
```

Figure 23. Deleting an invalid column from table

Backup methods for the demo application were not created but when the database is used for other applications, backup will be handled by the IT support of the company.

5 FURTHER DEVELOPMENT

Further development for the database and the database class was already considered during the selection of the database and its design. Data growth and expansion of the database were taken into consideration by selecting and designing the table structure so that it would not affect negatively the database usage in the future. The code of the database class was made to be generic in order to be utilized with other applications if wanted. In the future, an application could be created allowing the database to be used with other SQL-based databases as well and not to be dependent on MS SQL Server.

In the future, a web UI could be implemented to the database. It would provide the class methods to the user, so the user would not need to know anything about the code or the database. Web UI would take care of the database handling and provide the user with a web page where all the functionality and the requirements such as graphical data presentation and adding own notes are supported. It would provide a great compilation of the database and the database class making it easy for anyone to use.

An existing ready-made tool, for example earlier mentioned Grafana, could be used for data visualization. It creates fast and flexible charts, it has dynamic dashboards, and the user can define alert rules for the most important data bringing more usability. It also provides wide graphical use that might be needed in different use cases. In addition, Grafana could be embedded to the web UI.

Overall, this implementation has great potential to develop and improve in the future.

6 CONCLUSION

The purpose of this thesis was to improve the company's software quality and performance metrics tracking applications because a proper database for storing and monitoring data from the applications was needed. Selection of the database that met the company's need the best was made by comparing five different databases. From these five, due to model choice of the database, two were left for benchmarking. Based on the benchmarking, MS SQL Server was selected due to its pivotal features, reliability, and speed.

The implementation met the database requirements mentioned in chapter 2.1. For example, MS SQL Server supports large amount of data, adding own notes to the database is possible, and it has an UI. MS SQL Server implementation was effortless after the tools and T-SQL became familiar. With its UI, outlining of the database was easy and it helped in the implementation. Implementation of Pyodbc module was simple and did not cause any bigger issues. The designed goals were achieved, and the implementation worked as expected. Nothing unexpected happened so, all in all, this work was uncomplicated making the work time-efficient and enabled full focus on it. With these outcomes, this work improved the company's software quality and performance metrics tracking applications by making them faster, more secure, more generic, and more scalable corresponding to the purpose of this work.

The thesis work was split into weekly tasks to ensure its progress. First, the work was scheduled, and its structure was designed into a plan. After the plan was completed, research of the databases was done. When enough information on the databases was gathered, benchmarking and selection work started. Implementation work after the selection included design and creation of the database as well as the implementation of the demo application. The final part of the work was to finish the remaining documentation work. With the plan, working was consistent and straightforward, which ended up in a successful result.

By completing this work I gained a lot of new knowledge and developed my academic writing skills as well as my coding skills that were my personal aims of this work. With this work, I was able to do a bigger project by myself, which was very interesting and educational and proved to me the importance of every step in long-term projects. This work was related to my previous school projects in the company thus making the last courses of my studies a great compilation.

REFERENCES

1. About Nordic Semiconductor. Who we are. Date of retrieval 09.11.2022. https://www.nordicsemi.com/about-us.

2. DB-Engines Ranking. Date of retrieval 23.01.2023. https://db-engines.com/en/ranking.

3. SQL Server 2022. Date of retrieval 23.01.2023. https://www.microsoft.com/en-us/sql-server/sql-server-2022.

4. Introduction to SQL. What is SQL? Date of retrieval 10.11.2022. https://www.w3schools.com/sql/sql_intro.asp.

5. What is NoSQL? Date of retrieval 15.11.2022. https://www.mongodb.com/nosql-explained.

6. Python. Date of retrieval 23.01.2023. https://www.python.org/.

7. Getting Started with SQL Server. What is SQL Server. Date of retrieval 14.11.2022. https://www.sqlservertutorial.net/getting-started/what-is-sql-server/.

8. SQL Server Basics. SQL Server Data Types. Date of retrieval 14.11.2022. https://www.sqlservertutorial.net/sql-server-basics/sql-server-data-types/.

9. Martyna Sławińska 2021. MS SQL Server vs. PostgreSQL: Which Should You Choose for a New Project? Date of retrieval 18.11.2022. https://learnsql.com/blog/ms-sql-vs-postgresql/.

10. SQL Server Tutorial. SQL Server Simple Recovery Model. Date of retrieval 21.12.2022. https://www.mssqltips.com/sqlservertutorial/4/sql-server-simple-recovery-model/.

11. Relational Databases. Backup & restore. Recovery Models (SQL Server). Date of retrieval 21.12.2022. https://learn.microsoft.com/en-us/sql/relational-databases/backup-restore/recovery-models-sql-server?view=sql-server-ver16.

12. Salman Ravoof 2022. PostgreSQL vs SQL Server: 16 Critical Differences. Date of retrieval 17.11.2022. https://kinsta.com/blog/postgresql-vs-sql-server/.

13. Relational Databases. Manage. Change the Configuration Settings for a Database. Date of retrieval 21.12.2022. https://learn.microsoft.com/en-us/sql/relational-databases/databases/change-the-configuration-settings-for-a-database?view=sql-server-ver16.

14. PostgreSQL. About. Date of retrieval 17.11.2022. https://www.postgresql.org/about/.

15. Grafana. Overview. Date of retrieval 15.11.2022. https://grafana.com/grafana/.

16. MongoDB Manual. What is MongoDB? Date of retrieval 14.11.2022. https://www.mongodb.com/docs/manual/.

17. MongoDB Manual. Introduction to MongoDB. Date of retrieval 14.11.2022. https://www.mongodb.com/docs/manual/introduction/.

18. Petroc Taylor 2022. Most popular database management systems worldwide 2022. Date of retrieval 21.12.2022. https://www.statista.com/statistics/809750/worldwide-popularity-ranking-database-management-systems/#:~:text=As%20of%20August%202022%2C%20the,rounded%20out%20the%20top%20three..

19. XTIVIA 2019. The Pros and Cons of MongoDB. Date of retrieval 15.11.2022. https://virtual-dba.com/blog/pros-and-cons-of-mongodb/.

20. Debra Bruce. Understanding the Pros and Cons of MongoDB. Date of retrieval 16.11.2022. https://www.knowledgenile.com/blogs/pros-and-cons-of-mongodb/.

21. MongoDB Atlas. Date of retrieval 23.01.2023. https://www.mongodb.com/atlas.

22. MongoDB Cloud Services. Date of retrieval 23.01.2023. https://www.mongodb.com/cloud.

23. MongoDB Ops Manager. Ops Manager Overview. Date of retrieval 23.01.2023. https://www.mongodb.com/docs/ops-manager/current/application/.

24. MongoDB Manual. MongoDB Backup Methods. Date of retrieval 21.11.2022. https://www.mongodb.com/docs/manual/core/backups/.

25. MongoDB Documentation. MongoDB Compass. Overview. Date of retrieval 21.11.2022. https://www.mongodb.com/docs/compass/current/.

26. Cassandra Documentation. Overview. Date of retrieval 15.11.2022. https://cassandra.apache.org/doc/latest/cassandra/architecture/overview.html.

27. Cassandra Documentation. RDBMS Design. Date of retrieval 22.12.2022. https://cassandra.apache.org/doc/latest/cassandra/data_modeling/data_modeling_rdbms.html.

28. DBMS Tools. DbSchema. Date of retrieval 22.12.2022. https://dbmstools.com/tools/dbschema.

29. Python Data Persistence Tutorial. Python Data Persistence – Cassandra Driver. Date of retrieval 17.11.2022. https://www.tutorialspoint.com/python_data_persistence/python_data_persistence_cassandra_driver.htm.

30. InfluxDB. Overview. Date of retrieval 09.12.2022. https://www.influxdata.com/products/influxdb-overview/.

31. InfluxDB and Flux. What is Flux? Date of retrieval 16.11.2022. https://www.influxdata.com/products/flux/.

32. InfluxDB. Documentation. Python client library. Date of retrieval 18.11.2022. https://docs.influxdata.com/influxdb/v2.5/api-guide/client-libraries/python/.

33. InfluxDB. Documentation. Visualization types. Date of retrieval 18.11.2022. https://docs.influxdata.com/influxdb/v2.5/visualize-data/visualization-types/.

34. InfluxDB. Documentation. Back up data. Date of retrieval 21.11.2022. https://docs.influxdata.com/influxdb/v2.5/backup-restore/backup/.

35. ACID Model vs BASE Model For Database. Date of retrieval 18.11.2022. https://www.geeksforgeeks.org/acid-model-vs-base-model-for-database/.

36. Documentation. PostgreSQL Client Applications. Pgbench. Date of retrieval 25.11.2022. https://www.postgresql.org/docs/current/pgbench.html.

37. Pyodbc 4.0.35. Date of retrieval 29.11.2022.https://pypi.org/project/pyodbc/.

38. Python Documentation. Library. Argparse. Date of retrieval 14.12.2022. https://docs.python.org/3/library/argparse.html.

APPENDICES

Python script of PostgreSQL:

```
import psycopg2
from datetime import datetime
from dateutil.relativedelta import relativedelta
from config import config

def connect():
    """ Connect to the PostgreSQL database server """

    conn = None
    try:
        start = datetime.datetime.now()
        # Read defined connection parameters from config file
        params = config()

        # Connect to the PostgreSQL server
        print('Connecting to the PostgreSQL database...')
        conn = psycopg2.connect(**params)
        # Create a cursor
        cursor = conn.cursor()

        # Make queries
        select(cursor)

        # Close the communication with the PostgreSQL
        cursor.close()

    except (Exception, psycopg2.DatabaseError) as error:
        print(error)

    finally:
        if conn is not None:
            conn.close()
            print('Database connection closed.')
            took = datetime.datetime.now() - start
            print("\n[DEBUG] processing took: %s" % took)

def select(cursor):

    # Calculates the date based on given amount of months
    timediff = datetime.today() - relativedelta(months=int(12))
    timediff = str(datetime.date(timediff))

    # Different queries
    # select = "SELECT * FROM testidb WHERE (date >= {}) AND
    #  (release_version='version_x') ORDER BY date ASC".format(timediff)
    select = "SELECT * FROM testidb WHERE (date >= {}) AND
      (sw_branch='branch_x') ORDER BY date ASC".format(timediff)
    # Calculates average value
    # select = "SELECT AVG(data_value) AS avg FROM testidb WHERE
    #  (date >= {})".format(timediff)
    # select = "SELECT AVG(data_value) AS avg FROM testidb WHERE
    #  (date >= {}) AND (sw_branch='branch_x')".format(timediff)
   cursor.execute(select)

    while 1:
        row = cursor.fetchone()
        if not row:
            break
        # Print the records
        print(row)

if __name__ == '__main__':
    connect()
```

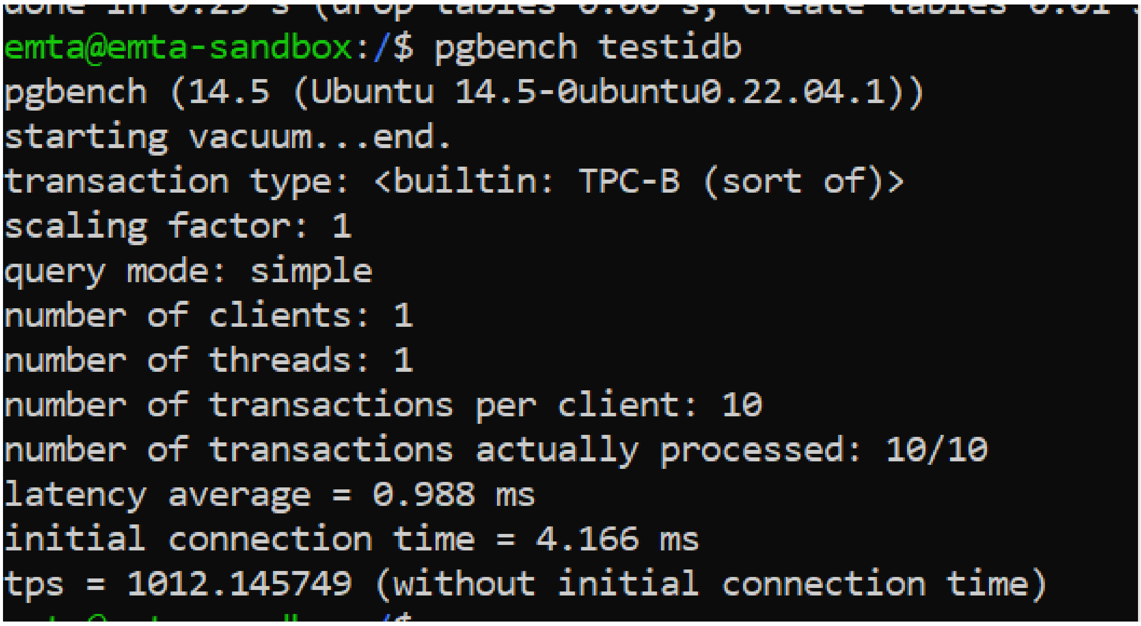
Figure 24. Execution time measurements script for PostgreSQL

Python script for MS SQL Server:

```
import pyodbc
from datetime import datetime
from dateutil.relativedelta import relativedelta

driver = '<driver>'
server = '<server_name>'
db = '<database name>'
user = '<user name>'
password = '<password>'


def connect():

    try:
        start = datetime.datetime.now()
        connection = pyodbc.connect("driver={%s};server=%s;database
                =%s;uid=%s;pwd=%s" % (driver, server, db, user, password))
        cursor = connection.cursor()
        select_data(cursor)

    except pyodbc.Error as error:
        print("Failed to connect and handle the db: {}".format(error))

    finally:
        if connection:
            cursor.close()
            connection.close()
            print("Connection is closed")
            took = datetime.datetime.now() - start
            print("\n[DEBUG] processing took: %s" % took)

def select_data(cursor):
    #  Calculates the date based on given amount of months
    timediff = datetime.today() - relativedelta(months=int(36))
    timediff = str(datetime.date(timediff))

    # Benchmarking queries
    # cursor.execute("SELECT * FROM testidb WHERE (date >= ?)
    #  ORDER BY date ASC", (timediff))
    # cursor.execute("SELECT * FROM testidb WHERE (date >= ?)
    #  AND (release_version='version_x') ORDER BY date ASC", (timediff))
    # cursor.execute("SELECT * FROM testidb WHERE (date >= ?)
    #  AND (sw_branch='branch_x') ORDER BY date ASC", (timediff))
    # cursor.execute("SELECT AVG(data_value) AS avg FROM testidb
    #  WHERE (date >= ?)", (timediff))
    cursor.execute("SELECT AVG(data_value) AS avg FROM testidb
     WHERE (date >= ?) AND (sw_branch='branch_x')", (timediff))

    while 1:
        row = cursor.fetchone()
        if not row:
            break
        print(row)


if __name__ == '__main__':
    connect()
```

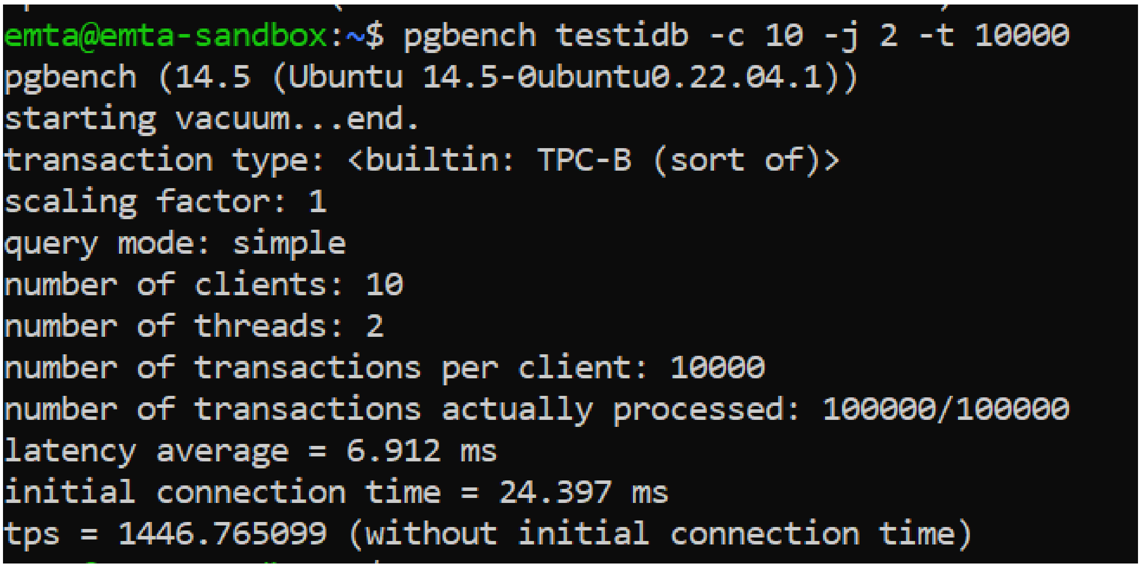
Figure 25. Execution time measurements script for MS SQL Server

Benchmark tests for PostgreSQL with pgbench-program.



```
emta@emta-sandbox:/$ pgbench testidb
pgbench (14.5 (Ubuntu 14.5-0ubuntu0.22.04.1))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
latency average = 0.988 ms
initial connection time = 4.166 ms
tps = 1012.145749 (without initial connection time)
```

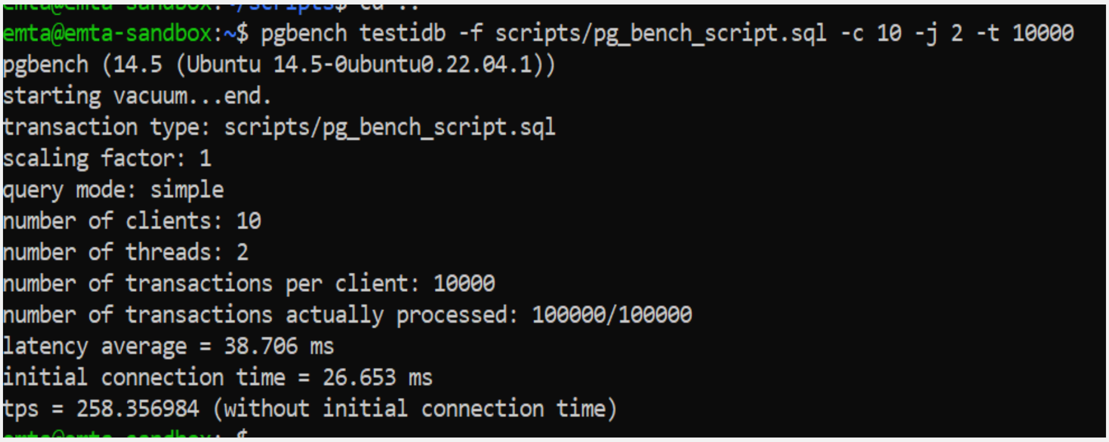
Figure 26. Default pgbench run



```
emta@emta-sandbox:~$ pgbench testidb -c 10 -j 2 -t 10000
pgbench (14.5 (Ubuntu 14.5-0ubuntu0.22.04.1))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 2
number of transactions per client: 10000
number of transactions actually processed: 100000/100000
latency average = 6.912 ms
initial connection time = 24.397 ms
tps = 1446.765099 (without initial connection time)
```

Figure 27. Default pgbench run with defined parameters

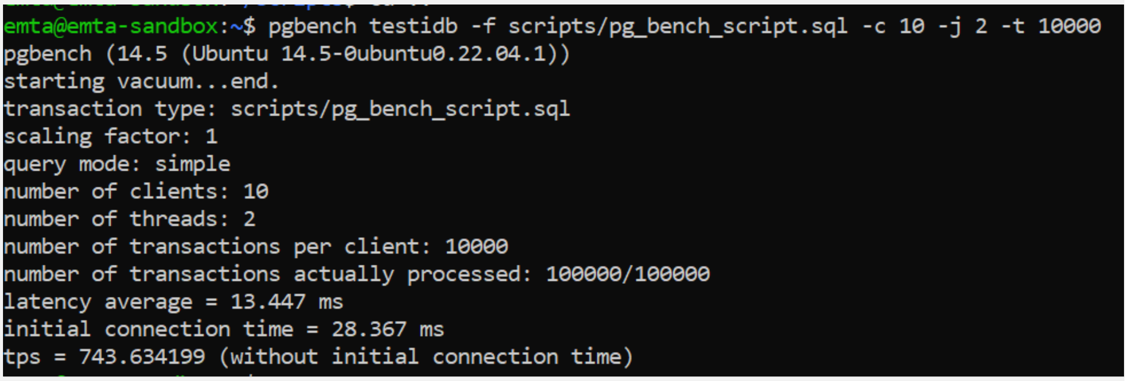
SELECT * FROM table where date >= '2019-11-24' ORDER BY date ASC;



```
emta@emta-sandbox:~$ pgbench testidb -f scripts/pg_bench_script.sql -c 10 -j 2 -t 10000
pgbench (14.5 (Ubuntu 14.5-0ubuntu0.22.04.1))
starting vacuum...end.
transaction type: scripts/pg_bench_script.sql
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 2
number of transactions per client: 10000
number of transactions actually processed: 100000/100000
latency average = 38.706 ms
initial connection time = 26.653 ms
tps = 258.356984 (without initial connection time)
```

Figure 28. Pgbench test for query with defined parameters

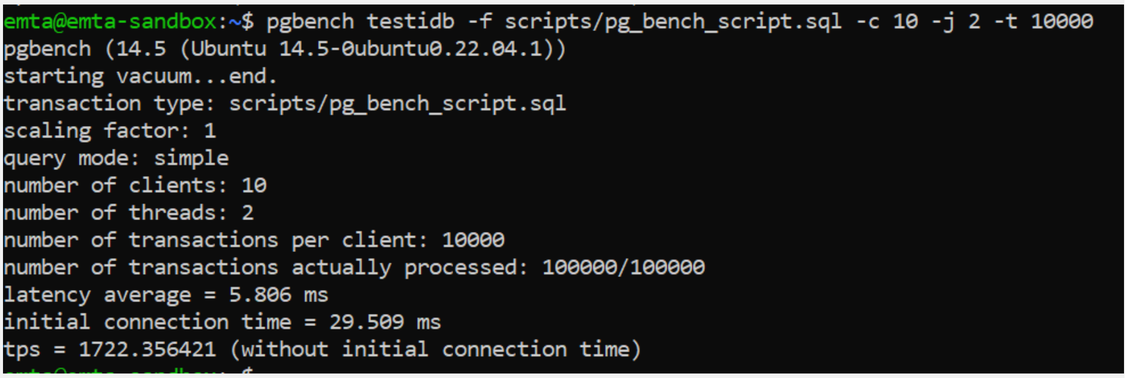
SELECT * FROM table where date >= '2019-11-24' and release_version=’version_x’ ORDER BY date ASC;



```
emta@emta-sandbox:~$ pgbench testidb -f scripts/pg_bench_script.sql -c 10 -j 2 -t 10000
pgbench (14.5 (Ubuntu 14.5-0ubuntu0.22.04.1))
starting vacuum...end.
transaction type: scripts/pg_bench_script.sql
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 2
number of transactions per client: 10000
number of transactions actually processed: 100000/100000
latency average = 13.447 ms
initial connection time = 28.367 ms
tps = 743.634199 (without initial connection time)
```

Figure 29. Pgbench test for select query

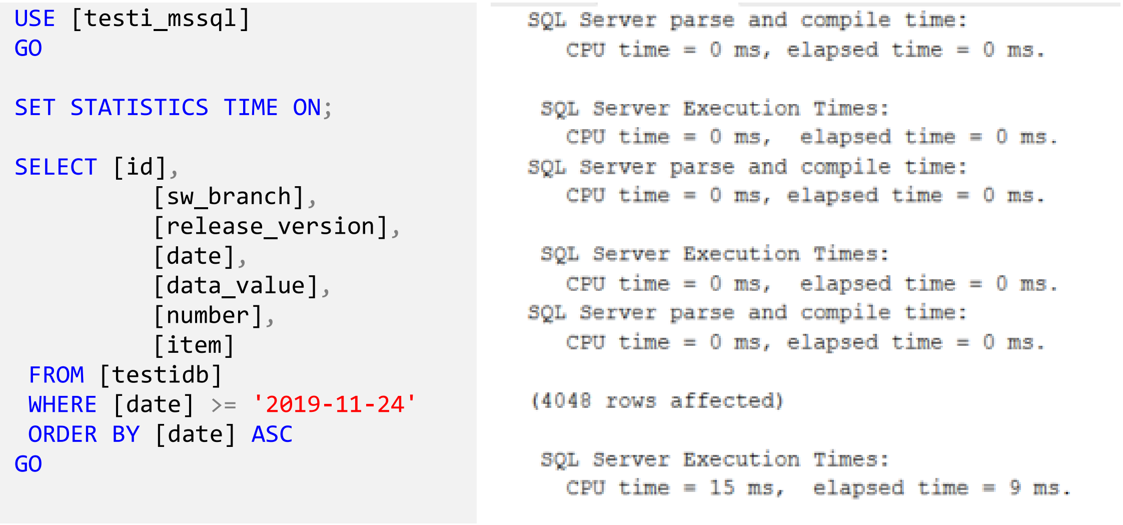
```
SELECT AVG(data_value) AS avg FROM table where date >= '2019-11-24';
```



```
emta@emta-sandbox:~$ pgbench testidb -f scripts/pg_bench_script.sql -c 10 -j 2 -t 10000
pgbench (14.5 (Ubuntu 14.5-0ubuntu0.22.04.1))
starting vacuum...end.
transaction type: scripts/pg_bench_script.sql
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 2
number of transactions per client: 10000
number of transactions actually processed: 100000/100000
latency average = 5.806 ms
initial connection time = 29.509 ms
tps = 1722.356421 (without initial connection time)
```

Figure 30. Pgbench test for calculating average value -query

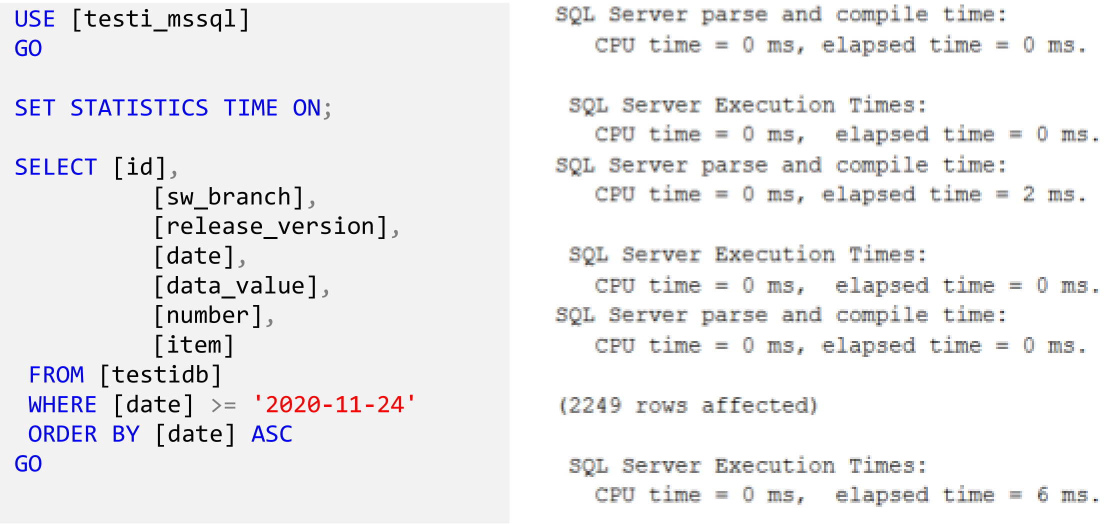
Elapsed time measurements for MS SQL Server.



```
USE [testi_mssql]
GO

SET STATISTICS TIME ON;

SELECT [id],
          [sw_branch],
          [release_version],
          [date],
          [data_value],
          [number],
          [item]
 FROM [testidb]
 WHERE [date] >= '2019-11-24'
 ORDER BY [date] ASC
GO
```

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

(4048 rows affected)

 SQL Server Execution Times:
   CPU time = 15 ms,  elapsed time = 9 ms.
```

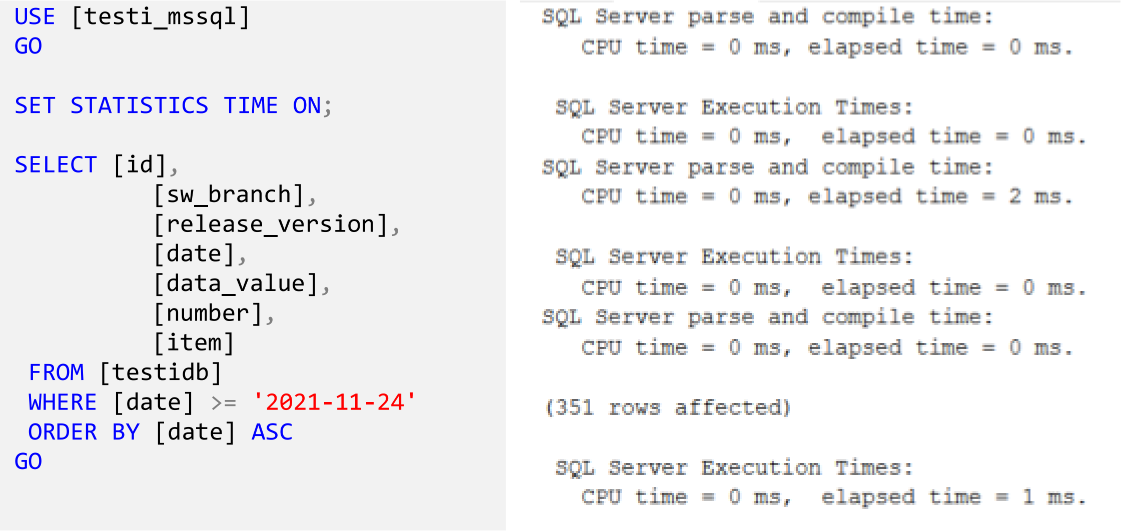
Figure 31. Execution time for selecting data from 3 years ago in date order



```
USE [testi_mssql]
GO

SET STATISTICS TIME ON;

SELECT [id],
          [sw_branch],
          [release_version],
          [date],
          [data_value],
          [number],
          [item]
 FROM [testidb]
 WHERE [date] >= '2020-11-24'
 ORDER BY [date] ASC
GO
```

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 2 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

(2249 rows affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 6 ms.
```

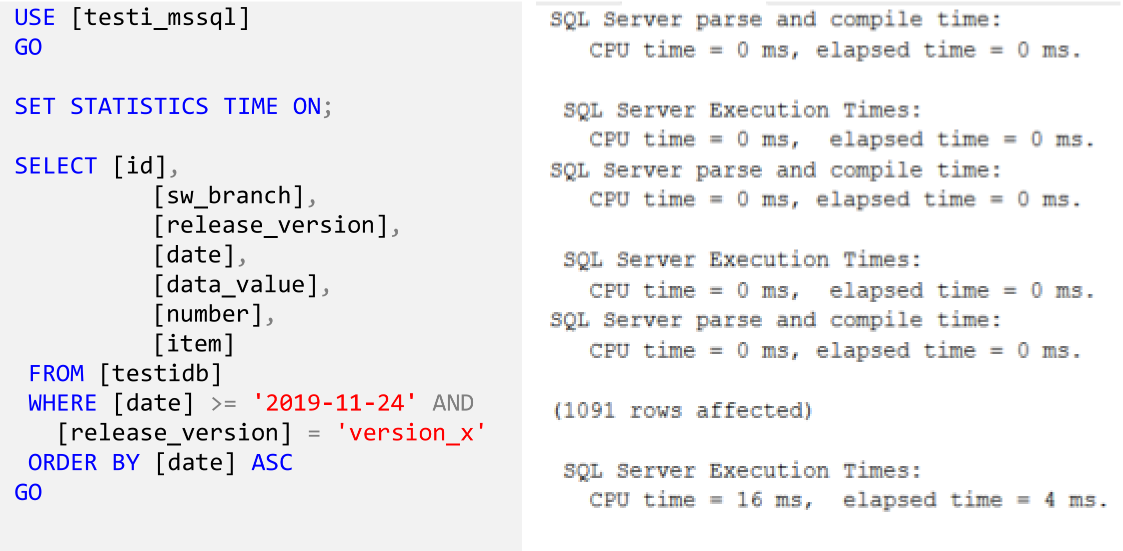
Figure 32. Execution time for selecting data from 2 years ago in date order



```sql
USE [testi_mssql]
GO

SET STATISTICS TIME ON;

SELECT [id],
          [sw_branch],
          [release_version],
          [date],
          [data_value],
          [number],
          [item]
 FROM [testidb]
 WHERE [date] >= '2021-11-24'
 ORDER BY [date] ASC
GO
```

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 2 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

(351 rows affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 1 ms.
```

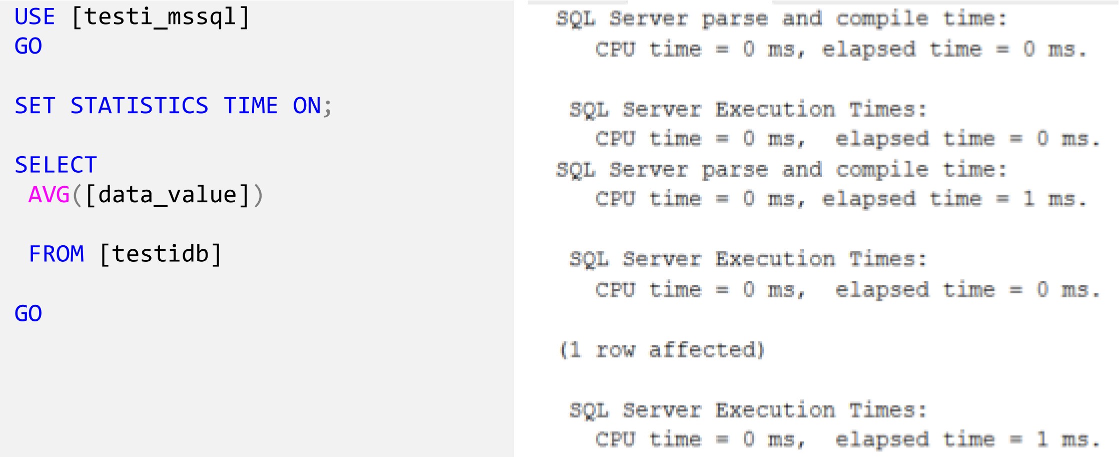
Figure 33. Execution time for selecting data from one year ago in date order



```sql
USE [testi_mssql]
GO

SET STATISTICS TIME ON;

SELECT [id],
          [sw_branch],
          [release_version],
          [date],
          [data_value],
          [number],
          [item]
 FROM [testidb]
 WHERE [date] >= '2019-11-24' AND
   [release_version] = 'version_x'
 ORDER BY [date] ASC
GO
```

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

(1091 rows affected)

 SQL Server Execution Times:
   CPU time = 16 ms,  elapsed time = 4 ms.
```

Figure 34. Execution time for selecting defined release version's data from 3 years ago in date order



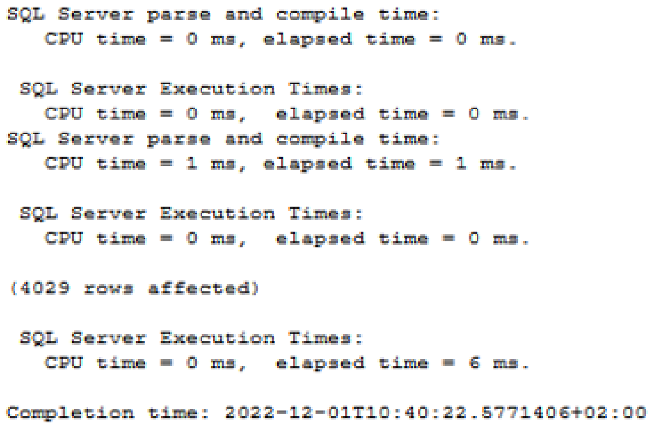
```
USE [testi_mssql]
GO

SET STATISTICS TIME ON;

SELECT
 AVG([data_value])

 FROM [testidb]

GO
```

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 1 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

(1 row affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 1 ms.
```

Figure 35. Execution time for selecting average value

Execution times measured for MS SQL Server with table style 2.

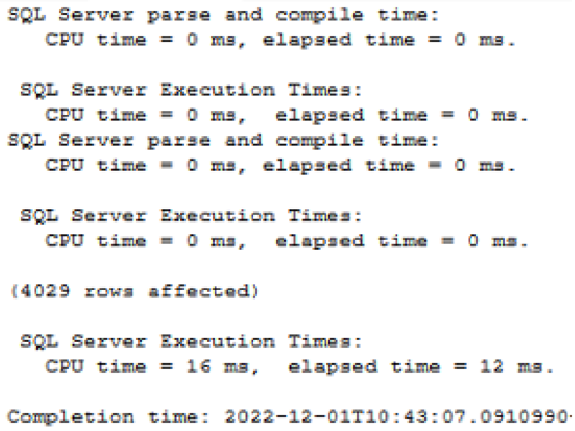
```
USE [testidb]
GO

SET STATISTICS TIME ON;

SELECT
[dbo].[style2].[id],
[dbo].[style2].[sw_branch],
[dbo].[branch_x].[version_x],
[dbo].[branch_x].[date],
[dbo].[branch_x].[data_value],
[dbo].[branch_x].[number],
[dbo].[branch_x].[item]

FROM [dbo].[style2]
INNER JOIN [dbo].[branch_x] ON
[dbo].[style2].[id] =
[dbo].[branch_x].[id]

GO
```



```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 1 ms, elapsed time = 1 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

(4029 rows affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 6 ms.

Completion time: 2022-12-01T10:40:22.5771406+02:00
```

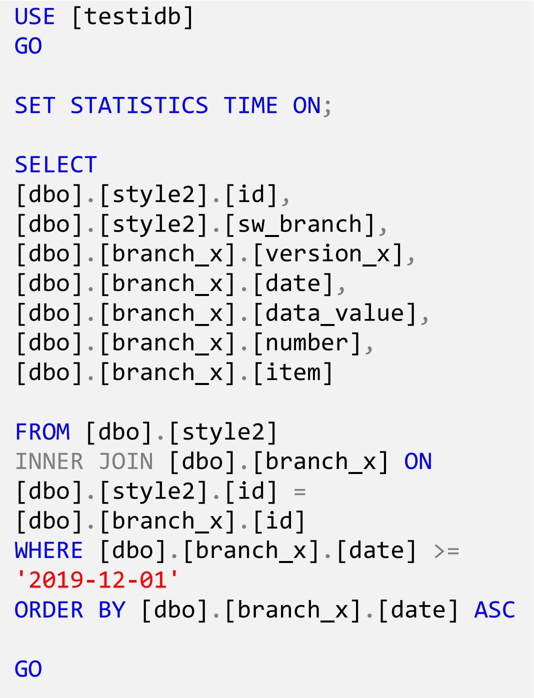
Figure 36. Execution time for joining two tables

```
USE [testidb]
GO

SET STATISTICS TIME ON;

SELECT
[dbo].[style2].[id],
[dbo].[style2].[sw_branch],
[dbo].[branch_x].[version_x],
[dbo].[branch_x].[date],
[dbo].[branch_x].[data_value],
[dbo].[branch_x].[number],
[dbo].[branch_x].[item]

FROM [dbo].[style2]
INNER JOIN [dbo].[branch_x] ON
[dbo].[style2].[id] =
[dbo].[branch_x].[id]
ORDER BY [dbo].[branch_x].[date]
ASC

GO
```

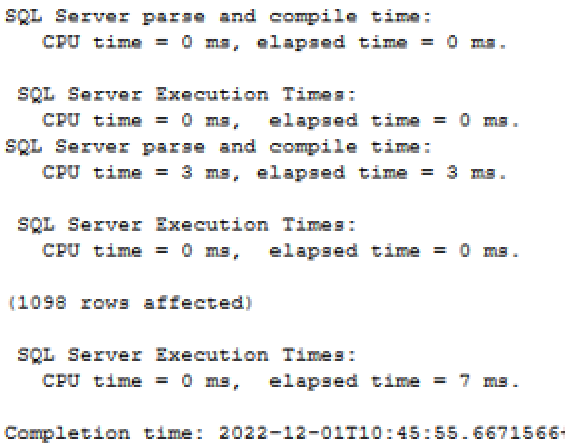


```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

(4029 rows affected)

 SQL Server Execution Times:
   CPU time = 16 ms,  elapsed time = 12 ms.

Completion time: 2022-12-01T10:43:07.0910990
```

Figure 37. Execution time for joining two tables in date order

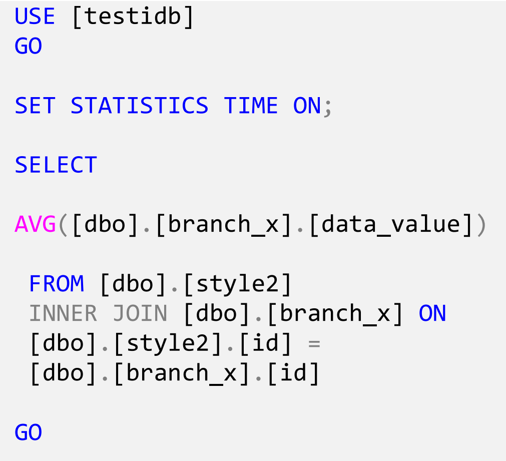


```
USE [testidb]
GO

SET STATISTICS TIME ON;

SELECT
[dbo].[style2].[id],
[dbo].[style2].[sw_branch],
[dbo].[branch_x].[version_x],
[dbo].[branch_x].[date],
[dbo].[branch_x].[data_value],
[dbo].[branch_x].[number],
[dbo].[branch_x].[item]

FROM [dbo].[style2]
INNER JOIN [dbo].[branch_x] ON
[dbo].[style2].[id] =
[dbo].[branch_x].[id]
WHERE [dbo].[branch_x].[date] >=
'2019-12-01'
ORDER BY [dbo].[branch_x].[date] ASC

GO
```

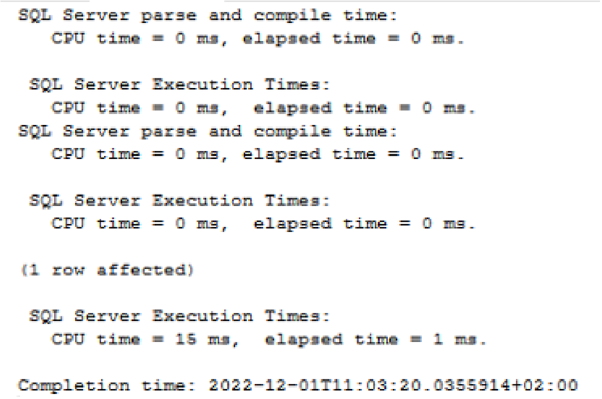


```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 3 ms, elapsed time = 3 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

(1098 rows affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 7 ms.

Completion time: 2022-12-01T10:45:55.6671566
```

Figure 38. Execution time for joining two tables from defined date in date order



```
USE [testidb]
GO

SET STATISTICS TIME ON;

SELECT

AVG([dbo].[branch_x].[data_value])

 FROM [dbo].[style2]
 INNER JOIN [dbo].[branch_x] ON
 [dbo].[style2].[id] =
 [dbo].[branch_x].[id]

GO
```



```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

(1 row affected)

 SQL Server Execution Times:
   CPU time = 15 ms,  elapsed time = 1 ms.

Completion time: 2022-12-01T11:03:20.0355914+02:00
```

Figure 39. Execution time for calculating average value from joined table