

A Query-Rewriting-based Implementation for Temporal Data Management

Reinhold Schlager¹, Johannes Schildgen¹

¹*Ostbayerische Technische Hochschule Regensburg (OTH Regensburg), Seybothstraße 2, 93053 Regensburg, Germany*

Abstract

Temporal data management is used to query historical data, identify trends and changes, and make forecasts for the future. This paper provides a query-rewriting approach to implement temporal data management in the data-warehouse system Exasol by using so-called preprocessor scripts. Preprocessor scripts allow for executing custom Lua code and modifying SQL commands. Our approach does not need to undertake additional manual effort by the user. History tables and associated views are created in the background. Data records are automatically archived if they are updated or deleted, and the SELECT command is extended with the AS OF SYSTEM TIME clause as proposed by the SQL Standard.

Keywords

Temporal data management, Data warehouse, Exasol, Database

1. Introduction

The temporal progression of data plays an important role in the world of modern information systems. Historically accurate data is important for gaining knowledge and to trace changes. Data analysis is important in many industries. In combination with current data, they form the basis for evidence-based decision-making. Historical storage is of great importance and the newly acquired additional information can be used to make data-driven decisions and recognise trends [1]. This is also useful for predictions and temporal analyses. With the help of time-travel-queries, it is possible to determine the status of a data record at a certain point in time in the past.

In this paper, we present an approach to implement system-versioned tables that is based on query rewriting. For this, we use the data-warehouse system Exasol. A proof-of-concept implementation will be used to show that this can be done using Exasol's preprocessor scripts. Care is taken to ensure that the temporal data management implementation conforms to the SQL Standard [2]. Our code is open-source and can be found on GitHub [3].

LWDA'24: *Lernen, Wissen, Daten, Analysen. September 23–25, 2024, Würzburg, Germany*


✉ reinhold.schlager@st.oth-regensburg.de (R. Schlager); johannes.schildgen@oth-regensburg.de (J. Schildgen)

🌐 <https://github.com/RashSR/Exasol-Preprocessor-Temporal> (R. Schlager)

🆔 0000-0002-2450-0152 (J. Schildgen)



© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

2. Temporal data management

Temporal data management is the process of historical data storage and access. In a database or data warehouse, only data entries that are valid at the current time are accessible by default. If a record is updated or deleted, its previous state is no longer available. In selected cases, it is profitable to have access to these previous states. For that reason, it is important that every change is fully recorded [4]. For each data record, a full history of its current and previous states is stored which allows users to access values for a specific point in time in the past.

2.1. Data validity

The literature typically distinguishes between two models for temporal data management. The first one is called *valid-time* and describes the time interval in which a data record is or was valid in the real world [5]. Typically, these timestamps are supplied by a user or an application. The start and end timestamp of the validity are noted, for example, in two attributes `ta_start` and `ta_end`. The validity can be set in advance or retrospectively. The SQL Standard also introduced the concept for time periods to allow complex temporal queries. To compare intervals, Allen's interval algebra [6] is used. As an example, for two given temporal intervals I_1 and I_2 . $I_1 < I_2$ means that the interval I_1 is before I_2 without any overlap.

In contrast to the valid-time model, the *transaction-time* describes the time at which a data record was inserted, updated, or deleted in the database [4]. This means that the transaction-time has a clearly defined start and end timestamp. The period begins when the element is added and ends when it is deleted or updated. In the valid-time model, the data mostly describe entities and facts together with their temporal validity. The transaction-time model is exempt from this limitation and can therefore be applied to any database entity. The database automatically provides its data with timestamps and therefore no manual work is required. In the following

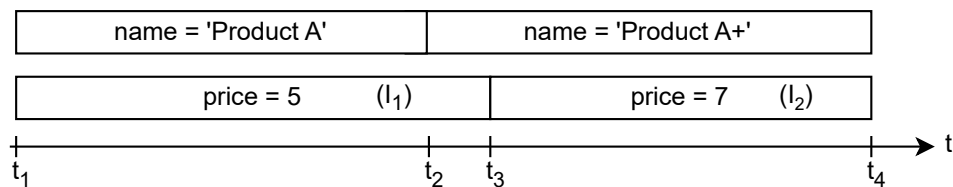


Figure 1: data validity time example

example, a department store sells products in a web store. An item is called “Product A”. Figure 1 shows the validity of the name and the price of that product. Both start values are entered in the database at insertion time t_1 . The name becomes invalid at time t_2 and is replaced by the updated name “Product A+”. The price is then raised at time t_3 . These changes are automatically visible within the webshop application because per default, only the current values are accessed. The product is deleted from the database at time t_4 and therefore all information about it is invalidated.

These two models use a uni-temporal representation [7], which means that each data item has one start and end time. When combining the two models, the term bi-temporal data management is used. This adds another start and end time to the uni-temporal representation. For example, the transaction-time and a further area of validity can be saved in a data record. This makes it possible to use the two aforementioned models at the same time. In comparison to these two types of representation, the non-temporal representation has no temporal addition.

2.2. Time-Travel-Queries in the Transaction-Time Model

To query an already outdated value, the user has to provide a timestamp of the past. In order for such time-travel-queries to be possible, each data record requires additional temporal information, as already discussed in the paragraph above. For example, Figure 2 shows the development of a product price over time. The start time is inclusive and the end time is exclusive. A query is made to determine the price at time t_2 . This time is between t_1 and t_3 and the query therefore returns the value 5. This shows that outdated values can be retrieved using a time-travel-query. In SQL, this query would correspond to `SELECT price FROM product AS OF SYSTEM TIME t_2 ;`

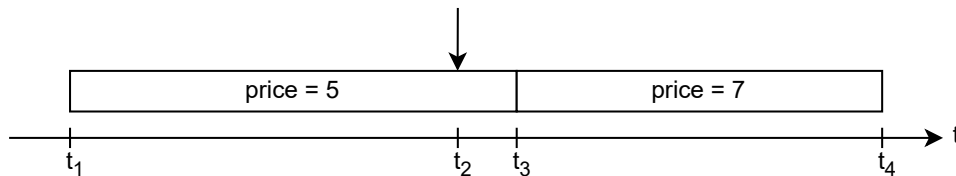


Figure 2: time-travel-query for a specific timestamp

3. Technological Background

Several DBMS have a built-in support for temporal data management, or they can be extended with this feature. Oracle provides temporal data management through Temporal Validity Support [8] for chosen columns. This is often combined with Oracle Flashback Technology [9], which is used to recover outdated values. A query can be extended by an `AS OF` or `VERSIONS BETWEEN` clause and the specified timestamp. Microsoft SQL Server uses system-versioned tables [10] to automatically maintain historical versions of data records. Temporal queries are made with the `FOR SYSTEM_TIME` clause. PostgreSQL doesn't support temporal data management out-of-the-box, but there is a widely used extension [11] that provides such a feature. The authors of this extension use database trigger to implement historical storage. MySQL supports basic temporal data types [12] and MariaDB supports temporal data management natively [13]. IBM's Db2 offers the most variety of temporal management [14] and has built-in features that support transaction-time, valid-time and bi-temporal tables that closely align with the SQL Standard [2]. Each DBMS has a unique approach to handle temporal data management.

3.1. Exasol and Preprocessor

Exasol is a data warehouse system and does not provide a temporal data management feature by default. Furthermore, Exasol has no support for trigger. This is why we want to use preprocessor scripts to implement temporal data management. Preprocessor scripts can be used to rewrite an SQL query before its execution [15]. Each SQL command is processed by the preprocessor. Within preprocessor scripts, it is possible to execute additional SQL commands. At the end of the script, the original command or its modification is executed by the DBMS. The developer of a preprocessor script can specify which SQL commands are changed and in what way. As long as the rewritten command corresponds to a valid SQL syntax, there are no restrictions. Preprocessor scripts are often used to implement unsupported SQL features and to simplify the use of existing constructs [15].

As an example, the TRUNCATE [16] command is intended to ensure greater compatibility with the help of the preprocessor. The syntax of this command in Exasol is TRUNCATE TABLE <tbl_name>, but legacy applications could use the Db2 syntax TRUNCATE <tbl_name>. If the preprocessor recognizes TRUNCATE followed by an identifier, it can modify the SQL command accordingly. This means that legacy applications that use such a syntax can also be executed in Exasol.

The preprocessor is deactivated by default and can be activated manually for each session or for the whole system. If the preprocessor uses the “myPreprocessor” script, the activation command is ALTER SESSION SET sql_preprocessor_script = myPreprocessor. To deactivate the preprocessor, the parameter sql_preprocessor_script must be set to null. The preprocessor script is executed for each query that is sent to the database except ALTER SESSION commands and commands that include passwords e.g. ALTER USER.

Parsing SQL commands is simplified by the sqlparsing library [15]. With this Lua library, it is therefore possible to tokenize the command into keywords, spaces, identifiers and much more. The preprocessor examines the input and can decide how to process it based on the underlying action. For example, if we want to extract the table name from this simple command: CREATE TABLE test_tbl (id INT), we can iterate over the list of tokens. The first token is an SQL keyword, which is CREATE. This can be checked using the isKeyword() method from the sqlparsing library. The next token is a whitespace followed by the keyword TABLE, again a whitespace, and then, an identifier. In our example, the first identifier is the table name. The sequence of tokens can be checked to determine which kind of statement was sent to the DBMS. Preprocessor scripts are particularly important for this work because they can be executed automatically. They are a simple way of implementing a data versioning feature that does not require any additional manual user effort. The preprocessor can also be debugged, making the development process easier.

Preprocessor scripts are coded in the programming language Lua. Lua is a powerful and efficient scripting language. It supports concepts such as object-oriented programming, functional programming, data-driven programming and procedural programming [17]. The complete versioning logic of our approach is programmed in Lua. The coded scripts process SQL commands and extend Exasol with a temporal data management feature.

3.2. Data Model

For the examples in this paper, we will use a simple data model including people and their bank accounts. Both tables contain a unique id. Table 1 and Table 2 show the structure of these tables together with a sample row. The person table is referenced using a foreign key. An account balance changes frequently due to incoming and outgoing payments and is therefore a good example for permanent maintenance of a history table. The transaction-time model is utilized as users do not have to manually maintain timestamps and to automate this process.

id	first_name	last_name
1	Marty	McFly

Table 1: Person table

id	balance	person_id
3	3000	1

Table 2: Account table

4. Implementation

4.1. Implementation Alternatives

There are several approaches to conceptualize a temporal data management feature. The first step is to create a table that persists data records together with temporal information. The preprocessor rewrites the `CREATE TABLE` command. There are single or multiple-table solutions and in the following, three possible proposals are discussed. The first approach is to persist both current and outdated data within one single table. Table 3 shows such a storage. A historical record is highlighted in grey. This colour-coded representation is inspired by [1]. The first entry is outdated as of July 1st at 2:56 pm and was replaced by the other row in the shown table. For current data, the end time is set to the maximum possible value '9999-12-31 23:59:59.999'. One disadvantage of this single-table approach is that both current and outdated data are arbitrarily mixed together and extended by the two timestamps. Nevertheless, the preprocessor script can modify user queries to hide the additional columns and filter the rows with respect to the users' needs.

id	balance	person_id	ta_start	ta_end
3	3000	1	2024-06-21 12:23	2024-07-01 14:56
3	2599	1	2024-07-01 14:56	9999-12-31 23:59

Table 3: Historical storage in one table

Another possible solution is to use two tables. One user table for the current data and one history table. New data is added to the user table and after an update or deletion, it is transferred to the history table and invalidated. This transfer can be done by the preprocessor script. Another approach following this idea would be to save all data in the history table, including the current data. This data would then also be available in the user table and stored redundantly. This has a number of positive aspects in terms of the implementation and performance. E.g., a historical query must both read the user table and the history table. This will be discussed in a later section.

An advantage of this two-table approach is that no data record in the history table is ever changed. Splitting the data into current and outdated data encapsulates them from another. A query that accesses current rows can directly be sent to the user table. Only the `ta_start` column should be hidden. The user table does not require a `ta_end` column, as the user table does not contain outdated records.

id	balance	person_id	ta_start
3	2599	1	2024-07-01 14:56

Table 4: improved user table

This is already a suitable solution but can be improved further. To give the user an unaltered experience, the idea was that the user table has no timestamp columns. This results in the problem that inserted lines in the user table must be redundantly saved in the history table. Without saved timestamps, it is not trivial to find out which lines have been added or updated. The first idea for a matching were set operations. It is possible to subtract one table from another using the `EXCEPT` set operation [18]. Figure 3 illustrates this process. This logic is used to find new data records in the user table. The historical table can then be subtracted from the user table and the result are the new data records. Such a set operation compares all line entries with each other and has therefore a bad performance.

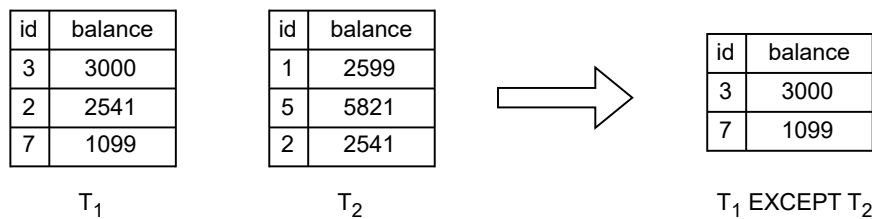


Figure 3: Example for `EXCEPT`

An approach with better performance is when each line can be uniquely identified, for example by a primary key. However, a table can also be created without a primary key and therefore this approach is not expedient as it is not universally applicable. Exasol offers an internal identifier for each row called `ROWID` [19]. If this address is persisted when adding or changing data records, each line can be determined very quickly. However, the `ROWID` is changed by the DBMS each time a data item is updated and so it is not suitable to keep track of the row.

The final solution concept uses a history table similar to the first approach. All data is saved in this table. Instead of a second table, a user view is created. SQL commands can be executed on this virtual table in the same way as on a normal table. In this case, only valid rows with the columns `id` and `balance` should be displayed. This checks whether the `ta_end` column is greater or equal than the current time. If this is the case, the row is displayed in the user view. The command to generate a view called `account` is `CREATE VIEW account AS SELECT id, balance FROM account_hist WHERE ta_end >= CURRENT_TIMESTAMP`. For

a given history (table 5) the user view is shown in Table 6. It is clearly to see that only the valid row is displayed without any temporal information. With this approach, less data is stored redundantly, as all data is logically stored in the history table.

id	balance	person_id	ta_start	ta_end
3	2599	1	2024-07-01 14:56	9999-12-31 23:59
3	3000	1	2024-06-21 12:23	2024-07-01 14:56

Table 5: history table for user view

id	balance	person_id
3	2599	1

Table 6: user view with only valid values

The user can make requests on the view. The resulting SQL command is modified using Exasol's preprocessor so that an INSERT query is forwarded directly to the history table. This happens with every SQL command. The system thus gives the impression that the user is working on the user view, although only the data in the history table changes. As a result, inserted data records do not need to be found as they already exist in the correct table. Queries that access current data can be executed directly on the view without any modification by the preprocessor script.

4.2. Implementation

4.2.1. Table creation

In our implementation, the user can not activate historical storage, because each table is versioned automatically. This means that if a table is created, the associated view and history table are generated automatically. As an example, the following SQL command is used: CREATE TABLE Person LIKE PersonTemplate. This command generates a table with the same columns like Table 1. At first, the table name gets extracted. Next, the SQL statement is executed and creates the table Person. Now the table Person is renamed to HIST_Person and the two timestamp columns ta_start and ta_end are added. The default value for ta_end is the greatest possible timestamp, and for ta_start, it is the current timestamp.

Each history table has the prefix HIST_. This history table is depicted in Table 7. In the last step, the view is created. Therefore, the saved column names are used to only show the needed columns without the temporal information and this can be seen in Table 8. The command that generates such a view is already discussed in Section 4.1.

id	first_name	last_name	ta_start	ta_end

Table 7: History table HIST_Person

id	first_name	last_name

Table 8: View Person

4.2.2. Insertion of data

There are many ways to insert data into a table. One command can insert one or many data records at once. Some information can be omitted e.g. the column names. The inserted data can be specified in the SQL command or added from another table or file. In this paper, we focus on INSERT INTO statements. The first one, that will be discussed is depicted in Figure 4. The user provided all required information. The preprocessor recognizes the INSERT INTO syntax followed by column names and the keyword VALUES and rewrites the query. The table name is replaced by the history-table name. Nothing more is necessary and after execution, the content of both tables is shown in Table 9 and 10. This is the case due to the fact that the timestamp columns have a default value.

INSERT INTO Person (id, First Name, Last Name) VALUES (1, 'Marty', 'McFly')

↓

INSERT INTO HIST_Person (id, First Name, Last Name) VALUES (1, 'Marty', 'McFly')

Figure 4: INSERT SQL command rewriting for a single insert

id	first_name	last_name	ta_start	ta_end
1	Marty	McFly	2024-06-20 12:23	9999-12-31 23:59

Table 9: History table HIST_Person after INSERT

id	first_name	last_name
1	Marty	McFly

Table 10: View Person after INSERT

Another SQL command we look at is one that adds more than one data record. This is pictured in Figure 5. The shown command inserts two rows without specifying the column names. In this situation, we need to read the list of columns from system tables. The preprocessor must not only change the table name to the history table name, but also add the column names. This is necessary due to the fact that the timestamp columns ta_start and ta_end need to be filled by their aforementioned default values.

An SQL command that inserts values from another table, like e.g., INSERT INTO account SELECT * FROM otherTable must also be rewritten. Like in every other rewriting process, the statement is going to be executed on the history table. The * symbol must be resolved into the actual column names. The preprocessor recognizes this and rewrites the statement to the following INSERT INTO HIST_account (id, balance) SELECT (id, balance) FROM otherTable.


```

INSERT INTO Person VALUES (2, 'Gretchen', 'Ross'), (3, 'Donnie', 'Darko')
      ↓
INSERT INTO HIST_Person (id, first_name, last_name) VALUES (2, 'Gretchen', 'Ross'), (3, 'Donnie', 'Darko')

```

Figure 5: INSERT SQL command rewriting for more than one value

4.2.3. UPDATE queries

The most complex step in the versioning of historical data are UPDATE queries on existing data records. As an example, we use the following command: `UPDATE Person SET last_name = 'Parker' WHERE id = 1`; The resulting history table is shown in Table 11. The first row has the time of the update in the column `ta_end`, because it got outdated at that timestamp. To make this possible, the first step is to find the data sets that are going to be changed and invalidate them. Therefore, the `ta_end` column is set to the current time. This time will be used later and is therefore stored temporarily in a variable `currentTime`. The lines are located with the `WHERE` predicate from the original SQL statement. This happens on the history table. Afterwards, these now outdated lines are inserted once more in the history table, but with the maximum timestamp possible in the `ta_end` column. The rows can be uniquely determined by the variable `currentTime`. These lines are now twice in the history table, but with different temporal information. The UPDATE command can then be executed on the current data. In our example, this happens with the following SQL statement `UPDATE HIST_Person SET last_name = 'Parker' WHERE id = 1 AND ta_end > CURRENT_TIMESTAMP`. The update is performed on the history table and to ensure that the right data is changed, the `WHERE` predicate is supplemented with the previously mentioned variable `currentTime`. After the execution of that sequence of SQL commands, the right data is updated and the result is depicted in Table 12. If such a valid SQL query is executed, the number of updated rows is given to the user as feedback from the preprocessor.

id	first_name	last_name	ta_start	ta_end
1	Marty	McFly	2024-06-20 12:23	2024-07-09 17:34
1	Marty	Parker	2024-07-09 17:34	9999-12-31 23:59

Table 11: History table HIST_Person after UPDATE

id	first_name	last_name
1	Marty	Parker

Table 12: View Person after UPDATE

4.2.4. DELETE queries

If a data record is to be deleted, it must be marked as outdated. This means that its end time column `ta_end` in the history table should be the time of deletion. In system-versioned tables,

no data really gets deleted. This means that all DELETE SQL commands are going to be rewritten by the preprocessor to UPDATE statements. To delete all data entries in the user view following SQL statement is used; `DELETE FROM Person`. Like in the previous cases, this command must be executed on the history table. Additionally, this command has to be rewritten by the preprocessor into an UPDATE command, which only is applied on current rows. The preprocessor rewrites the query to the following statement: `UPDATE HIST_Person SET ta_end = CURRENT_TIMESTAMP WHERE ta_end > CURRENT_TIMESTAMP`. This results in the ongoing example in an empty user view like Table 8, due to the fact that all data in the history table is outdated and therefore there are no current data records.

id	first_name	last_name	ta_start	ta_end
1	Marty	McFly	2024-06-20 12:23	2024-07-09 17:34
1	Marty	Parker	2024-07-09 17:34	2024-07-10 10:02

Table 13: History table HIST_Person after DELETE

To handle more specific DELETE FROM SQL statements, the WHERE predicate has to be used from the original SQL command like in section 4.2.3. The preprocessor checks if such a predicate is present and rewrites the query accordingly to the given use case. As feedback, the user gets the count of deleted rows. For this, the number of affected rows from the last update query is used.

4.2.5. SELECT: Querying historical data

All SELECT queries on current data do not have to be rewritten, because they are performed on the user view. To retrieve outdated data records, a specified timestamp in the past must be added to an SQL query. This is indicated by the “AS OF SYSTEM TIME” [20] clause in the FROM clause. Such a sequence is checked by the preprocessor script. It iterates through the tokens and if all the four keywords are in the right order, it also checks if the following token after the whitespace is a string literal in timestamp format. If this is the case, the historical query will be executed and returns the value for that moment in time. An example for such a query is `SELECT last_name FROM Person AS OF SYSTEM TIME '2024-07-27 15:41'`. When this query is executed, the retrieved value is “McFly” even if it is not the current value. This data record was valid at the specified time, which is depicted in Figure 6. To achieve this goal, the preprocessor rewrites the query to address the history table. The given timestamp :tStamp is wrapped into following syntax `WHERE ta_start <= :tStamp AND ta_end > :tStamp`. This ensures that only the values, which were current at :tStamp are shown in the result. If the original SQL command already has a WHERE predicate, this temporal part must be added with AND.

4.2.6. Automation with help of the Preprocessor

The preprocessor acts as a middleware to adapt the temporal data management feature. The preprocessor takes an SQL query and analyzes it with the help of the sqlparsing library and executes the associated Lua script. Each command like CREATE, DROP, INSERT, UPDATE, DELETE and SELECT has an own Lua script, which is called by the preprocessor. Once switched on, it is

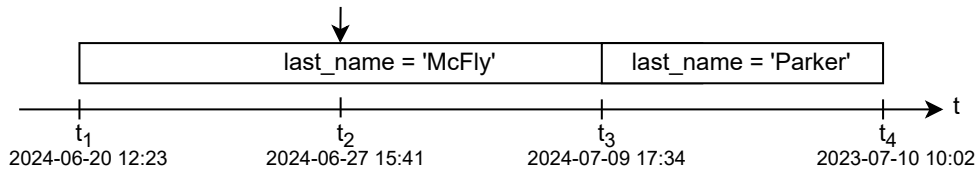


Figure 6: Example for a time travel query

continuously active and checks every command. The Lua scripts execute several SQL commands, which are then also being checked. Care must be taken that these resulting commands are not rewritten. The preprocessor checks if the command affects a history table and executes it unmodified. All history tables have the prefix “HIST_” and can be identified easily. After a script is executed, all temporal tasks are fulfilled. It is not possible to discard the original SQL query and so the preprocessor modifies it too. It is rewritten to a meaningful status text that tells the user, for example, how many entries were added by their INSERT command. This information can be retrieved from the result of a SQL query.

5. Evaluation

5.1. Discussion

The proposed solution shown allows historical data versioning in Exasol. Despite the numerous implemented functionalities, the implemented solution concept of historical storage is still only a proof of concept. In order for the project to leave this status, a few incompatibilities still need to be fixed and quality-of-life features added. There is room for improvement to support more SQL commands. Another improvement that has not yet been implemented is more detailed error handling. So far, the user only receives very simple feedback as to whether a command was successful or not. More meaningful error messages are a good enhancement. A Lua script processes several SQL commands. If one of these fails, all previous commands must be reset. This is not yet integrated in the current implementation. It is also possible to further expand the time-travel queries. Currently, only a fixed time can be searched for, as this is expected as a string literal. In future, it should be possible to modify the specified time temporally, for example by subtracting 3 days from it.

5.2. Quality Assurance

This specific prototype is tested using JUnit [21]. In the test implementation, the connection to the Exasol data warehouse is established in a BeforeClass block using JDBC [22] and the connection is closed once all tests have been completed. For a closer look, the test can be viewed at [3]. All supported SQL transformations have been tested and found to be correct as intended.

To ensure that the performance is not impacted in a great negative way, the application was tested using PyExasol [23]. With this tool, SQL queries were sent to the Exasol database and the

execution time was recorded. In the first part of the measurement, the historical storage was turned off and in the second part it was active. This was repeated 10 000 times and the result of this performance measuring comparison is depicted in Figure 7. The biggest performance loss is the creation of a new table and this can be explained by the steps involved, such as the creation of the user view and the renaming to the history table. Tables are only created once and therefore this is negligible. Most of the other commands extend the runtime of an SQL query between 30 and 40 percent. Across all of the tests, it can be seen that the processing time of the commands with active query rewriting is only 31% longer than usual. The benefits of historical storage outweigh the loss in performance.

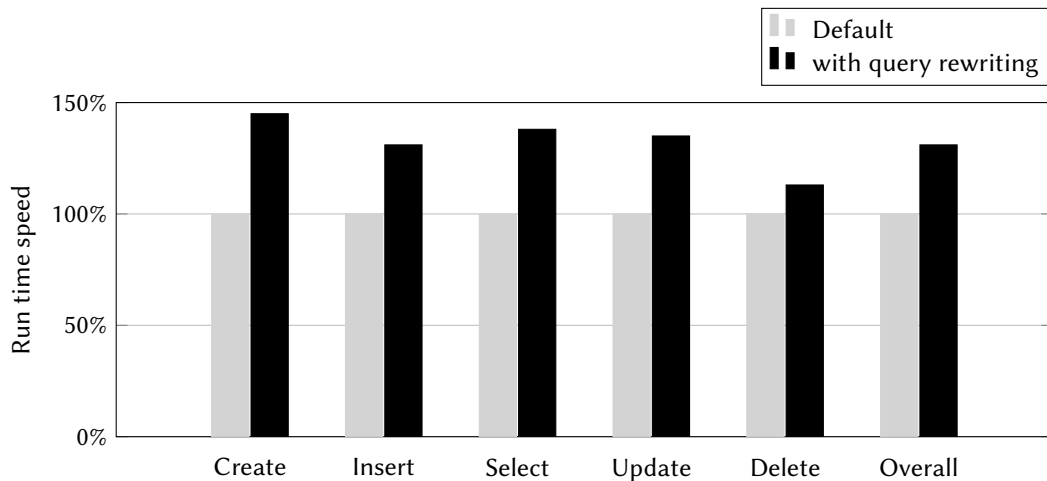


Figure 7: Performance measurement graph

6. Conclusion

The motivation for this work was to enable the historical storage of data in Exasol. The main focus was the traceability of data changes and their retrieval. The final solution concept is implemented with Exasol's preprocessor. The preprocessor rewrites SQL commands by using Lua scripts. All current and outdated data sets are saved in a history table. The validity is indicated by two timestamps `ta_start` and `ta_end`, which represent the transaction-time model. The timestamps are set automatically by the preprocessor script. Additionally, there is an SQL view that only contains the currently valid data without any temporal information. If a point in time in the past is of interest, the database status at this point in time can be queried. This proof of concept shows that automatic historical versioning can be implemented in Exasol and other DBMS that support query rewriting. However, there are some SQL commands that are not yet compatible with the prototype. The more specific the queries, the more special cases accumulate. The future of this tool is to close these compatibility gaps and make it fully operational.

References

- [1] C. M. Saracco, M. Nicola, L. Gandhi, A matter of time: Temporal data management in db2 for z, IBM Corporation, New York 7 (2010).
- [2] ISO/IEC 9075:2023, Information technology – Database languages SQL, Standard, International Organization for Standardization, Geneva, CH, 2023.
- [3] R. Schlager, Exasol preprocessor temporal, <https://github.com/RashSR/Exasol-Preprocessor-Temporal>, 2023.
- [4] C. S. Jensen, R. T. Snodgrass, Temporal data management, *IEEE Transactions on knowledge and data engineering* 11 (1999) 36–44.
- [5] C. S. Jensen, R. T. Snodgrass, L. Liu, Valid time., 2018.
- [6] M. Grüninger, Z. Li, The time ontology of allen’s interval algebra, in: 24th International Symposium on Temporal Representation and Reasoning (TIME 2017), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
- [7] T. Johnston, R. Weis, Managing time in relational databases: how to design, update and query temporal data, Morgan Kaufmann, 2010.
- [8] Oracle Corporation, Database development guide, 2024. URL: https://docs.oracle.com/en/database/oracle/oracle-database/23/adfns/design_basics.html, [Online; accessed 22-June-2024].
- [9] Oracle Corporation, Flashback technologies, 2024. URL: <https://www.oracle.com/database/technologies/flashback/>, [Online; accessed 22-June-2024].
- [10] Microsoft Corporation, Temporal tables, 2024. URL: <https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables?view=sql-server-ver15>, [Online; accessed 22-June-2024].
- [11] V. A. Martin Clausen, Exasol preprocessor temporal, https://github.com/arkhipov/temporal_tables, 2023. [Online; accessed 22-June-2024].
- [12] Oracle Corporation, Date and time data types, 2024. URL: <https://dev.mysql.com/doc/refman/8.0/en/date-and-time-types.html>, [Online; accessed 22-June-2024].
- [13] MariaDB Corporation, Temporal tables, 2024. URL: <https://mariadb.com/kb/en/temporal-tables/>, [Online; accessed 07-July-2024].
- [14] International Business Machines Corporation, Temporal tables and data versioning, 2024. URL: <https://www.ibm.com/docs/en/db2-for-zos/13?topic=tables-temporal-data-versioning>, [Online; accessed 22-June-2024].
- [15] Exasol AG, Exasol documentation: Sql preprocessor, 2024. URL: https://docs.exasol.com/db/latest/database_concepts/sql_preprocessor.htm, [Online; accessed 20-May-2024].
- [16] W3 Schools, Sql tutorial: Sql drop table and truncate table keywords, 2024. URL: https://www.w3schools.com/sql/sql_ref_drop_table.asp, [Online; accessed 20-May-2024].
- [17] Pontifical Catholic University of Rio de Janeiro, Lua about, 2024. URL: <https://www.lua.org/about.html>, [Online; accessed 19-May-2024].
- [18] Exasol AG, Exasol documentation: Table operators, 2024. URL: https://docs.exasol.com/db/latest/sql/table_operators.htm, [Online; accessed 26-May-2024].
- [19] Exasol AG, Exasol documentation: Rowid, 2024. URL: https://docs.exasol.com/db/latest/sql_references/functions/alphabeticallistfunctions/rowid.htm, [Online; accessed 26-May-2024].

- [20] Cockroach Labs, Cockroach labs documentation: As of system time, 2024. URL: <https://www.cockroachlabs.com/docs/stable/as-of-system-time>, [Online; accessed 21-June-2024].
- [21] JUnit, Junit 5 user guide, 2024. URL: <https://junit.org/junit5/docs/current/user-guide/>, [Online; accessed 25-May-2024].
- [22] Oracle Corporation, Java jdbc api, 2024. URL: <https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/>, [Online; accessed 25-May-2024].
- [23] Exasol AG, Exasol documentation: Pyexasol, 2024. URL: https://docs.exasol.com/db/latest/connect_exasol/drivers/python/pyexasol.htm, [Online; accessed 25-May-2024].