Benchmarking Bitemporal Database Systems: Ready for the Future or Stuck in the Past?

Martin Kaufmann[†§], Peter M. Fischer[#], Norman May[§], Donald Kossmann[†]

[†]Systems Group
ETH Zürich, Switzerland
{martinka,donaldk}@inf.ethz.ch

[#]Albert-Ludwigs-Universität
Freiburg, Germany
peter.fischer@cs.uni-freiburg.de

[§]SAP AG
Walldorf, Germany
norman.may@sap.com

ABSTRACT

After more than a decade of a virtual standstill, the adoption of temporal data management features has recently picked up speed, driven by customer demand and the inclusion of temporal expressions into SQL:2011. Most of the big commercial DBMS now include support for bitemporal data and operators. In this paper, we perform a thorough analysis of these commercial temporal DBMS: We investigate their architecture, determine their performance and study the impact of performance tuning. This analysis utilizes our recent (TPCTC 2013) benchmark proposal, which includes a comprehensive temporal workload definition. The results of our analysis show that the support for temporal data is still in its infancy: All systems store their data in regular, statically partitioned tables and rely on standard indexes as well as query rewrites for their operations. As shown by our measurements, this causes considerable performance variations on slight workload variations and significant overhead even after extensive tuning.

1. INTRODUCTION

A large number of database applications make use of the time dimension, e.g., to plan for the delivery of a product or to record the time a state of an order changed. Especially the need to trace and audit the changes made to a data set and the ability to plan based on past or future assumptions are important use cases for temporal data. These use cases rely on the *bitemporal* data model, i.e., temporal data can refer to the state of the database at a certain time (called *system time*) or the time a fact has been valid in the real world (called *application time*). Until recently, temporal data could be represented in most relational database systems only by adding additional columns and using the date or timestamp data type of SQL without associating further semantics with these values. As a consequence, many applications had to implement basic temporal operations like time travel in the application logic. This not only leads to a tremendous development overhead and often faulty semantics, but also to sub-optimal performance due to naive modeling and execution plans. Providing native temporal data management support inside a DBMS therefore provides a tremendous possibility for improved performance. As database vendors now provide implementations of such temporal data and operators, a natural question is how well they actually perform for typical temporal workloads.

Since no generally accepted workload for the evaluation of temporal DBMS exists, we developed a comprehensive benchmark proposal and presented it at TPCTC 2013 [11]. This benchmark extends the TPC-H schema with a temporal component, provides a definition of updates with well-defined data evolution as well as a wide range of synthetic and application-oriented queries. It therefore allows us to examine the functional coverage as well as the performance of different databases regarding temporal features.

In this paper, we perform a comprehensive analysis of temporal operations on common commercial DBMS. The contributions are:

- Analysis of architecture and implementation of temporal data management of a range of temporal DBMS
- Implementation of the benchmark on four commercial and open-source DBMS with different temporal support and different query languages
- A thorough performance evaluation of the DBMS
- A study of the possibilities and impact of tuning of this workload on the DBMS

The remainder of the paper is structured as follows: It commences with an overview of temporal support in available databases (Section 2). In Section 3 we summarize the main ideas of the TPC-BiH benchmark [11]. We focus on the aspects needed to understand our analysis of the temporal databases tested in this paper. Section 4 provides details and extensions to our previous work, e.g., non-trivial aspects such as (temporal) consistency. The core contribution of this paper, Section 5, is a detailed analysis of four widely available databases on this benchmark. In particular, our evaluation provides insights on tuning temporal databases. Section 6 summarizes our results and discusses future work.

2. TEMPORAL DATABASE SYSTEMS

In this section we give a brief overview on temporal data management as it is the conceptual background of our work. We then describe the temporal support of four production-quality systems as we can derive them from the available resources. Since we are prohibited by the usage terms from publishing explicit results for most of the contenders, we will describe the publicly available information including the name of the systems. Information that stems from our analysis is presented in an anonymized form in Section 5. We were able to get plausible result for four systems,

(c) 2014, Copyright is with the authors. Published in Proc. of the 17th International Conference on Extending Database Technology (EDBT), March 24-28, 2014 - Athens, Greece, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

which we refer to as System A to D. Other database systems also support temporal features, but we did not investigate them either because they do not support the temporal features of SQL:2011, or they were not easily available for testing.

2.1 Temporal Data Management

Representing temporal data in SQL has been acknowledged for a long time by providing basic data types such as DATE and TIMESTAMP. To work with this data, in most database systems they are complemented by various built-in functions to work with this data. However, many database applications require more advanced support for temporal data.

These requirements lead to the development of the bitemporal data model in TSQL2 [22]. In this model modifications of a single row are captured by the *system time* (in [22] called *transaction time*). The system time is immutable, and the values are implicitly generated by the temporal database during transaction commit. Orthogonal to that, validity intervals on the application level are denoted as *application time* (called *valid time* in the original paper).

A rich set of research is available on temporal data management, see e.g. [21, 17] for early work on disk-based systems and [13] for recent work on in-memory databases. But despite the fact that many applications rely on temporal support [21], this feature was not included into the SQL standard until recently [14].

We observed that the standardization revitalized the interest in temporal data management, and hence the performance analysis of temporal databases gains importance. Consequently, we proposed a benchmark for analyzing temporal databases [11]. One insight of our previous work was that an extensive performance evaluation of temporal database systems is needed. Moreover, no comprehensive guidelines for physical database design of temporal databases is available yet. For non-temporal databases, adviser tools for physical database have been developed [4, 19]. In particular, the comments on physical database design in [21] only mention temporal partitioning as a technique for physical database design. Finally, we want to provide some guidance for tuning existing databases for temporal workloads.

In the remainder of this section, we discuss the temporal features of five temporal database systems. This brief survey provides the foundation for our experimental analysis in Section 5.

2.2 Teradata

Architecture. Teradata processes temporal queries by compiling them into generic, non-temporal operations [3]. Hence, only the query compiler contains specific rules for temporal semantics. For example, temporal constraints are considered for query simplification. Consequently, the cost-based optimizer is not able to choose an index structure that is tailored to temporal queries.

SQL Syntax. For defining and using temporal tables, Teradata follows the T-SQL2 specification [2] which is not compatible with SQL:2011. As disscussed below, Teradata supports a large range of temporal features.

Time Dimensions. Teradata supports bitemporal tables, where at most one system time and at most one application time is allowed per table. The system time is defined as a column of type PERIOD(TIMESTAMP) marked with TRANSACTION TIME. Similarly, the application time is represented by a column of type PERIOD(DATE) or PERIOD(TIMESTAMP) marked as VALID TIME. Primary keys can be defined for system time, application time or both. Physical database design seems to be fully orthogonal to temporal support, i.e., table partitions or indexes can be defined in the same way as other columns.

Temporal Operators. Teradata also inherits a rich set of temporal operators from T-SQL2 [2]. Time travel on transaction time is formulated via CURRENT TRANSACTIONTIME, TRANSACTIONTIME AS OF TIMESTAMP âĂŸ<date-or-timestamp>âĂŹ before the query or after a table reference. Similarly, time travel on application time is supported using VALIDTIME instead of TRANSACTIONTIME. The semantics of joins and DML operations can be specified as SEQUENCED VALIDTIME or NONSEQUENCED VALIDTIME as defined in [21]. We did not find examples for a temporal aggregation operation as defined in [13].

Temporal Algorithms and Indexes. As discussed above, specific treatment of temporal operations seems to be limited to the query compiler of Teradata. The query compiler implements special optimizations for join elimination based on constraints defined on temporal tables [3]. As temporal queries are compiled into standard database operations, no specific query processing features for temporal queries seem to be exploited.

2.3 IBM DB2

Architecture. Recently, IBM DB2 announced support for temporal features, see [18] for an overview. In order to use system time in a temporal table, one has to create a base table and a history table with equal structure. Both are connected via a specific ALTER TABLE statement. After that, the database server automatically moves a record that has become invisible from the base table into the history table. Access to the two parts of the temporal table is transparent to the developer; DB2 automatically collects the desired data from the two tables.

SQL Syntax. The SQL syntax of DB2 follows the SQL:2011 standard. There are some differences such as a hard-coded name for the application time which provides convenience for the common case but limits temporal tables to a single application time dimension.

Time Dimensions. DB2 supports bitemporal tables: Application time support is enabled by declaring two DATE or TIMESTAMP columns as PERIOD BUSINESS_TIME. This works similarly for the system time using PERIOD SYSTEM_TIME. Furthermore, the system checks for constraints such as primary keys or non-overlapping times when DML statements are executed.

Temporal Operators. Like the SQL:2011 standard, DB2 mostly addresses time travel on an individual temporal table. Certain temporal joins can be expressed using regular joins on temporal columns, but more complex variants (like outer temporal joins) cannot be expressed. No support for temporal aggregation is provided. Queries referencing a temporal table can use additional temporal predicates to filter for periods of the application time. Following the SQL standard, time travel on both time dimensions is possible to filter ranges of system or application time. DML statements are based on the SEQUENCED model of Snodgrass [21], i.e., deletes or updates may introduce additional rows when the time interval of the update does not exactly correspond to the intervals of the affected rows.

Temporal Algorithms and Indexes. From the available resources it seems that no dedicated temporal algorithms or indexes are available. But of course, DB2 can exploit traditional indexes for the current and temporal table. Also, DB2 does not automatically create any index on the temporal table.

2.4 Oracle

Architecture. Oracle introduced basic temporal features a decade ago with the Flashback feature in Oracle 9i. Flashback comes in different variants: 1) Short time row level restore using UNDO information, 2) restoring deleted tables using a recycle bin, and 3) restoring a state of the whole database by storing previous images

of entire data blocks. With version 11g, Oracle introduced the Flashback Data Archive [16] which stores all modifications to the data in an optimized and compressed format using a background process. Before a temporal table can be used, the data archive has to be created. An important parameter of the data archive is the so-called Retention Period, which defines the minimum duration Oracle preserves undo information. With Oracle 12c the support for system time was complemented by application time.

SQL Syntax. The syntax used by Oracle for temporal features seems proprietary but similar to the SQL:2011 standard. Time travel uses the AS OF syntax, and range queries use the PERIOD FOR clause.

Time Dimensions. As mentioned above, Oracle 12c supports application time, which can be combined with Flashback to create a bitemporal table [15]. Multiple valid time dimensions (i.e., application time) per table are allowed. While the system time is managed by the database alone, it seems that the semantics of DML statements for the application time are handled by the application, i.e., the application is responsible for implementing the (non-) sequential model for updates and deletes.

Temporal Operators. Oracle pioneered the use of the time travel operator on system time and provides a rich set of parameters that go beyond the SQL:2011 standard. The Flashback feature is used to implement time travel on the system time, but the accessible points in time depend on the Retention Period parameter of the Flashback Data Archive. For application times, such constraints do not exist. Like in DB2, there is no explicit support for temporal joins or temporal aggregation.

Temporal Algorithms and Indexes. Starting with Oracle 11g, Oracle has significantly overhauled its Flashback implementation, relying on the Flashback Data Area (which are regular, partitioned tables) for system time versioning instead of undo log analysis. Application time is expressed by additonal columns. Since no specialized temporal storage or indexes exist, Oracle relies on its regular tables and index types for temporal data as well.

2.5 PostgreSQL

Architecture. The standard distribution of PostgreSQL does not offer specific support for temporal data beyond SQL:2003 features. However, patches for support of temporal features as defined in the SQL:2011 standard are available. Consequently there is no support available in PostgreSQL for creating, querying and modifying temporal tables.

Temporal Algorithms and Indexes. We were interested in analyzing PostgreSQL because it implements the GiST data structure [9]. Thus, it may offer superior performance compared to B-Trees on the columns of a period. As GiST can be used to implement spatial indexes such as the R-Tree [7] or Quad-Tree [6] we were able to analyze this advanced index type without having to deal with the extension packs of the commercial alternatives.

2.6 SAP HANA

Architecture. To implement the system time dimension, the SAP HANA database offers the *history table* [20]. A history table is a regular columnar table equipped with two (hidden) columns `validfrom` and `validto` to keep track of the system time of a record. For a visible record the value of the `validto` column is NULL. The system maintains the respective information when a new version of a record is created, or the record is deleted. Furthermore, history tables are always partitioned into at least two parts: In addition to user-defined partitions, the data is partitioned into the currently visible records, and older versions of those records. During a merge operation, records are moved from the current partition to the "history partition" which guarantees fast access to the currently visible records.

SQL Syntax. Currently, SAP HANA does not support the SQL:2011 standard syntax. Only the AS OF operator for time travel can be used globally in a query.

Time Dimensions. HANA provides native support for system time by keeping the snapshot information with the update information and not removing rows that have become invisible. There is no specific support for application time in SAP HANA, but standard predicates on DATE or TIMESTAMP columns can be used to query those columns, and constraints or triggers can be used to check semantic conditions on these columns.

Temporal Operators. The SAP HANA database supports a time travel operator in order to view snapshots of the history table of a certain snapshot in the past (known as AS OF operator). Conceptually, this operator scans both the current and the history partition to find all versions which were valid at the specified point in time. As mentioned above, only one point in time is supported per query.

Temporal Algorithms and Indexes. Time travel is implemented by recomputing the snapshot information of the transaction as it was when it started. This information is used to restrict the query results to the rows visible according to this snapshot.

3. TEMPORAL BENCHMARK

Performance evaluations of the temporal dimension has been the focus of a small number of studies, but a comprehensive benchmark has only recently been established by the authors of this paper.

In 1995, Dunham et al. [8] outlined possible directions and requirements for such a temporal benchmark and included an approach for building such a benchmark using different classes of queries. A study on spatio-temporal databases by Werstein [23] evaluated existing benchmarks and emphasized their insufficient temporal aspects. At TPCTH 2012, Al Kateb et al [2] presented the sketch of bitemporal benchmark based on TPC-H.

In this paper we utilize and refine our initial benchmark proposal [11]. The full SQL statements for various temporal SQL dialects can be found in [12] as well as the schema and generator details. The benchmark definition in [11] includes a schema, the workload description and a set of queries, but only performs a cursory evaluation on a small set of DBMSs. In order to make this paper reasonably self-contained, we will summarize this benchmark here. The database schema is based on the schema of the well-known TPC-H benchmark with extensions to support temporal aspects. The data set is consistent with the TPC-H data for each time in system time history. The data also features non-uniform distributions along the application time dimension. This challenges the databases under test for queries on the application time. In addition, we can use the TPC-H queries without modification for time travel queries (i.e., on system time). As the benchmark mainly targets the current SQL:2011 standard, queries are expressed in the standard syntax where possible. As, due to space constraints, showing the full SQL code for all statements is not feasible, we will provide representative examples in Section 5 of this paper.

3.1 Schema

The schema shown in Figure 1 extends the TPC-H schema with temporal columns in order to express system and application time intervals. This means that any query defined on the TPC-H schema can run on our benchmark and will give meaningful results, reflecting the current system time and the full range of application time versions. The added temporal columns are chosen in such a way that the schema contains tables with different temporal properties: Some tables are kept unversioned, such as REGION and NATION,

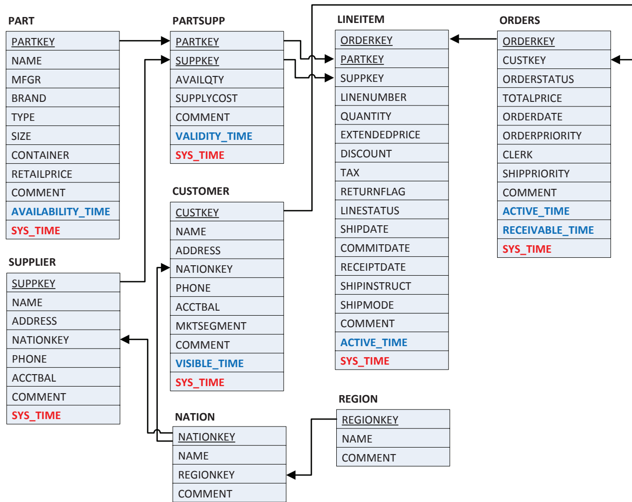


Figure 1: Schema

as this information rarely changes. SUPPLIER simulates a degenerated table by only giving a system time which also serves as an application time. Most other tables are fully bitemporal, and ORDERS represents the case in which a table has multiple application times: `active_time`, i.e., when the order was "active" (placed, but not delivered yet) and `receivable_time`, i.e., when the bill for the order should be paid (i.e., invoice sent to customer, but not paid yet). For DBMS without native support for more than one application time, the first time dimension is defined as native application time, whereas the second application time is expressed with normal time attributes. All time information is derived from existing values present in the data; details can be found in [11].

3.2 Data and Update Workload

Update Scenario	Probability
New Order	0.3
New Customer	0.5
Select existing Customer	0.5
Cancel Order	0.05
Deliver Order	0.25
Receive Payment	0.20
Update Stock	0.05
Delay Availability	0.05
Change Price by Supplier	0.05
Update Supplier	0.049
Manipulate Order Data	0.01

Table 1: Update Scenarios of the History Generator

While the standard TPC-H provides only a limited number of updates in terms of the "refresh" queries, the data produced by the traditional TPC-H data generator serves as a good "initial" data set.

To express the evolution of data, the refresh queries are not considered due to limited impact on the data. Instead, nine updates scenarios are used, stressing different aspects between tables, values and times. The probability of each scenario is given in Table 1. Since the initial data generation and the data evolution mix are modeled independently, we can control the size of the initial data (called h like in TPC-H) and the length of the history (called m) separately and permit arbitrary combinations. The same scaling settings as in TPC-H are used (e.g., $h = 1.0$ yields 1 GB of data), for m a scale of 1.0 corresponds to 1 million updates. For all dimensions, the factor has a linear impact on the data sizes.

Table 2 describes the outcome of applying a mix of these scenarios on the various tables. We distinguish between operations which change the application time and non-temporal operations which only affect the system time. PART and PARTSUPP only receive `UPDATE` statements, while the remaining bitemporal relations will see a mix of operations. LINEITEM is strongly dominated by `INSERT` operations ($>$ 60 percent) whereas ORDERS has 50 percent inserts and 42 percent updates. CUSTOMERS in turn sees mostly `UPDATE` operations ($>$ 70 percent). The temporal specialization follows the specification in the schema, providing SUPPLIER as a degenerate table. The history growth ratio describes how many update operations per initial tuple happen when $h = m$. As we can see, CUSTOMER and SUPPLIER get a fairly high number of history entries per tuple, while ORDERS and LINEITEM see proportionally fewer history operations. Finally, existing application time periods can be overwritten with new values for CUSTOMER, PART, PARTSUPP and ORDERS which triggers more complex update operations [21].

3.3 Queries

The query workload of the benchmark provides a comprehensive coverage of data access patterns [17], temporal operators and specific temporal correlations, using a mix of synthetic and application-oriented queries. There are in total 5 groups (each denoted by a different letter), which we describe briefly; we will highlight individual queries in the experiments in Section 5. As a yardstick we introduce a query (ALL) which retrieves the complete history of the ORDERS table and should set an upper limit to all single-table operations. All queries are available in [12].

Synthetic Time Travel. The first group of queries (prefixed with **T**) covers time travel, i.e., establishing the state concerning a specific time for a table or a set of tables. The time travel operator is expressed in SQL by the AS OF clause for most systems. Given that time in a bitemporal database has more than one dimension, one can specify different slicing options for each of these dimensions, which can be treated as a point or as complete slice.

The first queries cover point-point access for both temporal dimensions, e.g., tomorrow's state in application time, as recorded in the system yesterday. T1 uses CUSTOMER, a table with many update operations, large history, but stable cardinality. In turn, T2 uses ORDERS, a table with a generally smaller history growth ratio and a focus on insertions. This way, we can study the cost of time travel on significantly different histories. The second group, T6, performs temporal slicing on ORDERS, i.e., retrieving all data of one temporal dimension, while keeping the other to a point. This provides insight into whether the DBMS prefers any dimension.

The benchmark also includes a number of queries to test for more specific aspects, such as sharing multiple time-travel operations on the same table (T3, T4), early stops (T4), implicit vs. explicit time specifications (T7) and simulated application time: T8 for point data (like T2) and T9 for slices (like T6).

Time Travel for Application Queries. The second set of timeslice queries (prefixed with **H**) focuses on application-oriented workloads, providing insights on how well synthetic time travel performance translates into complex analytical queries, e.g., accounting for additional data access cost and, possibly, disabled optimizations. For this purpose, we use the 22 standard TPC-H queries (similar to [2]) and extend them to allow the specification of both a system and an application time point. While the semantics for system time specification are rather straightforward, application times need to consider the application time evolution assigned in the data generation process (Section 3.2) when comparing semantics and cost to non-temporal executions of TPC-H.

Table	App.Time Insert	App.Time Update	Non-temp. Insert	Non-temp. Update	Delete	History growth ratio	Overwrite App.Time
NATION	-	-	-	-	-	-	-
REGION	-	-	-	-	-	-	-
SUPPLIER	-	*0.05*	-	-	-	5	no
CUSTOMER	0.15	0.08	-	0.38	-	3.7	yes
PART	-	0.05	-	-	-	0.25	yes
PARTSUPP	-	0.05	-	0.59	-	0.72	yes
LINEITEM	1.20	0.03	-	0.54	0.20	0.32	no
ORDER	0.30	0.25	-	0.54	0.20	0.4	yes

Table 2: Average Operations per Table for History Generator m=1.0 (Million)

Pure-Key Queries (Audit). The third class of queries (prefixed with **K**) retrieves the history of a specific tuple or a small set of tuples along the system time, the application time(s) or both.

The first query (K1) selects the tuple from the ORDER relation (to compare with ALL and T6) using a primary key, returns many columns and does not place any constraints on the temporal range. As such, it should give an understanding of the storage of temporal data. K2 and K3 refine K1 by placing a constraint on the temporal range to retrieve and the number of columns.

K4 and K5 complement K2 by constraining not the temporal range (by a time interval), but the number of versions by using Top-N (K4) or a timestamp correlation (K5). K6 chooses the tuples not via a key of the underlying table, but using a range predicate on a value (o_totalprice).

Range-Timeslice Queries. The fourth class of queries (denoted with **R**) contains a set of application-derived workloads, accessing both value and temporal aspects. As before, these queries contain keep one time dimension to a point, while analysing the other.

R1 and R2 express state modeling, capturing state changes and state durations, respectively. The SQL expressions involve two temporal evaluations on the same relation and then a join.

R3 expresses temporal aggregation, i.e., computing aggregates for each version or time range of the database, with two different aggregation functions[1]. At SAP, this turned out to be one of the most sought-after analyses. However, SQL:2011 does not provide much support for this operator. Since SQL:2011 does not have native support for temporal aggregations, a rather costly join over the time interval boundaries followed by a grouping on these points for the aggregate is required.

R4 computes the products with the smallest difference in stock levels over the history. Whereas the temporal semantics are rather easy to express, the same tables need to be accessed multiple times, and a significant amount of post-processing is required. R5 covers temporal joins by computing how often a customer had a balance of less than 5000 while also placing orders with a price greater than 10. The join, therefore, not only includes value join criteria (on the respective keys), but also time correlation. R6 combines a temporal aggregation and a join of two temporal tables. R7 computes changes between versions over a full set, retrieving those suppliers who increased their prices by more than 7.5 percent in one update. R7 thus generalizes K4/K5 by determining previous versions for all keys.

Bitemporal Queries. The fifth class of queries (denoted with the prefix **B**) covers correlations between different time dimensions. This complements the previous classes which typically varied only one dimension while maintaining a fixed position on the other.

Snodgrass [21] provides a classification of bitemporal queries. Our bitemporal queries follow this approach and create complementary query variants to cover all relevant combinations. These variants span both time dimensions and vary the usage of each time dimension: a) current/(extended to) time point, b) sequenced/time range, c) non-sequenced/agnostic of time. The non-temporal baseline query B3 is a standard self-join: What (other) parts are supplied by the suppliers who supplies part 55? Table 3 describes the semantics of each query. It covers the 9 cases present in [21]. For queries working on points in system time (B3.1, B3.6), we also consider the fact that the data is partitioned by system time with one partition for current values and another one for past values.

Name	App Time	System Time	System Time value
B3.1	Point	Point	Current
B3.2	Point	Point	Past
B3.3	Correlation	Point	Current
B3.4	Point	Correlation	-
B3.5	Correlation	Correlation	-
B3.6	Agnostic	Point	Current
B3.7	Agnostic	Point	Past
B3.8	Agnostic	Correlation	-
B3.9	Point	Agnostic	-
B3.10	Correlation	Agnostic	-
B3.11	Agnostic	Agnostic	-

Table 3: Bitemporal Dimension Queries

4. IMPLEMENTATION

In this section we describe the implementation of the TPC-BiH benchmark introduced in Section 3. The implementation comprises three steps: 1) The Bitemporal Data Generator computes the data set using a temporary in-memory data structure and the result is serialized in a generator archive. 2) The archive is parsed and the database systems are populated. 3) The queries are executed and the execution time is measured.

For experimentation we used the Benchmarking Service described in [10]. We extended this system to consider temporal properties in the metadata, such as the temporal columns in the schema definition as well as particular temporal properties in the selection of parameter to queries (e.g., the system time interval for generator execution).

4.1 Bitemporal Data Generator

The Bitemporal Data Generator computes a temporal workload and produces a system-independent intermediate result. Thus, the same input can be applied for the population of all database systems, which accounts for the different degrees of support for temporal data among current temporal DBMS.

The execution of the data generator includes two steps: 1) loading the output of TPC-H `dbgen` as version 0 and 2) running the update scenarios to produce a history. First, `dbgen` is executed with scaling factor *h* and the result is copied to memory. While parsing the output of `dbgen`, the application time dimensions are derived based on the existing time attributes such as `shipdate` or `receiptdate` of a lineitem. In the second step, *m*1 Mio* update scenarios are executed and the data is updated in-memory.

[1]Our definition of temporal aggregation creates a new result row for each timestamp where data changed. Other definitions may rely on the minimal change of the associated view [5].

The data generator keeps its state in a lightweight in-memory database and supports the bitemporal data model including both application and system time. For the implementation of the system time, we only need to keep the current version for each key in memory. To reduce the memory consumption of the generator, invalidated tuples are written to an archive on disk as it is guaranteed that these tuples will never become visible again. In contrast to this, all application time versions of a key need to be kept in memory as these tuples can be updated at any later point in time. Therefore, an efficient access of all application time versions for a given primary key is necessary. On the other hand, memory consumption has to be minimized as temporal databases can become very large. Since a Range Tree of all application time versions for each primary key turned out to be too expensive, we mapped each primary key to a double linked list of all application time versions which were visible for the current system time version. This representation requires only little additional memory and allows the retrieval of all application time versions for a given key with a cost linear to the maximum number of versions per key.

Initial evaluations show that this generator can generate 0.6 Million tuples/s, compared to 1.7 Million tuples of `dbgen` on the same machine. The data generator can also be configured to compute a data set consisting only of tuples that are valid at the end of the generation interval, which is useful when comparing the cost of temporal data management on the latest version against a non-temporal database.

4.2 Creating Histories in Databases

The creation of a bitemporal history in a database system is a challenge since all timestamps for system time are set automatically by the database systems and cannot be set explicitly by the workload generator (as is possible for the application times). For this reason, bulkloading of a history is not an option since it would result in a single timestamp of all involved tuples. Therefore, all update scenarios are loaded from the archive and executed as individual transactions, using prepared update statements. The reconstruction of the transactions is implemented as a stepwise linear scan of the archive tables sorted by system time order.

In addition, the generator provides an option to combine a series of scenarios into batches of variable sizes, as to determine the impact of such update patterns on update speed as well the data storage, which in turn may affect query performance.

5. EVALUATION AND RESULTS

This section presents the results of the benchmark described in Section 3 that assess the performance of four database systems for temporal workloads. These systems were carefully tuned by taking into account the recommendations of the vendors, and we evaluated different types of indexes. We decided to show both in-memory column stores and disk-based row stores in the same figures as there is no single systems which performs best for all use cases and it is meaningful to compare the effects of the different system architectures. All figures for the experiments are available at [1].

5.1 Software and Hardware Used

In our experiments we compare the performance of four contenders: Two commercial RDMBS (System A and System B) which provide native support of bitemporal features. In addition, we measured a commercial in-memory column store (System C) which supports system time only. As a further baseline, we investigated a disk-based RDMBS (System D) without native temporal support. For each system, we simulated missing native time dimensions by adding two traditional columns to store the validity intervals. As our analysis result will show, this is mostly a usability restriction, but does not affect performance (relative to the other systems).

All experiments were carried out on a server with 384GB of DDR3 RAM and 2 Intel Xeon E5-2620 Hexa-Core processors at 2 GHz running a Linux operating system (Kernel 3.5.0-17). With these resources, we could ensure that all read requests for queries are served from main memory, leveling the playing fields among the systems. If not noted otherwise, we repeated each measurement ten times and discarded the first three measurements. We deviated from this approach under two circumstances: 1) If the measurements showed a large amount of fluctuation, we increased the number of repetitions. 2) For very long-running measurements (several hours), we reduced the number of repetitions since small fluctuations were no longer an issue.

We generally tuned the database installations to achieve best performance for temporal data and used out-of-the-box settings for non-temporal data. If best practices for temporal DBMS management were available, we set up the servers accordingly. In addition, we used three index settings to tune the storage for temporal queries: A) *Time Index*: Add indexes on all time dimensions for RDMBSs, i.e., app time index on current table, app+system time indexes for history tables. B) *Key+Time Index*: Provide efficient (primary) key-based access on the history tables, as several queries rely on this. C) *Value Index*: For a specific query we added a value index, as noted there. These indexes can be implemented by different data structures (e.g., B-Tree or GiST). We also experimented with various combinations of composite time indexes or key-only indexes on the history tables. In the workloads we tested, they did not provide significant benefits compared to single-time indexes.

5.2 Architecture Analysis

Our first evaluation consists of an analysis of how the individual systems actually store the temporal data in physical form; we derived from the documentation, the system catalogs, analysis of query plans and feedback from the vendors. From a high-level perspective, all systems follow the same design approach:

- System time is handled using horizontal partitioning: All data tuples which are valid according to the current system time are kept in one table (which we call *current table*), all deleted or overwritten tuples are kept in a physically separate table (which we call *history table*).
- None of the system provides any specific temporal indexes.
- None of the systems puts any index on the history table, even if it exists by default in the current table.
- All systems support B-Tree indexes.

There are also several differences among the systems:

- System B records more detailed metadata, e.g., on transaction identifiers and the update query type.
- System A and System C use the same schema for current and historic tables whereas System B follows a more complex approach: The current table does not contain any temporal information, as it is vertically partitioned into a separate table. The history table extends the schema of the current table with attributes for the system time validity.
- Updates are implemented differently: System A saves data instantly to the history tables, System B adds updates first to an undo log, System C follows a delta/main approach.

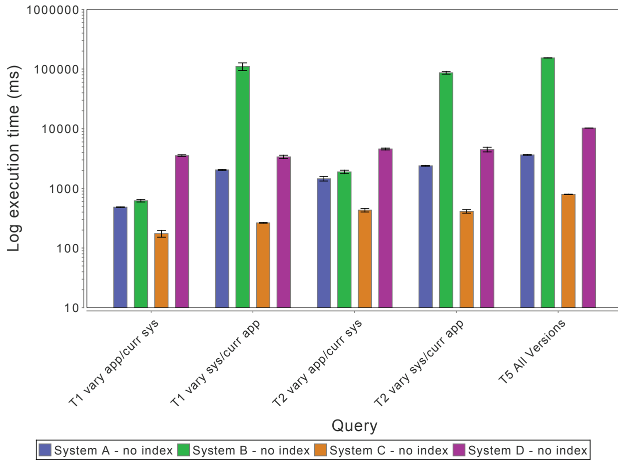


Figure 2: Basic Time Travel (Scaling 1.0/10.0)

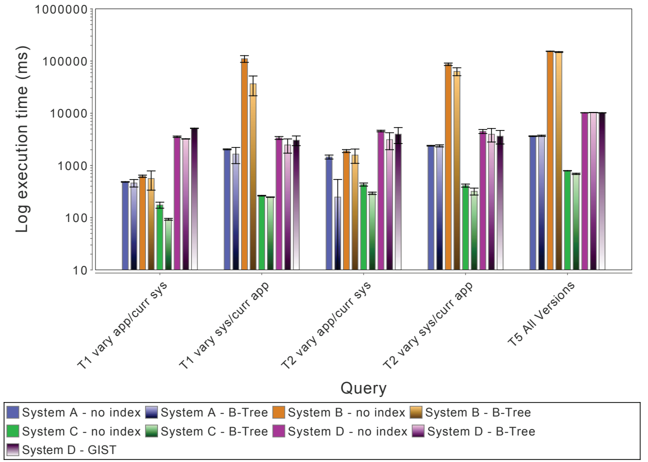


Figure 3: Index Impact for Basic Time Travel (Scaling 1.0/10.0)

- System D stores all information in a single non-temporal table. All other sytems use horizontal paritioning to separate current from historic data.
- Besides B-Tree indexes, System D additional supports indexes based on GiST [9].

5.3 Time Travel Operations

Our evaluation of temporal queries starts with time travel, since this is the most commonly supported and used temporal operation. We utilize the full range of of queries specified in Section 3.3 to stress various aspects of time travel.

5.3.1 Point Time Travel

Our first experiment focuses on point-point time travel, since it provides symmetric behavior among the dimensions and potentially smaller result sizes than temporal slicing, providing more room for optimization. Following the benchmark definition, we use three temporal queries: T1 is a point-point time travel on a stable (non-growing current data) relation, namely PARTSUPP. T2 is a point-point time travel on a growing (current) relation (ORDERS). We compare two orthogonal temporal settings: 1) current system time, varying app time and 2) current app time, varying system time. Finally we evaluate ALL in T5, which retrieves the entire history. This query provides a likely upper bound for temporal operations as a reference. As an example, we give the SQL:2011 (DB2 dialect for application time) representation of T1:

```
SELECT AVG(ps_supplycost), count(*)
FROM partsupp
FOR SYSTEM_TIME AS OF TIMESTAMP '[TIME1]'
FOR BUSINESS_TIME AS OF '[TIME2]'
```

In this experiment, all systems are run with out-of-the box settings without any additional indexes. Figure 2 shows all results grouped by query and temporal dimensions: T1 (stable table) on current system time with varying application time is cheapest for all systems. All systems besides D only access the "current" table and perform a table scan with value filters on the application time, since none of the systems creates any index that would support such temporal filters. T1 over varying system time and fixed application time sees cost increases since the "history" table needs to be accessed as well. Since no indexes are present in the out-of-the-box settings for all systems, this turns into a union of the table scans of both tables. System B sees the most prominent increase, larger than the growth in data in the related tables. The reasons for this increase are not fully clear, since neither the expected cost of combining the three tables nor the EXPLAIN feature of the database account for such an increase. One likely factor is the combination of the vertically partitioned temporal information with the current table, which is performed as a sort/merge join with sorting on both sides. A system-created index on the join attribute is not being used in this workload. T2 is generally more expensive due to the larger number of overall tuples. Since the majority of tuples is current/active, the difference between queries on current system and past system time is somewhat smaller. Again, we see a significant cost increase when utilizing historic data on System B. ALL is most expensive, since it not only needs to scan all tables, but also process all tuples.

5.3.2 Impact of Optimizations

Since no system provides indexes on history tables, and as no index is created to support application time queries on the current table, we examine the benefits of adding temporal indexing. We add the *Time Index* and repeat the previous experiment. For System D we use both B-Tree and GiST versions of the index. As shown in Figure 3, there is limited impact in this particular setting, which is consistent over the DBMSs. Since T2 works on a growing current table, it provides a good opportunity for index usage. System A sees a significant benefit, while System B and D do not draw a clear benefit from this index. In turn, only System B benefits clearly for T1 from the system time index when varying the system time, but is not able to overcome the high additional cost we already observed in the previous experiment. System C does not benefit at all from the additional B-Tree index, which only works if the query is extremely selective. The GiST index does not provide any significant benefit for System D, which we also observed in the following experiments. For the remainder of the evaluation, we will therefore use B-Tree indexes on all RDBMS and no index for System C.

5.3.3 Sensitivity Experiments

To get a better understanding how data parameters affect the execution and gain more insight in the usefulness of indexing, we vary the history length and run T1 with fixed temporal parameters: System time after the initial version and the maximum application time. This way, the query produces the same result regardless of the history scaling and should provide the possibility for constant response time (either by cleverly organizing the base data or use of an index). In contrast to most other measurements, we perform this

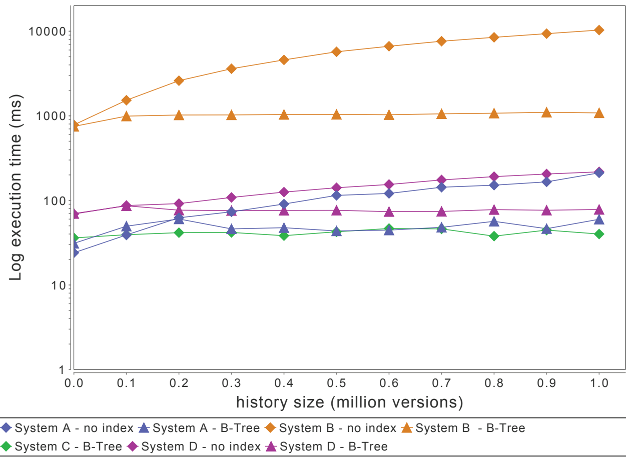


Figure 4: T1 for Variable History Size (Scaling 0.1/1.0)

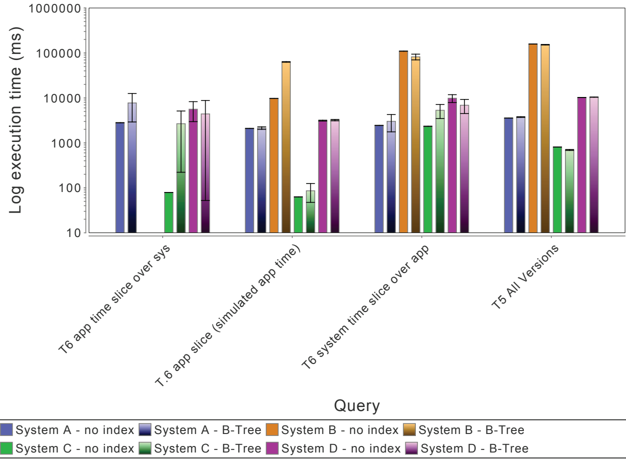


Figure 5: Temporal Slicing (Scaling 1.0/10.0)

experiment on a smaller data set 0.1/0.1 to 0.1/1.0, growing in steps of 0.1 million updates. This is due to the extremely long loading times of a history and the need to perform a full load for all each history. As Figure 4 shows, System A, B and D without indexes scale linearly with the history sizes, as they rely on table scans. With time indexes, all RDBMS (A, B and D) achieve a mostly constant cost. The actual plans change with different selectivity, but once the result becomes small enough relative to the original size, an index-based plan is used. System C is able to achieve constant response times even without an index. As System C does not profit from an index in this experiment, we removed this measurement for a better readability. The GiST index for System D had constantly higher cost than the B-Tree index and is used less frequently.

5.3.4 *Temporal Slicing*

The next class of queries targets temporal slicing, meaning that we fix one dimension (using the AS OF operator) and retrieve the full range of the other dimension. We measure three settings for the same query (T6): 1) fix application time over all complete system time 2) use simulated application time over complete system time 3) fix system time over complete application time: This is the typical behavior of the AS OF SYSTEM TIME clause in SQL:2011 if there is no application time specified. As before, we investigate out-of-the-box and Time Index settings. Figure 5 contains no results for application time slicing for system B due to the bug

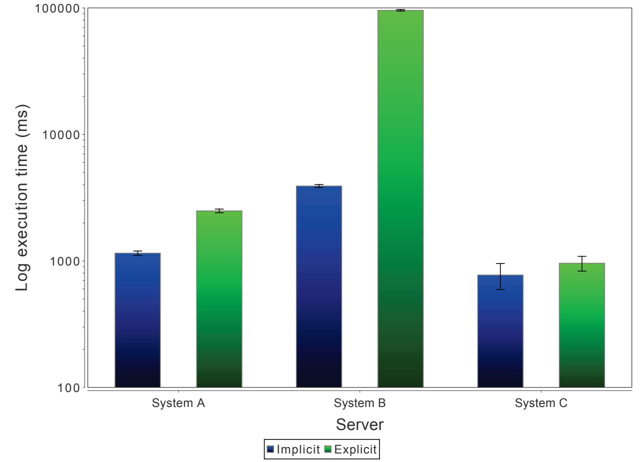


Figure 6: Current TT Implicit vs. Explicit (Scaling 1.0/10.0)

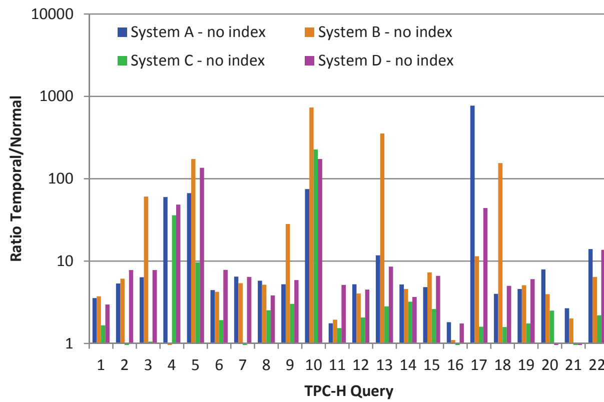
mentioned in Section 5.3.1. The workaround corresponds to the simulated application time. Due the significantly bigger result sizes indexes are of not much use here. Interestingly, temporal slicing results in faster response times than point time travel, in particular for System C due to somewhat lower complexity of the query.

5.3.5 *Implicit vs. Explicit Time Travel*

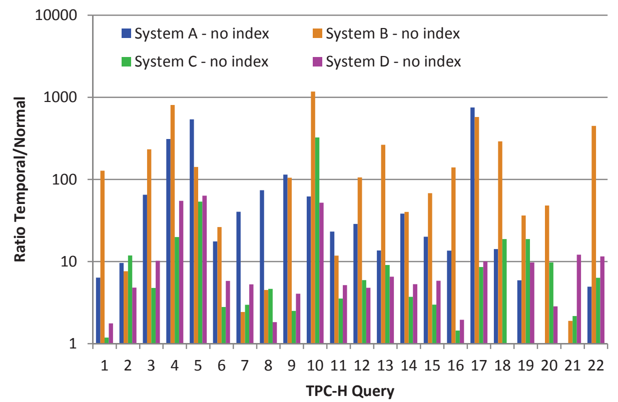
In the previous experiments, we observed quite different cost depending on the usage of history table. We used dedicated "current" queries (not specifying a system time) to ensure only access to the current table – which we call implicit current time. An alternative is to provide an explicit system time statement which targets the current system time – which we call explicit current time. The second option is more flexible and general, and should be recognized by the optimizers when it considers which partitions to use. For this experiment, we consider the systems (A, B, C) with native temporal support only, since we do not use a partition in System D. As the results in Figure 6 and the query plans show, all three system access the history table when using the explicit version as none of them recognizes this optimization.

5.4 Time Travel for Complex Analysis

Time travel measurements so far focused on the behavior of individual time travel operators in otherwise "light" workloads. We complement this fine-grained, rather synthetic workload with complex analytical queries (the original TPC-H workload), but let them move through time. As before, we stress both the application time and the system time aspect. The measurement on the bitemporal data table with scaling factor 1.0/10.0 is compared to a measurement on non-temporal tables that contain the same data as the selected version. Clearly, the bitemporal version will have to deal with more data (and thus more cost). Given the large design space of possible index settings for TPC-H even in the absence of temporal data, we opted for a two-pronged approach: Our baseline evaluation performs all workloads using only the default indexes on all systems. In addition, we performed a more detailed study (see Appendix 1 on [1]) on indexing benefits using the index advisor for one of the candidates (System A). As input for the advisor, we used TPC-H queries 1 to 22 in equal frequency, but not our update statements, as to focus on the benefits for retrieval. We created all indexes proposed by the advisor, which netted to 54 indexes in the non-temporal case, 30 for the application-time query workload and 309 indexes for the system-time query workload. Generally speaking, indexes for the non-temporal workload were extended



(a) TPC-H with application time TT



(b) TPC-H with system time TT

Figure 7: TPC-H for Different Time Dimensions (Scaling 1.0/10.0)

with the time fields in the temporal workloads. The reduction of indexes for the application-time workloads can be attributed to indexes that allow index-only query answering for non-temporal workloads which cannot be used in the temporal case. In turn, the increased number of indexes for the system-time workloads reflects the history table split.

5.4.1 App Time Travel on current Sys Time

Our first measurement compares data with valid application time intervals on the current system against a non-temporal table that records the same update scenarios.

Figure 7(a) shows the slowdown factor between the queries on the non-temporal tables and the time-travel queries on the temporal tables, both without any additional indexes. Several queries show slowdowns by several orders of magnitude. For some queries (like Q5, Q10), all systems are affected, for other others only a single system (e.g. Q3 and Q13 on system B, Q17 on system A) is affected. In turn, several queries only saw minor cost increases, such as Q11 or Q16. Overall, the geometric mean increased by a factor of 8.8 on System A, 9.3 on system B, 2.5 on System C, and 6.4 on System D. There are several different causes for this slowdown: Despite having the same indexes available as in the non-temporal version, several queries use a table scan instead of an index scan, often combined with a change in join strategies (hash or sort/merge vs index nested loop). This affects for example Q3 on System B, Q4 on System B, Q5 on A, B and D. Some parts of this behavior can be attributed to the fact that the split between current and history tables does not cater well for insert-heavy histories which lead to a growing number of "active" entries. These entries are all stored in the current tables (such as the LINEITEM or ORDERS tables). This is not the only cause, as we see similar plan changes even on "stable" relations (e.g., in the case of queries Q10 and Q13, access to customer changes from index lookup to a table scan). Furthermore, some query rewrites would not be performed, such as in Q17 for System A, keeping a complex nested query. System C sees an overall much smaller slowdown, since its main-memory column store relies much more on scans, and is thus not as sensitive to plan changes as the RDBMSs.

Running the queries on the indexed tables of System A showed a smaller slowdown, the geometric mean now being 5.71. Yet, the indexes are not evenly distributed, ranging from slowdown reduction by a factor of 1000 (Q17) to an relative slowdown by almost a factor of 10 (Q22), as only the non-temporal workload benefits from an index.

5.4.2 System Time Travel

Given our experience with significant performance overheads when accessing history tables (as in Section 5.3.5), our second experiment with complex analytical queries performs a time travel on past system time on the temporal tables. All accesses would go to the version directly before the history evolution, returning the initial TPC-H data.

Figure 7(b) shows the performance overhead of querying the bitemporal data instead of the non-temporal data. Since we use the same queries and access a significant amount of current state, the overall results resemble the results in the previous application time experiment. Yet, the performance overhead is significantly higher. Queries 20 and 21 in System A would not finish within our timeout setting of 1.5 hours. The geometric mean (excluding these two queries for all systems) increased by a factor of 26 for System A, a factor of 73 for System B a factor of 7 for System C and a factor of 12.1 on System D – much more than for the application time experiment before. The slowdowns are much more specific to individual queries and system as in this previous experiment. In particular System B sees quite significant slowdown on queries that were not much affected by application time travel. The most extreme cases are Q4, Q17 and Q22 which see slowdown of factor 1000, 50 and 70 between these two experiments. While these queries already use a table scans and two joins in their non-temporal and application time versions, the plan for accessing system time history involves now several more joins, unions and even anti-joins to completely reassemble system time history. A similar effect can be seen for Q9 on System A. With its focus on scan-based operations, System C is least affected, but we also much more pronounced slowdowns than in the previous experiment. System D has the least overhead among RDBMS, since it does not use the current/history table split.

Using indexes for System A does not really change the story, since the geometric means of the relative overheads is reduced to 11.9. Again the relative overheads vary significantly.

5.5 Key in Time/Audit

Our next set of experiments focuses on evaluating the evolution of an individual tuple or a small set of values over time, as outlined in Section 3.3. We start with experiments studying the full history, study various way to restrict the history, selection by value and the sensitivity against different data set parameters.

5.5.1 Complete Time Range

Our initial experiment aims to understand how the individual systems handle workload which focuses on a small set of tuples

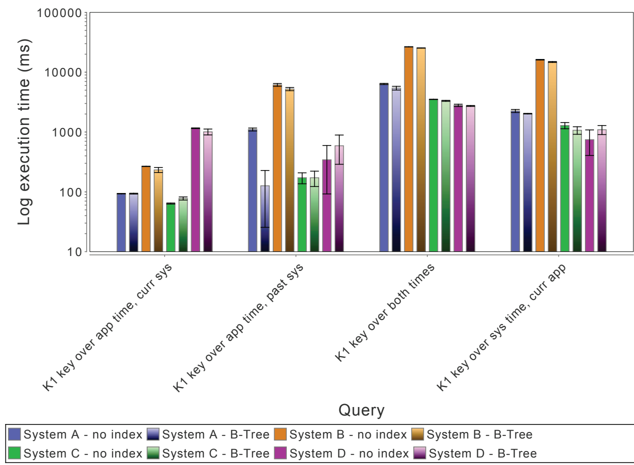


Figure 8: Key in Time - Full Range (Scaling 1.0/10.0)

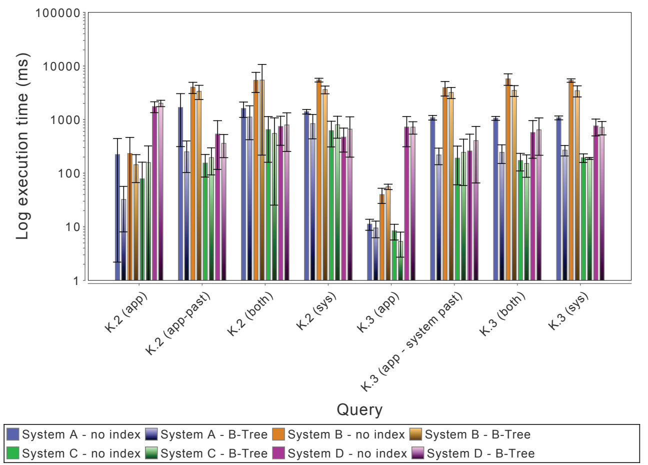


Figure 9: Key in Time - Time Restriction (Scaling 1.0/10.0)

identified by key (aka "key in time"). It evaluates query K1, which accesses an individual customer and traces its evolution over various aspects of time. Like in the previous experiments, we consider application time for current and past system time (as to stress the history tables) as well as system time and a full history over both aspects. We select the customer with most updates, which is still just a small fraction of the small table. For illustration, we give the SQL:2011 code of K1 querying tuples in a system time range and a point in application time:

```
SELECT c_custkey, c_name, c_address, c_nationkey,
    c_phone, c_acctbal,  sys_time_start
FROM ${CUSTOMER}
    FOR SYSTEM_TIME FROM '[SYS_BEGIN]' TO '[SYS_END]'
    FOR BUSINESS_TIME AS OF '[APP_TIME]'
WHERE c_custkey = [CUST_KEY]
ORDER BY sys_time_start
```

As a result, there should be significant optimization potential, which we explore by measuring both nonindex and Key + Time index settings. Figure 8 shows the results: Both System A and System B benefit from a system-defined index on the current table when only querying app time evolution in current system time. When performing the same query in past system time (on the history table), the cost significantly increases, as this triggers a table scan on the history table. System A clearly benefits from adding an index, while System B uses the index, but suffers from the high cost of history reconstruction. In particular, performing a sort/merge join between the vertical partitions of the current table have a significant impact given the overall low cost of the remaining query plan. For histories including system time ranges, the overall cost is higher, while the index benefit is somewhat smaller. System C has to perform table scans for all accesses, thus having a fairly high relative cost. The missing current/history split of System D makes application time history at current system time more expensive.

5.5.2 Constrained Time Ranges

In our second experiment we investigate if the systems are able to benefit from restrictions on the time dimensions for key-in-time queries. Figure 9 shows the results when constraining the time range (K2) and in addition just retrieving a single column (K3), indicating that time range restrictions have little impact in K2 and K3 when comparing against K1. Figure 10 performs a complementary restriction, based not on time but on version count. As a result, we only consider only each individual time dimension,

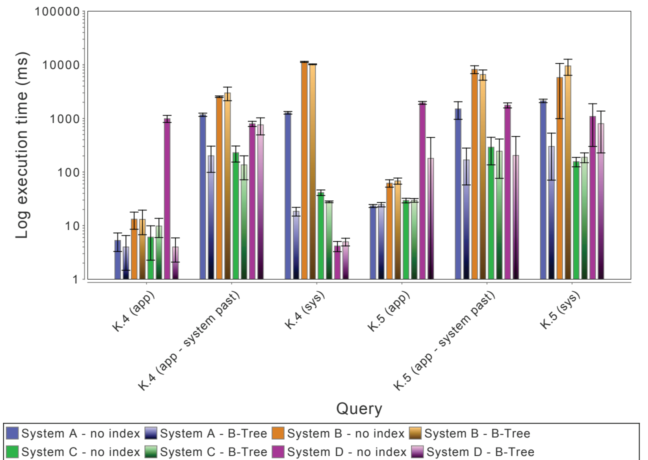


Figure 10: Key in Time - Version Restriction (Scaling 1.0/10.0)

not their combination. K4 implements this version count using a Top-K expression. K5 investigates an alternative implementation retrieving only the latest previous version. Top-K optimizations work in some cases (as shown K4), while the alternative approach in K5 is not beneficial.

5.5.3 History of Non-Key Attributes

Beyond analyzing histories of tuples identified by their keys, we also investigated the cost of choosing tuples by non-key values. Figure 11 shows the cost of K6, which traces the evolution of customers exceeding a certain balance. Without an index, all systems need to rely on table scan. A value index on the balance attribute significantly speeds up the queries, but clearly is influenced by the selectivity of the filter. Due to space constraints, we only show the results for a very selective filter. For the non-selective cases, the index is of little use, so all systems rely on table scans.

5.5.4 Sensitivity Experiments

Similar to experiment in Section 5.3.3, we want to understand how data set changes affect the results. In Figure 12 we vary the history length on the query that following the application time evolution at a fixed system time (once at the begin of the history). System A, C and D manage to keep are more or less constant performance. While System B successfully uses the index for the

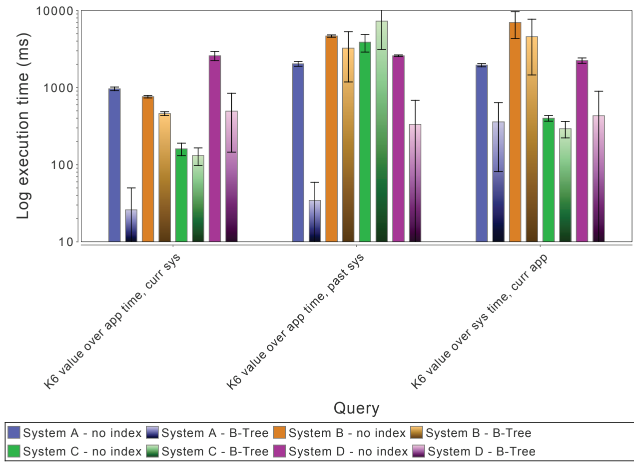


Figure 11: Value in Time (Scaling 1.0/10.0)

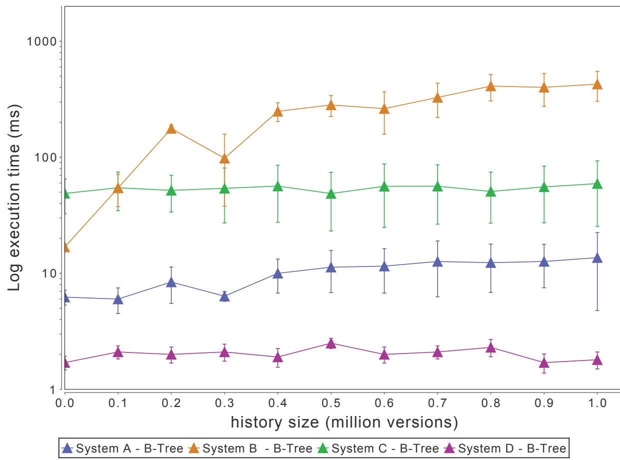


Figure 12: Key-Range Variable History Size (Scaling 0.1/1.0)

actual data, it suffers from the cost of reconstructing the vertical partition on the current table.

We also change the size of the update batches, i.e., how many scenario executions are combined into a transactions as to understand if the number of transactions has an impact. As we can see in Figure 13, System B is impacted most. As they number of transactions decreases, the performance increases. The reasons for this behavior, however, do not become clear from the system description and EXPLAIN output.

5.6 Range-Timeslice

For the application-oriented queries in range-timeslice, we notice that the cost can become very significant (see Figure 14). To prevent very long experiment execution times, we measured this experiment on a smaller data set, containing data for h=0.01 and m=0.1. Nonetheless, we see that the more complex queries (R3 and R4) lead to serious problems: For Systems A and D, the response times of R3a and R3b (temporal aggregation) are more than two orders of magnitude more expensive than a full access to the history (measured in ALL). While System B and C perform better on the T3 queries, System C it runs into a timeout after 1000 seconds on R4. Generally speaking, the higher raw performance of System C does not translate into lower response times for the remaining queries.

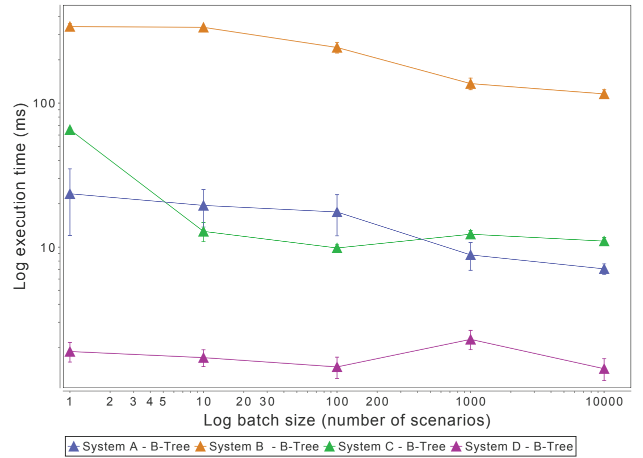


Figure 13: Key-Range for Variable Batch Size (Scaling 0.1/1.0)

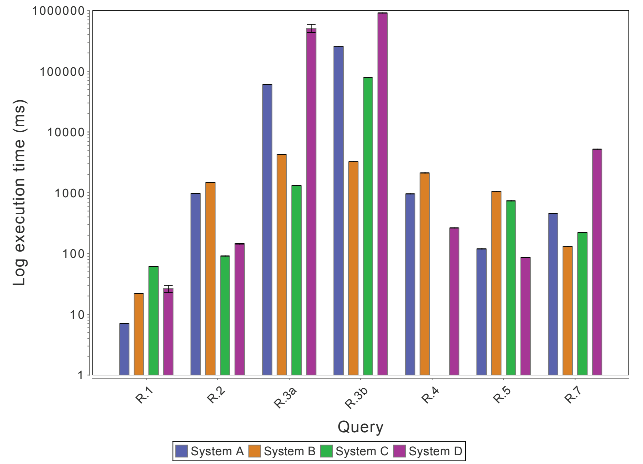


Figure 14: Range Timeslice (Scaling 0.01/0.1)

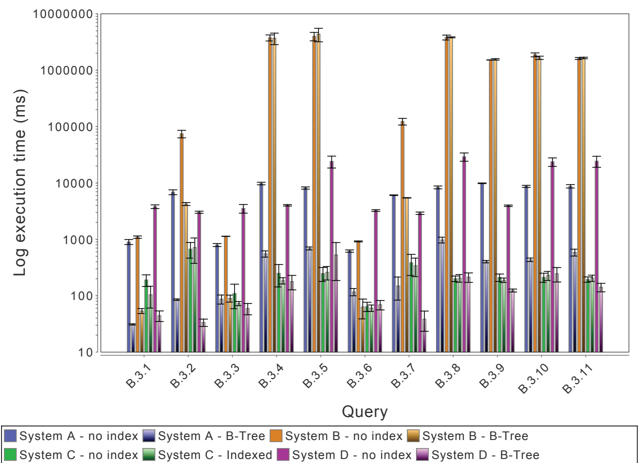


Figure 15: Bitemporal dimensions (Scaling 1.0/10.0)

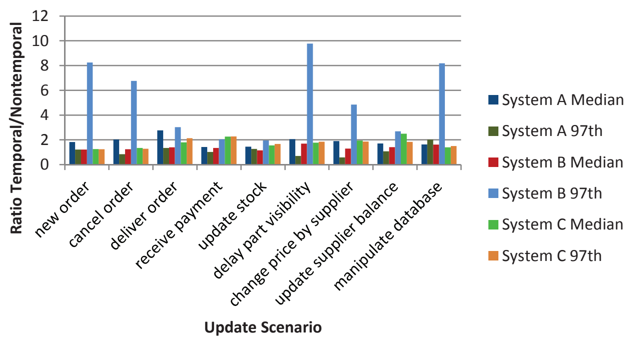


Figure 16: Loading Time per Scenario (Scaling 0.1/1.0)

5.7 Bitemporal Coverage

We measured bitemporal coverage on the 1.0/10.0 data set. As shown in Figure 15, without indexes, most queries turn into tables scan and non-indexed joins. System A and D draw some benefits from the key and time attributes in the indexes, while System B only benefits in selected cases. Generally speaking, the absence of any temporal join operators lead to rather very slow operations when performing correlations.

5.8 Loading and Updates

We measured both the total loading time of the history in native temporal systems. For the workload size 1.0/10.0 the total loading time on System A was 9.7h, on System B 12.4h and on System C 11.3h. Figure 16 shows the average loading time for the transaction of each scenario. As the variance was very high, we computed both the median and the 97th percentile of the execution times. The 97th percentile is very high for System B, as 5% of the values were two orders of magnitudes higher (around 100ms), which can be explained by the background process writing the information to the history table. Since System D does not have native system time, its cost is much lower since we can set the timestamps manually and perform a bulk load.

5.9 Summary

Despite the long history of research in temporal databases, most of today's commercial DBMS only recently adopted temporal features. Therefore, all tested temporal operators are not as mature as these outside the temporal domain. Even though SQL:2011 provides a standard, so far only one system supports this standard which forced us to provide variants in different language dialects for all our queries. In addition, very little documentation is available, in particular on the aspect of tuning, which makes configuration very hard and time consuming. In order to achieve good performance results, extensive manual tuning is required (e.g., by creating indexes), and for many workloads these indexes go unused, since they only work on very selective workloads. In general, temporal features seem to see relatively little usage so far. E.g., we encountered a bug in System B which prevented us from accessing the current data in combination with a specific syntax. System B and C only provided missing or conflicting information about the query plans.

6. CONCLUSION

In this paper we performed a thorough analyis of the temporal data management features of current DBMS. Since all of these systems utilize only standard storage and query processing techniques, they are currently not able to outperform a standard DBMS with a fairly straightforward modeling of temporal aspects. Almost ironically, the system that puts the most effort into system time managment often performs worst in this area. The usefulness of tuning with conventional indexes varies a lot and depends mostly on the overall selectivity of the queries. Temporal operators that are not directly supported in the current language standards (such as temporal aggregation) fare even worse. As management and analysis of temporal data are becoming increasingly important, we hope that the evaluation performed in this paper provide a good starting point for future optimizations of temporal DBMS.

As currently several commercial vendors are planing to release an in-memory column store extension for their database, we plan to repeat this study for the new systems as part of our future work.

7. REFERENCES

[1] Benchmarking Bitemporal Database Systems: Experimental Results. Website, 2014. http://www.pubzone.org/resources/2468150.

[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 et al. Temporal query processing in Teradata. In *EDBT*, 2013.

[4] S. Chaudhuri and V. R. Narasayya. Microsoft index tuning wizard for SQL Server 7.0. In *SIGMOD*, 1998.

[5] R. Fagin, J. D. Ullman, and M. Y. Vardi. On the semantics of updates in databases. In *PODS*, 1983.

[6] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4, 1974.

[7] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[8] M. H. Dunham et al. Benchmark Queries for Temporal Databases. Technical report, Southern Methodist University, 1993.

[9] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *VLDB*, 1995.

[10] M. Kaufmann, P. M. Fischer, D. Kossmann, and N. May. A Generic Database Benchmarking Service. In *ICDE*, 2013.

[11] M. Kaufmann, P. M. Fischer, N. May, A. Tonder, and D. Kossmann. TPC-BiH: A Benchmark for Bi-Temporal Databases. In *TPCTC*, 2013.

[12] M. Kaufmann, D. Kossmann, N. May, and A. Tonder. Benchmarking Databases with History Support. http://www.benchmarking-service.org/pub/tpcbih/. Technical report, SAP AG, 2013.

[13] M. Kaufmann, A. Manjili, P. Vagenas, P. Fischer, D. Kossmann, F. Faerber, and N. May. Timeline Index: A Unified Data Structure for Processing Queries on Temporal Data in SAP HANA. In *SIGMOD*, 2013.

[14] K. G. Kulkarni and J.-E. Michels. Temporal Features in SQL: 2011. *SIGMOD Record*, 41(3), 2012.

[15] Oracle. Oracle database development guide, 12c release 1 (12.1).

[16] R. Rajamani. Oracle Total Recall / Flashback Data Archive. Technical report, Oracle, 2007.

[17] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2), 1999.

[18] C. M. Saracco et al. A Matter of Time: Temporal Data Management in DB2 10. Technical report, IBM, 2012.

[19] K. B. Schiefer and G. Valentin. DB2 universal database performance tuning. *IEEE Data Eng. Bull*, 22, 1999.

[20] V. Sikka et al. Efficient transaction processing in sap hana database: the end of a column store myth. In *SIGMOD*, 2012.

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

[22] R. T. Snodgrass et al. TSQL2 language specification. *SIGMOD Record*, 23(1), 1994.

[23] P. Werstein. A Performance Benchmark for Spatiotemporal Databases. In *In: Proc. of the 10th Annual Colloquium of the Spatial Information Research Centre*, 1998.