Temporal Reasoning of LLMs in Relation to Text-to-SQL

Dušan Petković
Technical University of Applied Sciences
Rosenheim, 83024, Germany
ORCID: 0000-0002-5188-2811

ABSTRACT
Querying data stored by database systems has been one of the most important tasks of these systems. Hence, there were always efforts to generate such queries using text written in a natural language. In this article we use a part of the SQL Standard called SQL/OLAP and ChatGTP as a representative of Large Language Models (LLMs) to evaluate the process described above. Our evaluation shows that different subgroups of queries deliver significantly different results. There are two subgroups of queries, for which ChatGTP generates many erroneous queries. The first subgroup comprises queries related to non-standardized temporal features and predicates. In other words, the main property of the queries in this group is that they are not specified in the standard, but are supported by several RDBMSs. The second subgroup contains queries in relation to standardized temporal features and predicates not implemented in the chosen RDBMS. Our evaluation shows that for the subgroup of queries related to non-standardized temporal features, ChatGTP generates only one query correctly. If the generation process demands the use of standardized temporal features, which are not implemented in the DBMS, the error rate is 43% and 75%, respectively.

General Terms
Relational database systems

Keywords
Large Language Models, RDBMSs, Text-to-SQL, SQL/OLAP

Artifact Availability:
The data and/or other artifacts have been made available at https://doi.org/10.6084/m9.figshare.32625528.

1. INTRODUCTION
Relational database systems exist already for more than sixty years. During that time, several database languages have been introduced and applied. In spite of it, the only language that has been supported by all database systems is SQL (Structured Query Language). The importance of SQL shows the fact that it is the only standardized database language.
The main property of SQL is that its foundation is based upon a strictly mathematical model. In other words, relational algebra is a background for SQL Therefore, the use of this language can be difficult for database users without mathematical background.
Hence, from the beginning of existence of SQL there existed efforts to support SQL users with systems that support the generation of queries from the given text. These systems, called Text-to-SQL systems, accept an input text and the system's task is to generate the corresponding SQL query. Text-to-SQL systems can be divided into four groups: rule-based, deep-learning based, PLM-based (pre-trained learning models) and LLM-based (large language models).

The use of rules is typical for early systems. The main property of these systems is that rules and assessment criterions were applied to generate SQL queries from the given text. The main advantage of them is their deterministic nature: It allows correct generation of simple queries. On the other hand, these methods usually generate faulty queries, in case of complex questions.

With the emergence of deep neural networks, deep-learning based models were used to support the Text-to-SQL task. The existence of a huge amount of trained data allows the systems to transform complex questions into correct SQL queries. The main disadvantage of these models is when amount of training data is small. (In case of Text-to-SQL, this is especially true for recursive queries and queries with the window function.) Additionally, they sometimes generate faulty or incomplete queries due to limited understanding of structures inside the query. (The typical example for this are nested queries, which can comprise many nesting levels.)

The next step in development of Text-to-SQL systems has been the introduction of PLM-based systems. These systems use the large amount of knowledge in relation to linguistic, as well deep understanding of language semantics to apply it during the training process of data. Fine-tuning on such systems in relation to the Text-to-SQL task allows generation of correct SQL queries.

Large language models (LLMs) are the newest group of Text-to-SQL systems. Their capability to generate meaningful output for a lot of different areas make them the best solution for Text-to-SQL. In other words, the generation of SQL queries using the given text can be executed significantly better and less faulty than with other models.

During the lifetime of real-world entities, some of their properties change over time. For such attributes, it is essential to consider their temporal aspects. Hence, if applications process various temporal evolutions of values, the database system must support the processing of all attribute values that are valid at different points in time. These attributes are called temporal attributes.

There are several application scenarios where temporal data play an important role:

– For certain types of contracts (e.g., various types of insurance), a specific amount of money is agreed upon for a fixed period of time. This amount changes over time.

– In forecasting methods, existing patterns in data are uncovered by examining past values, and conclusions about future behavior are drawn from them.

– A retailer must ensure that only one special price applies to product P for each time period.

The rest of the paper is structured as follows. Section 2 briefly describes temporal features, specified in the SQL Standard. Section 3 explains, why MariaDB has been used for the evaluation. Section 4 is the main part of the paper: It shows the results of evaluation of two groups of SQL queries: for valid and system time. The results are evaluated from different angles.

Concerning Text-to-SQL systems, we use ChatGTP as a representative. (The phrases “ChatGTP” and “the system” will be used interchangeable in the rest of this article.) ChatGTP has been used for the following reasons:

- ChatGTP is trained especially for the generation of SQL queries;
- ChatGTP outperforms other systems significantly in relation to performance.

Our experiments are conducted with the use of ChatGPT-5.

1.1 Related Work

The main source in relation to SQL/Temporal is Part 2 of the SQL Standard [1]. The detailed description of all temporal features and predicates is given in [2, 3]. Melton in [4] gives, among the desription of other topics, deep analysis of temporal features. The discussion of such features related specifically to data warehouses can be found in [5].

There are just a few articles in relation to empirical study of Text-to-SQL. In [6], Nascimento et al empirically evaluate different large language models in relation to Text-to-SQL for complex databases with large schemas. The article [7] evaluates the capabilities of ChatGTP, as a representative of LLMs, to generate queries for a strictly specialized and standardized area of SQL, called SQL/OLAP. This evaluation shows that for the subgroup of examined examples with invalid cases, only one generated query was correct. Liu et al in [8] provide an empirical discussion of abilities of LLMs in relation to performance. They show that ChatGTP outperforms other LLM systems significantly.

On the other hand, there is a lot of articles dealing with Text-to-SQL benchmarks. The most important benchmarks are Spider[9] and BIRD [10]. The first one is a cross-domain benchmark for Text-to-SQL with huge number of queries (more than 10,000). BIRD is a Text-to-SQL benchmark, which comprises even more queries (ca. 12,500 queries).

2. INTRODUCTION OF TEMPORAL FEATURES IN SQL STANDARD

The part of the SQL Standard concerning temporal features is called SQL/Temporal. The main concepts in SQL/Temporal are time dimensions. There are two such dimensions: valid time and system time. A data model that supports only valid time is called valid-time model and one that supports only system time is called system-time model. Bitemporal model supports both of them.

2.1 Valid Time

Valid time is the time period during which table’s rows store information, which is true for purposes of real-world applications. For this reason, information is independent of time and can concern past, present and future data values.

In SQL/Temporal, valid time support is provided by application-time period tables. These tables are used to track when certain business conditions are valid.. An example of such a condition would be the increasing price of a product.

When creating application-time period tables, two additional columns must be defined. The first one stores the beginning values of the time period, while the second contains the ending values. To specify these two columns, the PERIOD FOR clause of the CREATE TABLE statement is used. In addition, the clause implicitly defines that the start date is less than the end date.

The two columns mentioned above are explicit columns, meaning that they are defined by the user. The columns must be specified with the NOT NULL property and their data type is one of standard date data types, i.e., TIMESTAMP or DATE. The time interval, specified with the starting and ending values is half-open, meaning that it contains the start value, but not the end one.

Example 1 shows the creation of an application-time period table.

Example 1

```
CREATE TABLE dept(dept_id VARCHAR(30),budget DEC(7,2),
   start1 DATE NOT NULL, end1 DATE NOT NULL,
    PERIOD FOR dept_time (start1, end1));
```

2.2 System Time

System time concerns the time when an event is represented as stored data in the corresponding database. In contrast to valid time, timestamps of system time are specified by the system. Therefore, the history of all temporal data concerning system time can be built in relation to the current as well as past time. (The consequence is that only current values may be updated/deleted.)

In SQL/Temporal, system time support is provided by system-versioned tables. System-versioned tables contain system times that are stored every time changes are made to the table content. For example, if data from a person's insurance policy need to be stored in such a way that all changes over the policy period can be seen, the tables mentioned above are used.

Just like application-time period tables, system-versioned tables contain two additional columns that specify the start and end of the time period. Unlike the corresponding columns in application-time period tables, the values for the system time period columns are inserted by the system.

The names of the columns, indicating the start and end of the time period, are specified by users, but contain additional clauses. The GENERATED ALWAYS AS ROW START clause specifies the column with start values, and the GENERATED ALWAYS AS ROW END clause specifies the column with end values. Example 2 shows the creation of a system-versioned table.

Example 2

```
CREATE TABLE insurance_dept
( id VARCHAR(30) NOT NULL, dept_id VARCHAR(30),
      name CHAR(10),
  sys_start TIMESTAMP(6)
      GENERATED ALWAYS  AS ROW START,
  sys_end TIMESTAMP(6)
      GENERATED ALWAYS AS ROW END,
  PERIOD FOR SYSTEM_TIME(sys_start, sys_end)
   ) WITH SYSTEM VERSIONING;
```

The sys_start column in the insurance_dept table specifies the beginning and sys_end the end of the system time period. The corresponding PERIOD clause with the SYSTEM_TIME specification contains the names of the two columns that specify the beginning (the first one) and the end (the last one) of the system time period. (This clause implies the rule that sys_start` < sys_end.). The most important property of system-versioned tables is that the old versions of the tuples are preserved. (Tuples whose time periods contain the current system time are called "current system rows," and the others are called "historical system rows.") Only the current system rows are modified during change operations.

3. SQL/TEMPORAL SUPPORT IN RDBMSs

To choose the database system, which provides the best support for SQL/Temporal, we evaluated the following systems:

- Oracle;
- SQL Server;
- IBM db2;
- PostgreSQL;
- MariaDB.

The evaluation of Oracle was easy: This DBMS has not implemented standardized temporal features at all. SQL Server 2019 [11] has almost full support for system time as it is described in SQL/Temporal, but no support for valid time. The same is true for IBM db2. As can be seen in [12], this database system supports, with slight changes in the CREATE TABLE statement, system time, but the implementation of valid time is significantly different in relation to the specification in SQL/Temporal.

In PostgreSQL community, there are claims that PostgreSQL supports temporal part of SQL:2011 standard [13], but this is not true. First, there is no support for valid time at all. Second, the syntax of the CREATE TABLE statement for system-versioned tables differs significantly from the specification in SQL/Temporal. As can be seen in [14], MariaDB, since Version 10.3, supports almost all features specified in SQL/Temporal. For this reason, we use MariaDB for our evaluation.

After the evaluation described above, we separated all queries in two groups:

- Queries in relation to valid time,;
- Queries in relation to system time.

The sample of the queries related to valid and system time contains 35 examples each. The text for all queries mentioned above, together with the generated queries from ChatGTP and corresponding results with MariaDB have been made available at https://doi.org/10.6084/m9.figshare.32625528.

4. RESULTS OF EVALUATION

Our evaluation comprises several steps. We started with the „general“ evaluation of all 70 queries in relation to valid and system time. As can be seen from Figure 1, the percentage of erroneous queries in case of queries related to valid time was significantly higher (40%) than the percentage of erroneous queries related to system time (ca. 25%). The reason for this mismatch is that almost all temporal predicates, specified in the standard as well as in the Allen's paper [15] are not supported by MariaDB, but the system generates the query explicitly using such an operator.

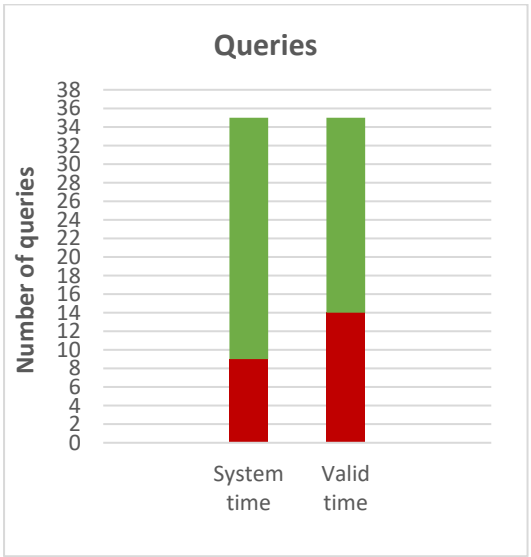

Figure 1 General Evaluation of all queries

According to this knowledge, in the following steps, we narrowed down the evaluation. In the second step, we consider the queries related to non-standardized temporal features and predicates. In other words, the main property of the queries in this group is that they are not specified in SQL/Temporal, but are supported by several RDBMSs.

After that, in the third step, we evaluated another subgroup, which contains queries in relation to standardized temporal features and operators. Finally, we assessed separately the group of queries in relation to metadata. (The results of evaluation of these subgroups are described in the following three subsections.)

4.1 Evaluation of Non-standardized Temporal Features

The non-standardized queries are all queries, which are not specified in SQL/Temporal but are implemented in one or several RDBMSs. (Following examples are related to non-standardized temporal features. For system time: #9, #12, #13, #17, #18, #19 and #27. For the valid time: #20 till #25 and #11.) As can be seen in Figure 2, all such queries in case of valid time are faulty, while in case of system time all but one of generated queries are wrong. This problem arises, because the system knows that such temporal features are implemented in RDBMSs, but does not check whether they are implemented in MariaDB.

The following two queries are typical for such behavior of the system. The first example is related to valid time, while the second one concerns system time. The text in Example 3 explicitly addresses the LAST temporal predicate, which is one of Allen's temporal operators [15] and is implemented in a few RDBMSs, but not in MariaDB. This temporal predicate returns the last value of the time period argument (that is, the ending bound minus one granule of the element type of the argument).

Example 3 (corresponds to #20, see file with valid time examples)

Use the MariaDB database system. Using the SELECT statement, display the last value of all the values of the emp_period period. Use the LAST temporal predicate.

(MariaDB displays the error message: "Error: The LAST function does not exist"".)

ChatGTP generates the following erroneous query:

```
SELECT emp_no, emp_dept, emp_period
    FROM Employee WHERE emp_period LAST;
```

Example 4 uses CONTAINED IN, which is a non-standardized option of the FOR SYSTEM TIME clause. (This option returns all historical system rows that were opened and closed within the specified time interval.)

On the other hand, for the same clause, SQL/Temporal supports several other options:

- FOR SYSTEM TIME AS OF <date>
- FOR SYSTEM TIME FROM <start_date> TO <end_date>
- FOR SYSTEM TIME BETWEEN <start> AND <end>

Example 4 (corresponds to #9, see file with system time examples)

Use the MariaDB database system. Using the SELECT statement and the FOR SYSTEM_TIME CONTAINED IN option, display all historical system rows with the values that were opened and closed within the following time interval:
('2025 -05-19 06:41:07.2902041' ,
'2025-05-20 06:41:07.2902041`)

(MariaDB displays the error message: "MariaDB does not support CONTAINED IN")

ChatGTP generates the following faulty query.

```
SELECT * FROM dept_system_temp
FOR SYSTEM_TIME CONTAINED IN (
  TIMESTAMP '2025-05-19 06:41:07.290204',
   TIMESTAMP '2025-05-20 06:41:07.290204' );
```

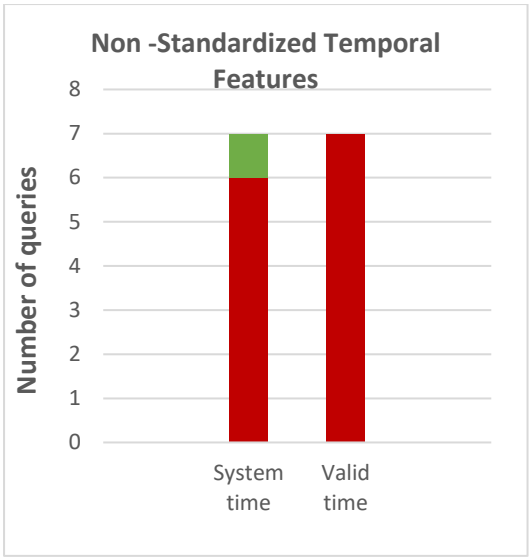

Figure 2. Non-standardized features

4.2 Evaluation of Standardized Temporal Features

The standardized temporal features are all features specified in SQL/Temporal. As can be seen in Figure 3, the percentage of erroneous queries in relation to system time is ca. 43%, while the percentage of erroneous ones in relation to valid time is 75%. The explanation for the high percentage of generation of faulty queries for system time, as well as especially for valid time is that ChatGTP knows that the features are standardized, but does not check, whether they are implemented in MariaDB. The following two queries are typical for such behavior of the system. The first example is related to valid time, while the second one concerns system time.

Text in Example 5 requires the explicit use of the CONTAINS predicate. (The other standardized predicates are: OVERLAPS, EQUALS, PRECEDES, SUCCEEDS, IMMEDIATELY PRECEDES and IMMEDIATELY SUCCEEDS [2].)

Example 5 (corresponds to #12 in the valid time examples)

Use the MariaDB database system and the dept_table table. Using the SELECT statement, find the employee numbers and the corresponding department numbers, for all employees, which have worked in the time period from '2025-07-01' to '2025-08-01'. Use the CONTAINS predicate.

*(*MariaDB does not support the CONTAINS predicate.)

ChatGTP generates the following query, which is wrong,

```
SELECT emp_no, emp_dept FROM Employee
    WHERE CONTAINS(emp_period, DATE '2025-07-01', DATE '2025-08-01');
```

Text in Example 6 contains two parts. In the first part, the new system-versioned table called t1 has been created and two rows are inserted in the table. In the second part, the same table has been deleted with the TRUNCATE TABLE statement. (The TRUNCATE TABLE statement is an alternative to the DELETE statement, and is used to remove all rows from a table quickly and efficiently, without logging individual row deletions.) It is obvious that this statement cannot be used for system-versioned tables, because in such tables historical system rows must be secured.

Example 6 (corresponds to #14, see file with system time examples)

#1 Use the MariaDB database system. Using SQL, create the system-versioned table called t1. The table has one column, t1, of the INT data type. Additionally, insert two rows into the t1 table. The values of the t1 column are 1 and 2, respectively.

#2 Use the MariaDB system. Use the TRUNCATE TABLE statement to delete all rows from the t1 table.

ChatGTP generates the t1 table and inserts two rows correctly, but use the TRUNCATE TABLE statement in the second step:

```
CREATE TABLE t1 (t1 INT) WITH SYSTEM VERSIONING;
INSERT INTO t1 VALUES (1); INSERT INTO t1 VALUES (2);
TRUNCATE TABLE t1;
```

The execution of the last statement above with MariaDB provides the following error message: “System-versioned tables do not support TRUNCATE TABLE”.

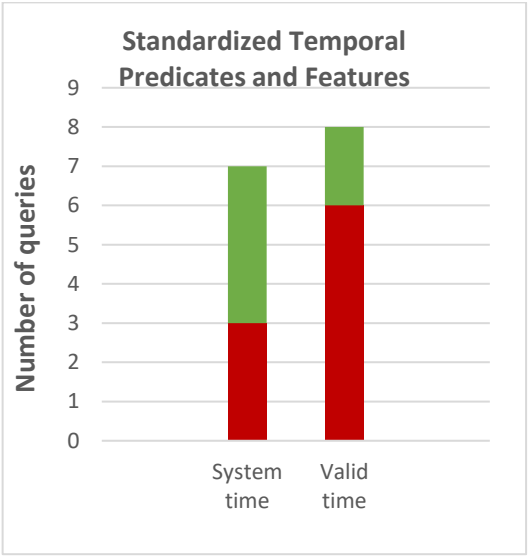

Figure 3. Standardized features

4.3 Evaluation of Queries Related to Metadata

As can be seen from Figure 4, all queries in relation to metadata has been generated correctly. (Following examples are related to metadata. For system time: #21, #22, #32, #33, #34 and #35. For valid time: #27 till #32 and #34.) The reason for this is that there is just a few system tables in relation to temporal data. In other words, all queries in relation to the system catalog of MariaDB are simple ones, because they are usually related just to one table, and in case of join operations, such queries connect only two tables.

MariaDB’s information schema tables provide metadata about all tables in the databases. (These tables are called system tables.) System tables concerning temporal data are divided in two groups. The first one is related to valid time, while the second one is related to system time. The most important system tables concerning valid time are information_schema.key_period_usage and information_schema.periods. The first one provides information about valid time periods, in particular, while the second one provides the same information, in general.

The most important system table concerning system time is information_schema. partitions. This table provides metadata about table partitions, including partition methods and data distribution.

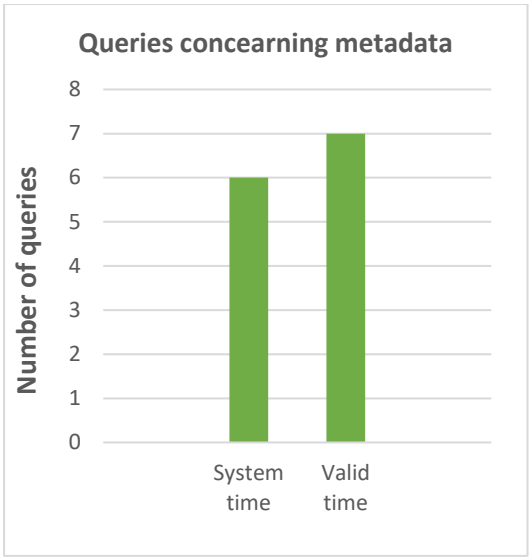

Figure 4. Queries concerning metadata

5. CONCLUSIONS AND FUTURE WORK

The focus of this article is to evaluate the process of Text-to-SQL using ChatGTP as the representative of LLMs,. We use the part of SQL called SQL/Temporal to assess this task, and MariaDB as a representative of RDBMSs The temporal features specified in SQL/Temporal are strictly defined, and this fact allows us to use the SQL standard as a source for definition and evaluation of all obtained results. Our results show that ChatGTP, for three different subgroups of queries, delivers significantly different outcome.

For the first subgroup of non-standardized temporal features, all queries related to valid time are faulty, while in case of system time all but one of generated queries are wrong. This problem arises, because ChatGTP "knows" that such temporal features are implemented in some RDBMSs, but does not check whether they are implemented in MariaDB. The second subgroup are standardized temporal features. Almost half of all generated queries are faulty, in relation to system time, while the result of evaluation for valid time is even worse: only fourth of queries has been generated correctly. The explanation for the high percentage of generation of faulty queries for system time, as well as especially for valid time is that ChatGTP makes only the first step, in the process, where *two steps* are necessary (These steps are to check furst in the SQL Standard whether the temporal feature is standardized, and after that, to verify whether this feature is implemented in MariaDB.) On the other hand, for the subgroup of queries concerning metadata, our evaluation shows that queries related to only one table or queries concerning two tables connected by one join operation have been generated with 100% success by ChatGTP. In our future work we will further consider to explore capabilities of LLMs for Text-to-SQL in relation to SQL/Spatial. This is another enclosed SQL subarea, which is specified in the SQL Standard.

6. REFERENCES

[1] ISO/IEC 9075-2:2011, Information technology –Database languages–SQL–Part 2:Foundation

[2] Kulkarni, K.,Michels, J., Temporal Features in SQL:2011, SIGMOD Records,Vol. 41(3), 2012

[3] Petković, D., Was lange währt, wird endlich gut: Temporale Daten im SQL-Standard (in German), Datenbank-Spektrum 13(2), 2013, https://doi.org/10.1007/s13222-013-0120-3

[4] Melton, J., SQL: 1999 – Advanced SQL:1999, Understanding Object-relational and Other Advanced Features, Morgan-Kaufman, 2002.

[5] Sannik, G., Daniels, F., Enabling the Temporal Data Warehouse, 2011, https://cs.ulb.ac.be/public/_media/teaching/infoh415/teradata_enabling_temporal.pdf

[6] E. R. Nascimento and G. García, 2024. *LLM-Based Text-to-SQL for Real-World Databases, SN Computer Science. https://doi.org/10.1007/s42979-025-03662-6.*

[7] Petković, D., Applying Text-to-SQL to SQL/OLAP Using LLMs, (MLNLP 2025), Hangzhou, China, November 2025, DOI: 10.1109/MLNLP66797.2025.11389090

[8] A. Liu and X. Hu.. A comprehensive evaluation of ChatGTP's zero-shot Text2SQL Capability,. https://doi.org/10.48550/arXiv.2303.13547.

[9] T. Yu and R. Zhang and K. Yang. 2018. *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL, In EMNLP, 2018*.

[10] J. Li and B. Hui., Can LLM Already Serve as A Database Interface? A Big Bench for Large-Scale Database Grounded Text-to-SQLs, *2023, https://doi.org/10.48550/ARXIV.2305.03111*

[11] Petković, D., SQL Server 2019 A Beginner's Guide, McGraw Hill, 978-1-260-45887-3, 2019.

[12] Saracco, C., Nicola, M., Gandhi, L, A matter of time: Temporal data management in DB2, www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/dm-1204db2temporaldatapdf.

[13] Clark, D. Historical records with PostgreSQL, temporal tables and SQL:2011, https://clarkdave.net/2015/02/historical-records-with-postgresql-and-temporal-tables-and-sql-2011/.

[14] MariaDB: Temporal Tables, https://mariadb.com/docs/server/reference/sql-structure/temporal-tables

[15] Allen J.F., Maintaining knowledge about temporal intervals. Com. ACM Vol. 26(11), 1983.