# Temporal Query Processing in Teradata

Mohammed Al-Kateb
Teradata Labs
100 N. Sepulveda Blvd.
El Segundo, CA 90245
mohammed.al-kateb
@teradata.com

Ahmad Ghazal
Teradata Labs
100 N. Sepulveda Blvd.
El Segundo, CA 90245
ahmad.ghazal
@teradata.com

Alain Crolotte
Teradata Labs
100 N. Sepulveda Blvd.
El Segundo, CA 90245
alain.crolotte
@teradata.com

Ramesh Bhashyam
Teradata Labs
Queens Plaza
Hyderabad, 500 003, India
bhashyam.ramesh
@teradata.com

Jaiprakash Chimanchode
Teradata Labs
Queens Plaza
Hyderabad, 500 003, India
jaiprakash.c
@teradata.com

Sai Pavan Pakala
Teradata Labs
Queens Plaza
Hyderabad, 500 003, India
sai.pakala
@teradata.com

## ABSTRACT

The importance of temporal data management is evident by the temporal features recently released in major commercial database systems. In Teradata, the temporal feature is based on the TSQL2 specification. In this paper, we present Teradata's implementation approach for temporal query processing. There are two common approaches to support temporal query processing in a database engine. One is through functional query rewrites to convert a temporal query to a semantically-equivalent non-temporal counterpart, mostly by adding time-based constraints. The other is a native support that implements temporal database operations such as scans and joins directly in the DBMS internals. These approaches have competing pros and cons. The rewrite approach is generally simpler to implement. But it adds a structural complexity to original query, which can pose a potential challenge to query optimizer and cause it to generate sub-optimal plans. A native support is expected to perform better. But it usually involves a higher cost of implementation, maintenance, and extension. We discuss *why* and describe *how* Teradata adopted the rewrite approach. In addition, we present an evaluation of our approach through a performance study conducted on a variation of the TPC-H benchmark with temporal tables and queries.

## Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*

## General Terms

Temporal

## 1. INTRODUCTION

Time is an important attribute of each and every real-world application. This importance made it necessary for a broad spectrum of database applications to be able to perform diverse and sophisticated temporal data analytics. In response to this necessity, ANSI SQL:2011 just came out with new SQL constructs to support some temporal functionality [9]. Commercial database vendors also acknowledged this indispensable feature as we recently witnessed a big boost of temporal support in major commercial database management systems (e.g., IBM DB2 [8], Oracle [12], and Teradata [15]). Teradata's temporal feature - made available in release 13.10 and further enhanced in subsequent releases - is based on the TSQL2 specification [13]. This feature comes with an ample scope of temporal elements, including temporal tables definitions (e.g., valid-time and transaction-time tables), temporal qualifiers (e.g., sequenced and non-sequenced), and others (e.g., temporal constraints and comparisons). The value of Teradata's temporal feature is evident as already being adopted by a class of data-intensive industries such as top retailers and leading insurers.

In this paper, we present the approach taken by Teradata to implement temporal query processing. There are two alternatives to consider for implementing temporal queries. The first is a functional rewrite approach to express the semantics of temporal queries in conventional (non-temporal) SQL. This approach is typically implemented by adding a pre-optimization phase to perform the functional rewrites, which can be in the form of adding new constraints and/or using derived tables. The second is through a native temporal support in the underlying database engine. This approach implements temporal database operations such as scans, projections, and joins directly in the DBMS internals.

To illustrate the difference between the rewrite and the native approaches, consider an example portraying a scenario for insurance policy coverage. Table 1 shows that policy "123-456-789" was initially for partial coverage. This is represented by a single record in Table 1 holding *current* (i.e., active) data on this policy. Table 2 shows that on "01-01-2012", the policy was upgraded to full coverage and this has been effective to date. Upon the upgrade, the former record became *history* and the new record has become *current.*

**Table 1: Policy: Current Data - no History**

| Policy_Number | Policy_Coverage | Coverage_Period |
|---|---|---|
| 123-456-789 | Partial | (01-01-2010, UNTIL_CHANGED) |

Assume that Customer Service wants to know the *current* coverage for policy "123-456-789". This business question can be expressed in a query as follows:

```
CURRENT VALIDTIME
SELECT Policy_Coverage
FROM Policy
WHERE Policy_Number = '123-456-789';
```

In this syntax, CURRENT VALIDTIME directs the query to go after rows whose time *validity* (i.e, Coverage Period) is current (i.e., present and active). This declarative syntax is part of TSQL2 [13] implemented in Teradata temporal feature. More details on this syntax are in Section 2.

A rewrite approach needs to transform the above temporal query to a semantically-equivalent non-temporal one. The non-temporal query is produced by attaching time-based predicates to the WHERE clause to filter out all the rows that are not *current* (i.e., to discard all inactive records from the result set and retain the remaining ones). For the above temporal query, its non-temporal equivalent is as follows (assuming current time is '10-11-2012' with closed-open semantics for the period representation):

```
SELECT Policy_Coverage
FROM Policy
WHERE BEGIN(Coverage_Period) <= DATE '10-11-2012'
AND   END(Coverage_Period)   >  DATE '10-11-2012'
AND   Policy_Number          =  '123-456-789';
```

An English-like execution plan description for the query after transformation is as follows[1]:

```
Step1: Do a RETRIEVE step from Policy table with a residual
condition of BEGIN(Coverage_Period)<= DATE '2012-10-11' AND
END(Coverage_Period) > DATE '2012-10-11' into Spool 1.
Step 2: Return the contents of Spool 1.
```

The plan with the rewrite approach actually deals with the temporal table as a regular table. Consequently, it plans for a regular RETRIEVE step to scan the table. But to reflect the temporal semantics of the CURRENT keyword, the query is augmented with a residual condition to filter out all the rows that do not overlap with current time.

A native implementation, however, would treat the temporal table as a different database object and produce an execution plan accordingly. That is, the underlying DBMS would need to recognize the temporal table, distinguish between history and current data, and provide efficient plans and access paths to either or both. Naturally, this leads to a new retrieve operator to carry out this execution. Therefore, an English-like execution plan following a native implementation would assume a TEMPORAL RETRIEVE step to scan the table and return CURRENT rows:

```
Step1: Do a TEMPORAL RETRIEVE step from Policy table for
CURRENT rows into Spool 1.
Step 2: Return the contents of Spool 1.
```

The tradeoff between these approaches is twofold. With regard to implementation, the rewrite approach is generally easier and more predictable, while a native support usually brings about a higher cost of implementation, maintenance, and extension. In the aforementioned temporal query example, the rewrite approach simply adds temporal constraints to the original query and the DBMS treats

[1]Throughout this paper, we present English-like execution plans as simplified version of Teradata EXPLAIN.

**Table 2: Policy: Current and History Data**

| Policy_Number | Policy_Coverage | Coverage_Period |
|---|---|---|
| 123-456-789 | Partial | (01-01-2010, 01-01-2012) |
| 123-456-789 | Full | (01-01-2012, UNTIL_CHANGED) |

(i.e., plans, optimizes, and executes) it as a regular query therein. In contrast, a native implementation needs to introduce a new temporal scan operator with new directives to go after either or both history and current rows in an efficient and optimized way. With regard to performance, the rewrite approach is likely to bring a challenge to the query optimizer and cause it to lose optimization chances or produce sub-optimal plans due to the structural complexity it contributes to query processing. The native implementation approach can avoid such potential negative performance issues as it is typically expected to deliver better performance by directly implementing temporal operations in the DBMS internals [10].

For commercial database vendors like Teradata, targeting industrial applications coupled with complex queries and heavy workloads, such a tradeoff between implementation feasibility and performance impact becomes a real challenge. To meet this challenge, Teradata made the decision to follow the functional rewrite approach and tightly integrate it with its existing optimizer rewrite rule engine (ORRE). The primary principle of this decision is to develop new functional rewrites to handle temporal semantics (e.g., temporal qualifiers and joins) and to leverage existing Teradata's optimization rewrites (e.g., predicate move-around and view folding) to optimize the performance [6, 7]. Implementation feasibility is then realized via the simplicity and extensibility of the rewrite approach and optimized performance is achieved through Teradata's robust and intelligent optimizer.

Our contributions in this paper are as follows. First, we discuss *why* Teradata adopted the rewrite approach. Specifically, we analyze in details the technical challenges of implementing temporal feature natively with regard to main DBMS components. Second, we describe *how* Teradata performs temporal rewrites. Particularly, we explain how functional temporal rewrites are actually done for different query clauses and how existing optimization rewrites are leveraged to address potential structural complexity in queries resulting from functional rewrites. Finally, we present an evaluation of the performance of our approach through an empirical study conducted on a variation of the TPC-H benchmark with temporal tables and queries [2]. The results of experiments show that temporal queries with our rewrite approach perform comparably to their non-temporal counterparts running on tables with the same row count and size.

The rest of the paper is organized as follows. Section 2 introduces an overall overview of Teradata temporal feature. Section 3 discusses the implementation approach. Section 4 presents the performance evaluation. Section 5 reviews related work. Finally, Section 6 concludes the paper and points to avenues for future work.

## 2. TERADATA'S TEMPORAL FEATURE

In this section, we give a general overview of the temporal feature in Teradata.

### 2.1 Temporal Data Model

Temporal data model in Teradata associates a temporal dimension (valid time and/or transaction time) to the entire row. Valid time is the time period during which the associated attribute values are deemed genuine in reality. Transaction time is the time period during which attribute values are actually stored in the database. Valid time and transaction time are stored as a PERIOD data type in a single column if that column is defined using the AS VALIDTIME construct and the AS TRANSACTIONTIME construct, respectively. A temporal table can contain only a valid time column (valid-time table), only a transaction time column (transaction-time table), or both (bi-temporal table).

In a valid-time table, a row can be *history* if its valid time ends before current time, *current* if its valid time overlaps current time, or *future* if its valid time begins after current time. Current and future rows can be assigned an open-ended value of UNTIL_CHANGED for the end of their valid time column in case such a value is not known beforehand. In a transaction-time table, a row can be either *closed* if it has a transaction time that ends before current time, or *open* if has a transaction time that overlaps current time. An open row is automatically assigned an open-ended value of UNTIL_CLOSED for the end of its transaction time value.

## 2.2 Temporal Qualifiers

Temporal qualifiers, proposed by Snodgrass [13, 14], allow for time-slicing queries over temporal tables. These qualifiers are in the form of *current*, *sequenced*, and *non-sequenced* modes. The current mode (defined using CURRENT keyword) applies to current rows. The sequenced mode (defined using SEQUENCED keyword) pertains to rows whose valid time overlaps a time period given in the query. In the absence of a given period, it applies to history, current, and future rows. The non-sequenced mode (defined using NONSEQUENCED keyword) discards the temporal semantics and handles rows in a way similar to regular query processing. CURRENT queries do not return temporal information in the result, while SEQUENCED queries do. NONSEQUENCED queries return user specified columns including time period columns but without any temporal meaning.

## 2.3 Temporal Comparisons

Temporal comparisons are done using temporal predicates and functions. Temporal predicates, such as *overlaps* and *contains*, are constructs for comparing two time periods. Temporal functions, such as *begin* and *end*, apply to a time period and return a value extracted from that period.

## 2.4 Temporal Constraints

Temporal constraints can be enforced at the level of each table and between tables. Table-level (e.g., check and unique) constraints are expressed by associating a temporal qualifier and time dimension with the defined constraint. For a valid-time table, a constraint can be defined as CURRENT, SEQUENCED, or NONSEQUENCED. A transaction-time table only allows a constraint to be CURRENT. Constraints on bi-temporal tables apply to rows that qualify in the valid-time dimension and open in the transaction time dimension.

Temporal constrains across tables represent temporal referential integrity. Similar to table-level constraints, referential integrity becomes temporal when defined with respect to time.Temporal referential integrity can be flexibly defined with the child table being either temporal or regular table.

## 2.5 Other Temporal Elements

Temporal tables can be physically partitioned. Horizontally (i.e., row) partitioning is particularly useful if a table is partitioned in order to separate history and current rows, which can improve the performance of various processing modes. Vertical (i.e., column) partitioning over temporal tables can also improve the performance of temporal queries by directly projecting relevant columns without the need to access the entire set of columns. Join indexes can be also defined over temporal tables to enhance the performance of temporal queries by directly accessing a relatively smaller portion of the data without the need to physically access base tables. Other key elements of Teradata temporal features include, for example, archiving and loading utilities.

## 3. IMPLEMENTATION APPROACH

In this section, we present our contributions by discussing why and describe how Teradata implemented its temporal feature following the rewrite approach. First, we analyze the technical challenges of a native implementation and contrast that to the rewrite approach. Second, we describe how temporal functional rewrites are performed. Third, we explain how existing optimization rewrites are used to address complexity of queries resulting from functional rewrites. Finally, we conclude with a discussion on using the rewrite approach further to support other temporal-specific operators.

## 3.1 Native vs. Rewrite

A native temporal implementation is mainly about supporting temporal tables as a new type of objects. This involves different storage structure to accommodate current and history portions of the data. These new tables demand for changes to all SQL execution code from joins and aggregations to window function processing. For example, existing hash join execution code needs to be extended for SEQUENCED and CURRENT joins. Also, all aspects of query optimization need to be addressed for new temporal tables and their execution. Examples are as follows. Access path analysis and statistics need to work on new temporal tables. Query rewrite and join planning need to handle new SEQUENCED and CURRENT queries. Join planning search space should be expanded with new temporal operations. Calibration is needed for new temporal coefficients and low-level cost formulas should be developed accordingly. Beyond optimizer and execution engine, a native implementation impacts code quality and extensibility negatively with deeper changes and certain level of code duplication in the basic DBMS code areas. This can come with a higher risk on code quality and make extensibility harder and costly.

In contrast, the rewrite approach has no impact on execution and little impact on the optimizer and main code path. It is also less risky since it is applied as a separate component just before the query optimizer. The DBMS code can be generally extended easily because of early conversion to non-temporal qualifiers and constraints. The main issue of this approach is the added complexity of query structure. Section 3.3 shows an example of such complexity and how it is handled by the existing rewrite optimizations in Teradata.

## 3.2 The Rewrite Approach

As Figure 1 shows, functional and optimization rewrites jointly represent the core infrastructure for temporal query processing in Teradata. We developed new functional rewrites to manage temporal semantics (e.g., temporal joins) and leveraged existing optimization rewrites (e.g., predicate move-around and view folding) to optimize the performance.

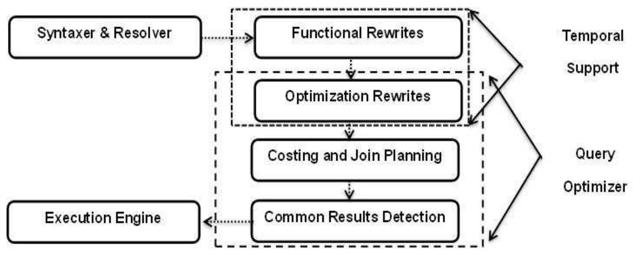

**Figure 1: Temporal Support in Teradata**

Functional rewrites for temporal support is essentially about the consequences of temporal qualifiers on: 1) projection list and 2) selection and join conditions. Projection concerns the inclusion or exclusion of time dimension from the SELECT clause resulting from the rewrites. Selection and join conditions pertain to the WHERE and the ON clauses. We explain our functional query rewrites mechanism for these two aspects with examples using the following temporal tables definitions and assuming current date is "10-11-2012":

```
Table1 (i1 int, c1 char(1), d1 double, f1 float,
        validity1 PERIOD(DATE) AS VALIDTIME);
Table2 (i2 int, c2 char(1), d2 double, f2 float,
        validity2 PERIOD(DATE) AS VALIDTIME);
```

### 3.2.1 *Projection list*

The central issue with projection list rewrites is about the resolution of "*". This particular issue has been a point for discussion for different temporal SQL proposals [9]. In Teradata's temporal implementation, the presence of "*" is resolved with regard to temporal qualifier specified in the query. For CURRENT queries, "SELECT *" is transformed to the list of all columns excluding valid or transaction time column since these queries are interested in recent data with no particular emphasis on the associated time periods. The following query is an example with CURRENT qualifier:

$\mathcal{TQ}$ 1.
```
CURRENT VALIDTIME
SELECT *
FROM Table1;
```

which is transformed to exclude valid time from query output (Note the condition added to the WHERE clause for the semantics of CURRENT. Details are in section 3.2.2):

$\mathcal{Q}$ 1.
```
SELECT i1, c1, d1, f1
FROM Table1
WHERE BEGIN(Table1.validity1) <= '10-11-2012'
AND   END  (Table1.validity1) >  '10-11-2012';
```

For SEQUENCED queries, "SELECT *" projects all columns with explicit VALIDTIME or TRANSACTIONTIME columns. Because these queries look for the history and current data, it becomes necessary to include the time dimension in the result. The following is an example of a SEQUENCED query:

$\mathcal{TQ}$ 2.
```
SEQUENCED VALIDTIME
SELECT *
FROM Table1;
```

which is transformed to include the valid time dimension as a VALIDTIME column:

$\mathcal{Q}$ 2.
```
SELECT i1, c1, d1, f1, validity1 as VALIDTIME
FROM Table1;
```

For NONSEQUENCED queries, all columns including the valid or transaction time column are projected as regular columns because these queries discard any temporal semantics. The following is an example NONSEQUENCED query:

$\mathcal{TQ}$ 3.
```
NONSEQUENCED VALIDTIME
SELECT *
FROM Table1;
```

which is rewritten with all columns in the projection list without distinguishing valid time column from other columns:

$\mathcal{Q}$ 3.
```
SELECT i1, c1, d1, f1, validity1
FROM Table1;
```

A relevant issue for projection list rewrites concerns SEQUENCED VALIDTIME queries with ORDER BY clause. For these queries, if valid-time is not explicitly listed in the ORDER BY clause, it will be appended at the end of the clause. The rationale behind that is to further order the rows based on their time validity since these queries are interested in history (SEQUENCED) and they are explicitly projecting valid-time column (VALIDTIME).

### 3.2.2 *Selection and join conditions*

For single-table queries, with or without selection conditions, the WHERE clause needs to be adjusted to go after rows overlapping with the time period of interest. For CURRENT qualifier and when SEQUENCED qualifier specifies an explicit time period, this is achieved by appending time-based predicates to the WHERE clause to filter out the remaining rows. For NONSEQUENCED qualifier or when SEQUENCED qualifier has no explicit time period defined, query should run after all rows in the table with no changes to the WHERE clause. Recall $\mathcal{TQ}$ 1 and its corresponding $\mathcal{Q}$ 1 for an example with CURRENT qualifier.

For join conditions, semantically, temporal qualifiers need to be applied on temporal tables prior to joins. For inner joins, it is semantically equivalent to apply join conditions first and then filter out non qualifying rows. Hence, functional rewrites for inner joins convert temporal qualifiers to predicates appended to the join condition in either the ON or the WHERE clause. Consider, for example, the following temporal inner join query:

$\mathcal{TQ}$ 4.
```
CURRENT VALIDTIME
SELECT Table1.c1, Table2.c2
FROM Table1 INNER JOIN Table2
ON Table1.i1 = Table2.i2;
```

which is transformed to a non-temporal counterpart with predicates attached to the ON clause:

$\mathcal{Q}$ 4.
```
SELECT Table1.c1, Table2.c2
FROM Table1 INNER JOIN Table2
ON  Table1.i1 = Table2.i2
AND BEGIN(Table1.validity1) <= '10-11-2012'
AND END(Table1.validity1)   >  '10-11-2012'
AND BEGIN(Table2.validity2) <= '10-11-2012'
AND END(Table2.validity2)   >  '10-11-2012';
```

The following is an example with SEQUENCED qualifier:

$\mathcal{TQ}$ 5.
```
SEQUENCED VALIDTIME
SELECT Table1.c1, Table2.c2
FROM Table1 INNER JOIN Table2
ON Table1.i1 = Table2.i2;
```

transformed to a non-temporal counterpart as follows:

$\mathcal{Q}$ 5.
```
SELECT Table1.c1, Table2.c2,
Table1.validity1 P_INTERSECT Table2.validity2
FROM Table1 INNER JOIN Table2
ON  Table1.i1 = Table2.i2
AND BEGIN(Table1.validity1) < END(Table2.validity2)
AND END(Table1.validity1)   > BEGIN(Table2.validity2);
```

For outer joins, however, following the same rewrite mechanism leads to a semantically different query that can produce incorrect results. By appending temporal predicates to the ON clause, for example, the outer table's rows belonging to the history and those that are current but do not qualify under the original join condition will both erroneously be in the result of outer join query. To address this problem, we use derived tables. The idea is to propagate the temporal qualifier of the original outer join query to the definition of derived tables defined on outer and inner tables. This propagation filters out rows that do not satisfy the original temporal qualifier and retain the remaining rows eligible for join. Then, the outer join is applied on the derived tables. In the following temporal outer join query example:

$\mathcal{TQ}$ 6.

```
CURRENT VALIDTIME
SELECT Table1.c1, Table2.c2
FROM Table1 LEFT OUTER JOIN Table2
ON  Table1.i1 = Table2.i2
AND Table2.f2 < 10;
```

we can see it is being rewritten with derived outer tables on inner and outer tables. Time-based predicates reflecting the semantics of temporal qualifiers are added as conditions on the derived tables. Then, the outer join condition is applied:

$\mathcal{Q}$ 6.

```
SELECT t1.c1, t2.c2
From
(
SELECT *
FROM Table1
WHERE BEGIN(Table1.validity1) <= '10-11-2012'
AND   END(Table1.validity1)   >  '10-11-2012'
) t1
LEFT OUTER JOIN
(
SELECT *
FROM Table2
WHERE BEGIN(Table2.validity2) <= '10-11-2012'
AND   END(Table2.validity2)   >  '10-11-2012'
) t2
ON  t1.i1 = t2.i2
AND t2.f2 < 10;
```

## 3.3 Query Rewrite Optimizations

As discussed before, temporal functional rewrites may add structural complexity to the original query. For example, the rewritten query $\mathcal{Q}$ 6 is more complex than the original query $\mathcal{TQ}$ 6. The join in $\mathcal{Q}$ 6 is applied between two derived tables rather than two base tables. A straightforward execution of $\mathcal{Q}$ 6 would scan both Table1 and Table2, produce two temporary results, and then apply the outer join between them. An English-like description of such execution is below.

```
Step1: Do a RETRIEVE step from Table1 table with a residual
condition of BEGIN(Table1. validity1)<= DATE '2012-10-11'
AND END(Table1. validity1) > DATE '2012-10-11' into Spool 2.
Step 2: Do a RETRIEVE step from Table2 table with a residual
condition of BEGIN(Table2.validity2) <= DATE '2012-10-11'
AND END(Table2. validity2) > DATE '2012-10-11' into Spool 3.
Step 3: Do an outer join between Spool 2 and Spool 3 with
join condition Table1.i1 = Table2.i2 and Table1.f2 < 10.
Save result in Spool 1.
Step 4: Return the contents of Spool 1.
```

Another issue with this execution plan is the late application of the "Table2.f2 < 10" filter which can be applied to Table2 before the join. Teradata's optimizer has a rich suite of optimization rewrites that are used to transform queries into a more performant structure [6, 7]. Join elimination, view folding, predicate derivation and move-around are examples of such rewrites. Applying the optimization rewrites simplifies $\mathcal{Q}$ 6 to $\mathcal{Q}$ 7 below by folding the two derived tables.

$\mathcal{Q}$ 7.

```
SELECT Table1.c1, Table2.c2
FROM Table1 LEFT OUTER JOIN Table2
ON    Table1.i1 = Table2.i2
AND   Table2.f2 < 10
AND   BEGIN(Table2.validity2) <= '10-11-2012'
AND   END(Table2.validity2)   >  '10-11-2012'
WHERE BEGIN(Table1.validity1) <= '10-11-2012'
AND   END(Table1.validity1)   >  '10-11-2012';
```

Note that placing temporal constraint of Table1 in the WHERE clause accomplishes the semantics of applying these constraints prior to the outer join. Similarly, the temporal constraints of Table2 is placed into the ON clause to insure the semantics of applying them prior to the join.The execution plan of $\mathcal{Q}$ 7 is shown below where the join is applied directly and "Table2.f2 < 10" filter is applied on Table2.

```
Step1: Do an outer join between Table1 with a condition of
BEGIN(Table1.validity1) <= DATE '2012-10-11' AND END(Table1
.validity1) > DATE '2012-10-11" AND Table2 with a condition
of BEGIN(Table2.validity2) <= DATE '2012-10-11' AND END(
Table2.validity2) > DATE '2012-10-11' AND Table2.f2 < 10.
The outer join is applied with  join condition Table1.i1 =
Table2.i2. Save results into Spool 1.
Step 2: Return the contents of Spool 1.
```

Note that applying "Table2.f2 < 10" condition prior to or during the join is semantically correct. However, it is more performant to apply the filter before the join, by moving the predicate from the ON clause to the access of Table2.

## 3.4 Further discussion

We envision that our rewrite approach can also be used for other temporal-specific operations like temporal coalescing. Coalescing is a core operation for temporal databases [4]. It combines value-equivalent rows having adjacent/overlapping timestamps. Due to its value to temporal query processing, coalescing is placed on top of a list of key future extensions for ANSI SQL [9]. Our proposal to support temporal coalescing in Teradata through query rewrites is based on using ordered analytic functions and run-time conditional partitioning [3]. Ordered analytic functions are SQL constructs to apply aggregation functions to a partition of the data. Runtime partitioning is a Teradata functional enhancement to ordered analytic functions to define data-based partitioning conditions on window aggregate processing using "RESETWHEN" construct. This novel rewrite idea makes it possible to express coalescing in a join-free single-scan query.

## 4. PERFORMANCE STUDY

To measure the performance of Teradata's temporal implementation, we used a workload similar to the one described in [2] that adds a temporal flavor to the TPC-H benchmark. TPC-H [16] represents a retailer shipping *parts* from *suppliers* to *customers* with *orders* of *lineitems*.

Experiments were run on a Teradata 8-node 2690 appliance with release 13.10, using the TPC-H workload at scale factor 1000 (1 TB). Temporal portion of the workload was based on three tables: Partsupp that tracks which suppliers provide what parts, Parttbl and Supplier that contain data on parts and suppliers, respectively. These tables were rendered temporal with the addition of VALIDTIME and TRANSACTIONTIME. But, their physical design is the same as their non-temporal counterparts to allow for apple-to-apple performance comparisons. Temporal tables were updated to produce history data, which increased the number of rows from twice for the bigger table to nine times

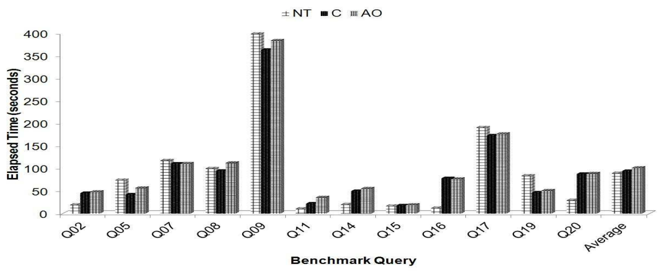

**Figure 2: Elapsed Time of Benchmark Queries in [2]**

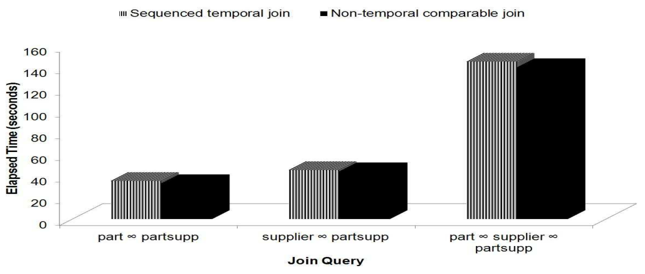

**Figure 3: Elapsed Time of Join Queries**

for the smaller ones. We focused on VALIDTIME and left TRANSACTIONTIME out of our SQL which is equivalent to using CURRENT for TRANSACTIONTIME.

As in [2], 13 TPC-H queries involving the three temporal tables were run. The queries were run for CURRENT and AS OF[2] in addition to the non-temporal original version from TPC-H. NONSEQUENCED queries were excluded because their execution follows the non-temporal path and is not representative of the temporal implementation. We did not performed SEQUENCED experiments with the TPC-H queries because the temporal implementation currently supports only inner joins and no aggregations for SEQUENCED.

Figure 2 and Figure 3 portray the results with elapsed times measured in *seconds*. Figure 2 shows that, on average, for CURRENT VALIDTIME (labeled C) the elapsed time increase over non-temporal (labeled NT) is only of the order of 5%. For AS OF version (labeled AO), we selected a date for which the number of rows qualifying was similar to CURRENT. There was only another 5% increase. For the SEQUENCED directive, we developed three SEQUENCED inner join queries between the three temporal tables. The first query joins Part and Partsupp. The second joins Supplier and Partsupp. The third introduces a three way join between the three tables. Figure 3 shows that elapsed time of SEQUENCED and non-temporal joins is nearly the same.

In summary, the performance of temporal queries in our system is comparable to their non-temporal counterparts running on tables with the same row count and size. This validates that the rewrite approach is not only simple and extensible, but also comes with good performance.

## 5. RELATED WORK

There is a little related work to the scope of our paper that deals with adding temporal dimension to an existing DBMS. A few temporal prototype implementations exist on top of Oracle [11], on top of DB2 [5], and by extending Ingress [1]. These prototypes were built outside the respective DBMS and have a different scope than what we are looking at.

Both Oracle and DB2 added temporal processing to their database technologies. Temporal support in Oracle is done via workspace manager [12]. The implementation is done through views which is the simplest form of a rewrite. There are no performance analysis done to that implementation. For DB2, temporal extensions were added in the "DB2 10 for Z/OS" release [8]. The implementation requires users to define current (i.e., base) and history tables separately and link them to form a space for a temporal table. For DML operations, users just reference the base table and DB2 will access the history table as needed based on the date ranges.

[2]AS OF is a directive applicable to valid and transaction time and it implies an overlap with a specific point in time.

## 6. CONCLUSION AND FUTURE WORK

In this paper, we shared our industrial experience with implementing temporal query processing in Teradata. We discussed in depth the pros and cons of the rewrite and native implementation approaches, and explained why and how we used the rewrite approach. In addition, we validated our approach with a performance study. Our contributions in this paper open avenues for future work. One example is to investigate how the flexibility of our implementation approach can be utilized to further implement other temporal-specific operators (e.g., temporal coalescing as suggested in 3.4).

## 7. REFERENCES

[1] I. Ahn and R. Snodgrass. Performance evaluation of a temporal database management system. In *SIGMOD*, pages 96–107, 1986.

[2] M. Al-Kateb, A. Crolotte, A. Ghazal, and L. Rose. Adding a temporal dimension to the TPC-H benchmark. In *TPCTC*, 2012.

[3] M. Al-Kateb, A. Ghazal, and A. Crolotte. An efficient SQL rewrite approach for temporal coalescing in the teradata RDBMS. In *DEXA*, pages 375–383, 2012.

[4] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in temporal databases. In *VLDB*, pages 180–191, 1996.

[5] C. X. Chen, J. Kong, and C. Zaniolo. Design and implementation of a temporal extension of SQL. In *ICDE*, pages 689–691, 2003.

[6] A. Ghazal, R. Bhashyam, and A. Crolotte. Block optimization in the teradata RDBMS. In *DEXA*, pages 782–791, 2003.

[7] A. Ghazal, D. Y. Seid, A. Crolotte, and B. McKenna. Exploiting interactions among query rewrite rules in the teradata DBMS. In *DEXA*, pages 596–609, 2008.

[8] IBM. Temporal data management in DB2 for z/OS: `http://www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/`.

[9] K. Kulkarni and J.-E. Michels. Temporal features in SQL:2011. *SIGMOD Rec.*, 41(3):34–43, Oct. 2012.

[10] D. B. Lomet and F. Li. Improving transaction-time DBMS performance and functionality. In *ICDE*, pages 581–591, 2009.

[11] R. Mata-Toledo and M. Monger. Implementing a temporal data management system within Oracle. *J. Comput. Sci. Coll.*, 23(3):76–81, Jan. 2008.

[12] Oracle. Oracle flashback technologies: `http://www.oracle.com/technetwork/database/features/availability/flashback-overview-082751.html`.

[13] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[14] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 1999.

[15] Teradata. Teradata temporal analytics `www.teradata.com/database/teradata-temporal/`.

[16] TPC. TPC-H benchmark: `http://www.tpc.org/tpch/spec/tpch2.14.4.pdf`.