TPC-BiH: A Benchmark for Bitemporal Databases

Martin Kaufmann[1,2], Peter M. Fischer[3], Norman May[1], Andreas Tonder[1], and Donald Kossmann[2]

[1] SAP AG, 69190 Walldorf, Germany
{norman.may,andreas.tonder}@sap.com
[2] ETH Zurich, 8092 Zurich, Switzerland
{martin.kaufmann,donald.kossmann}@inf.ethz.ch
[3] Albert-Ludwigs-Universität, Freiburg, Germany
peter.fischer@cs.uni-freiburg.de

Abstract. An increasing number of applications such as risk evaluation in banking or inventory management require support for temporal data. After more than a decade of standstill, the recent adoption of some bitemporal features in SQL:2011 has reinvigorated the support among commercial database vendors, who incorporate an increasing number of relevant bitemporal features. Naturally, assessing the performance and scalability of temporal data storage and operations is of great concern for potential users. The cost of keeping and querying history with novel operations (such as time travel, temporal joins or temporal aggregations) is not adequately reflected in any existing benchmark. In this paper, we present a benchmark proposal which provides comprehensive coverage of the bitemporal data management. It builds on the solid foundations of TPC-H but extends it with a rich set of queries and update scenarios. This workload stems both from real-life temporal applications from SAP's customer base and a systematic coverage of temporal operators proposed in the academic literature. We present preliminary results of our benchmark on a number of temporal database systems, also highlighting the need for certain language extensions.

Keywords: Bitemporal Databases, Benchmark, Data Generator

1 Introduction

Temporal information is widely used in real-world database applications, e.g., to plan for the delivery of a product or to record the time a state of an order changed. Particularly the need for tracing and auditing the changes made to a data set and the ability to make decisions based on past or future assumptions are important use cases for temporal data. As a consequence, temporal features were included into the SQL:2011 standard [9], and an increasing number of database systems offer temporal features, e.g., Oracle, DB2, SAP HANA, or Teradata. As temporal data is often stored in an append-only mode, temporal tables quickly grow very large. This makes temporal processing a performance-critical aspect

of many analysis tasks. Clearly, an understanding of the performance characteristics of different implementations of temporal queries is required to select the most appropriate database system for the desired workload. Unfortunately, at this time there is no generally accepted benchmark for temporal workloads.
For non-temporal data the TPC has defined TPC-H and TPC-DS for analytical tasks and TPC-C and TPC-E for transactional workloads. Especially TPC-H and TPC-C are popular for comparing database systems. These benchmarks query only the most recent version of the data. We propose to leverage the insights gained with TPC-H and to TPC-C while widening the scope for temporal data. In particular, it should be possible to evaluate all TPC-H queries at different system times. This allows us to compare results on temporal data with those on non-temporal data. We carefully introduce additional parameters to examine the temporal dimension. Furthermore, we propose additional queries that resemble typical use cases we encountered in real-world use cases at SAP but also during literature review. In some cases, the expressiveness of SQL:2011 is not sufficient to express these queries in a succinct way. For example, the simulation of temporal aggregation in SQL:2011 results in rather complex queries.
More precisely, we propose a novel benchmark for temporal queries which are based on real-world use cases. As such, these queries retrieve both previous states of the system (i.e., a certain system time) but they also examine time intervals defined in the business domain (i.e., application time); this concept was introduced as the bitemporal data model by Snodgrass [12]. The benchmark we propose contains a data generator which first generates a TPC-H data set extended with some temporal data. In contrast to previous related work (such as [1] and [2]) it also generates a history of values using various business transactions on this data to generate system times. These transactions are inspired by the TPC-C benchmark, and they are designed to keep the characteristics generated by TPC-H `dbgen` at every point in time. Consequently, all TPC-H queries can be executed on the generated data, and their result properties for certain system times are comparable to those in the standard TPC-H benchmark. However, over time the overall data set grows as the previous versions are preserved in order to support time travel to a previous state of the system. In order to evaluate the time dimension, for this benchmark, we define additional queries which retrieve data at different points in time.
The remainder of this paper is structured as follows: In Section 2 we summarize the design goals for our proposed benchmark TPC-BiH. We survey related work on benchmarking temporal databases in Section 3. In the core part of the paper (Section 4), we define the schema, the data generator for temporal data, and the queries comprising the benchmark. We analyze two systems that support temporal queries, and we present performance measurements for our benchmark (Section 5). In Section 6, we summarize our findings and point out future work.

2 Goals and Methodology

The goal of this paper is to present a comprehensive benchmark for bitemporal query processing. This benchmark includes all necessary definitions as well as the relevant tools such as data generators. The benchmark setting reflects real-life customer workloads (which have typically not been formalized to match the current expression of the bitemporal model) and is complemented by synthetic queries to test certain operations. The benchmark is targeted towards SQL:2011, which has recently adopted core parts of the temporal data model. Since the expressiveness of SQL:2011 is limited (no complex temporal join, no temporal aggregation), we provide alternative versions of the queries using language extensions. Similarly, in order to support DBMS's which provide temporal support, but have not (yet) adopted SQL:2011 (like Oracle or Teradata), we provide alternative queries.

The schema builds on a well-understood existing non-temporal analytics benchmark: TPC-H. Its tables are extended with different types of history classes, such as degenerated, fully bitemporal or multiple user times. The benchmark data is designed to provide a range of different temporal update patterns, varying the ratio of history vs. initial data, the types of operations (`UPDATE`, `INSERT` and `DELETE`) as well as the temporal distributions within and between the temporal dimensions. The data distributions and correlations stay stable with regard to system time updates and evolve according to well-defined update scenarios in the application time domain. The data generator we developed can be scaled in the dimensions of initial data size and history length independently, providing support for many different scenarios.

Our query workload provides a coverage of common temporal DB requirements. It covers operations such as *time travel*, *key in time*, *temporal joins*, and *temporal aggregations* – the latter is not directly expressible in SQL:2011. Similarly, we investigate many patterns of storage access and time- vs. key-oriented access with varying ranges and selectivity. The query workload also covers the different temporal dimensions (system and application time): The focus of the queries is on stressing the system for individual time dimensions while considering correlations among the dimensions whenever relevant.

In summary, our benchmark fulfills the requirements mentioned in the benchmark handbook by Jim Gray [3], i.e., it is

- *relevant*, since it covers all typical temporal operations.
- *portable*, since it targets SQL:2011 and provides extensions for systems not completely supporting SQL:2011.
- *scalable*, since it provides well-defined data which can be generated in different sizes for base data and history.
- *understandable*, since all queries have a meaning in application scenarios and in terms of operator/system "stress".

3 Related Work on Temporal Benchmarks

The foundation of the bitemporal data model was established in the proposal for TSQL2 [12]. For a single row the system time – in the original paper called *transaction time* – defines different versions as they were created by DML statements on a row. The system time is immutable, and the values are implicitly generated during transaction commit. Orthogonal to that, validity intervals can be defined on the application level – called *valid time* in the original paper. An example is the specification of the visibility of some marketing campaigns to users. Unlike the system time, the application time can be updated, and both interval boundaries may refer to times in the past, present, or future. The concept of the bitemporal model is now also applied in the SQL:2011 standard [9]. This standard focuses on basic operations like time travel on a single table. Complex temporal joins or aggregations are out of scope, but they are acknowledged as relevant scenarios for future versions of the SQL standard.

The benchmarks published by the TPC are the most commonly used benchmarks for databases. While these benchmarks used to focus either on analytical or transactional workloads, recently a combination has been proposed: The CH-benCHmark [2] extends the TPC-C schema by adding three tables from the TPC-H schema. Yet, no time dimension is included in these benchmarks.

Benchmarking the temporal dimension has been the focus of several studies: In 1995, a research proposal by Dunham et al. outlined possible directions and requirements for such a benchmark. The approach for building a temporal benchmark and the query classes come close to our methods. A later work by Kuala and Robertson [5] provides logical models of several temporal database application areas alongside with queries expressed in an informal manner. The test suite of temporal database queries [4] from the TSQL2 editors provides a large number of temporal queries focused on functional testing rather than performance evaluation. A study on spatio-temporal databases by Werstein [13] evaluates existing benchmarks and concludes that the temporal aspects of these benchmarks are insufficient. In turn, a number of queries are informally defined to overcome this limitation. The work that is most closely related to ours was presented at TPCTC 2012 and includes a proposal to add a temporal dimension to the TPC-H Benchmark [1]. The authors also use TPC-H as a starting point, extend some tables with temporal columns to express bitemporal data, and rely on the data generator and the original queries of TPC-H as part of their workload. Yet, this work seems to be more focused on sketching the possibilities for the bitemporal data model rather than providing explicit definitions of data and queries. Specific differences exist in the language used (we focus on SQL:2011, in [1] a variant of TSQL2 is applied) as well as the derivation of application timestamps (we use existing temporal information in TPC-H for the initial version). Our update scenarios and queries cover a broader range of cases and aim to provide more properties on data and queries.

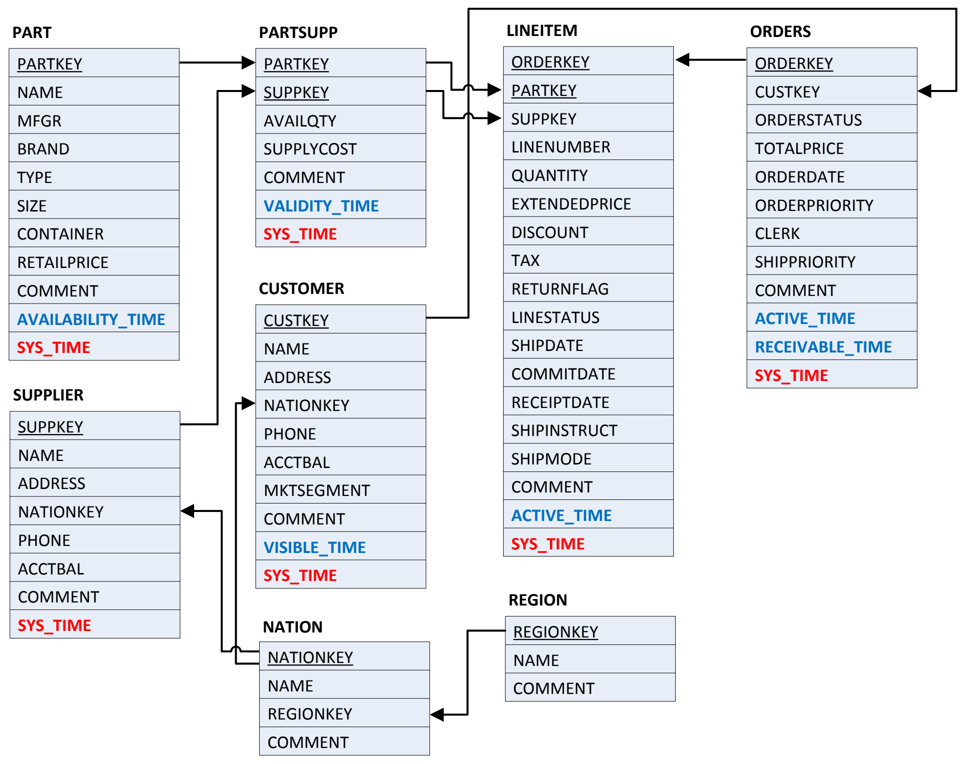

Fig. 1. Schema

4 Definition of the TPC-BiH Benchmark

The definition of our benchmark consists of a schema, properties of the benchmark data and a range of queries. Our benchmark mainly targets the current SQL:2011 standard, but we also show examples how it can be translated to a system with other temporal expressions.

Showing the full SQL code for all statements and queries is not possible due to the space constraints of a workshop paper. Thus, we describe representative examples in this paper and refer to a technical report [7] which includes all queries and definitions in detail.

4.1 Schema

The schema we use in our benchmark is shown in Figure 1. As stated before, it is based on the TPC-H schema and adds temporal columns in order to express system and application times. Each of these time dimensions is stored as an interval and represented physically as two columns, e.g., `sys_time_begin` and `sys_time_end`. This means that any query defined on the TPC-H schema can run on our data set, and will give meaningful results, reflecting the current system time and the full range of application time versions. Specific other temporal dimensions can be added in a fairly straightforward manner. The additional temporal columns are chosen in such a way that the schema contains tables with different temporal properties: Some tables are kept unversioned, some express a correlated/degenerated behavior. Most tables are fully bitemporal, and we also consider the case in which a table has multiple "user" times. Even if the latter is not well specified in the standard, we observe it a lot in customer use cases.

More specifically, we do not add any temporal columns to REGION and NATION. This is also plausible from application semantics, since this kind of information rarely ever changes. All other relations at least include a system time dimension. For SUPPLIER we simulate a degenerated table by only giving a system time. Since this single time dimension is determined by the loading/updating timing, we do not use any temporal correlation queries between this table and truly bi-dimensional tables. For all the remaining relations, we determine the application time from the existing information present in the data: Tuples in LINEITEM are valid as long as any operation like shipping them is pending. Likewise, tuples from PART are valid when they can be ordered, tuples from CUSTOMER when the customer is visible to the system, tuples from PARTSUPP when the price and the amount are valid. Finally, ORDERS has two time dimensions: *ACTIVE_TIME*: when was the order "active" (i.e., placed, but not delivered yet) and *RECEIVABLE_TIME*: when the bill for the order can be paid (i.e., invoice sent to customer, but not paid yet). Both application times become part of the schema. Since current DBMSs only support a single application time, we designate *ACTIVE_TIME* as such, and keep *RECEIVABLE_TIME* as a "regular" timestamp column. Likewise, if a DBMS does not provide any support for application time, application times are mapped to normal timestamp columns.

4.2 Benchmark Data

Complementing TPC-H with an extensive update workload has been proposed before. Given the structural similarity and the wide recognition, TPC-C has been used for this purpose, e.g., in [2]. We also used a similar approach (with additional timestamp assignment) in a previous version of the benchmark [8], but this proved to not be fully adequate: The set of update scenarios is quite small, and does not provide much emphasis on temporal aspects such as timestamp correlations. The query mix also constrains the flexibility in terms of temporal properties, e.g., since a fixed ratio of updates needs to go to specific tables.

The standard TPC-H has only a very limited number of "refresh" queries, which furthermore do not contain any updates to values. Nonetheless, the data produced by the data generator serves a good "initial" data set. The application time columns defined in the schema are initialized with the temporal data already present in this data: Extreme values of shipdate, commitdate and receiptdate define the validity interval of LINEITEM. Given the dependencies among the data items (e.g., LINEITMES in an ORDER), we can now derive plausible application times for all bitemporal tables. Where needed, we complement this information with random distributions. The resulting data will contain data tuples with "open" time intervals, since customers or parts may have a validity far into the future.

To express the evolution of data, we define nine update scenarios, stressing different aspects among tables, values, and times:

1. *New Order*: Choose or create a customer, choose items and create an order on them.
2. *Cancel Order*: Remove an order, its dependent lineitems and adapt the number of available parts
3. *Deliver Order*: Update the order status and the lineitem status, adapt the available parts and the customer's balance.
4. *Receive Payment*: Update currently pending orders and the related customers' balances.
5. *Update Stock*: Increase available parts of a supplier.
6. *Delay Availability*: Postpone the date after which items are available from a supplier to a later date, e.g., due to a shipping backlog.
7. *Price Change*: Adapt the price of parts, choosing times from a range spanning from past to future application time.
8. *Update Supplier*: Update the supplier balance. This update stresses a degenerated table.
9. *Manipulate Order Data*: Choose an "old" order (with the application time far before system time) and update its price. This update changes values while keeping the application times (i.e., trying to hide this change).

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, thus permitting cases like large initial data with a short history ($h \gg m$), small initial data with a long history ($h \ll m$) or any other combination. Similarly to the scaling settings in TPC-H, where $h = 1.0$ corresponds to 1 GB of data, we normalize $m = 1.0$ to the same size, and use the same (linear) scaling rules.

Table 1 describes the outcome of applying a mix of these queries on the various tables. 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. When taking the sizes of the initial relations into account, the bulk of history operations is still performed on LINEITEM and ORDER. A second aspect on which the tables differ is the kind of history operations: SUPPLIER, CUSTOMER and PARTSUPP only receive `UPDATE` statements, whereas the remaining bitemporal relations will see a mix of operations. LINEITEM is strongly dominated by `INSERT` operations ($>$ 60 percent), ORDERS less so (50 percent inserts and 42 percent updates). CUSTOMERS in turn see mostly `UPDATE` operations ($>$ 70 percent). The temporal specialization follows the specification in the schema, providing SUPPLIER as a degenerate table. Finally, existing application time periods can be overwritten with new values for CUSTOMER, PART, PARTSUPP and ORDERS which refers to the use case of updating application time, which is an important feature of the bitemporal data model.

Table	History growth ratio	Dominant Operations	Temporal Specialization	Overwrite App.Time
NATION	None	None	non-versioned	no
REGION	None	None	non-versioned	no
SUPPLIER	5	Update	degenerate	no
CUSTOMER	3.7	Update	fully bitemporal	yes
PART	0.25	Update	fully bitemporal	yes
PARTSUPP	0.72	Update	fully bitemporal	yes
LINEITEM	0.32	Insert	fully bitemporall	no
ORDER	0.4	Insert	fully bitemporal	yes

Table 1. Properties of the History for each Table

We implemented a generator to derive the application times from the TPC-H `dbgen` output for the initial version and generate the data evolution mix. The generator accounts for the different ways temporal data is supported by current temporal DBMS. Initial evaluations show that this generator can generate 0.6 Million tuples/sec, compared to 1.7 Million tuples/sec of `dbgen` on the same machine. The data generator can also be configured to compute a data set consisting purely of tuples that are valid at the end of the generation interval. This is useful when comparing the cost of temporal data management on the latest version against a non-temporal database.

4.3 Queries

Given the multi-dimensional space of possible temporal query classification, we cluster the queries among common dimensions: Data access [10], temporal operators and specific temporal correlations.

Pure-Timeslice Queries (Time Travel). The first group of queries is concerned with testing "slices" of time, i.e., establishing the state concerning a specific time for a table or a set of tables. Also known as *Time Travel*, this is the most commonly supported and used class of temporal queries. Given that time in a bitemporal database has more than one dimension, one can specify different slicing options for each of these dimensions: Each dimension could be treated as a point or as complete slice, e.g., fixing the application time to June 1st, 2013, while considering the full evolution through system time. Further aspects to study are the combination of time travel operations (e.g., to compare values at different points in time), implicit vs. explicit expressions for time and the impact of underlying data/temporal update patterns. The first set of queries is targeted for testing various aspect of time travel in isolation, consisting of nine queries with variants.

T1 and T2 are our baseline queries, performing a point-point access for both temporal dimensions. By varying both timestamps accordingly, particular combinations can easily be specified, e.g., tomorrow's state in application time, as recorded yesterday. The difference between T1 and T2 is according to the underlying data: T1 uses CUSTOMER, a table with many update operations and large history, but stable cardinalities. T2, in turn, uses ORDERS, a table with a generally smaller history and a focus on insertions. This way, we can study the cost of time travel operations on significantly different histories. T3 and T4 correlate data from two time travel operations within the same table. Comparing their results with T2 (very selective) and T5 (entire history) gives an insight into whether any sharing of history operations is possible. T4 adds a TOP N condition, providing possible room for optimization in the database system. T5 retrieves the complete history of the ORDERS table. Given that all data is requested, it should serve as a yardstick for the maximal cost of simple time travel operations. T6 performs temporal slicing, i.e., retrieving all data of one temporal dimension, while keeping the other to a point. This provides insights if the DBMS prefers any dimension, and a comparison of T2 and T5 yields insights if any optimization for points vs. slices are available. T7 complements T6 by implicitly specifying current system time, providing an understanding as to if different approaches of specifying current time work equally well. T8 and T9 investigate the behavior of additional application times, as outlined in Section 4.1. Since the standard currently only allows a single, designated application time, we can study the benefits of explicit vs. implicit application times. In that context, T8 uses point data (like T2), while T9 uses slicing (like T6).

The second set of timeslice queries 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 what [1] proposes) and extend them to allow the specification of both a system and an application time point. Possible evaluations might contain determining the cost of accessing the current version (in both system as well as current application time) compared against the logically same data stored in a non-temporal table (see Section 4.2).

Pure-Key Queries (Audit). The next class of queries we study poses an orthogonal problem: Instead of retrieving all tuples for a particular point in time, we process the history of a specific tuple or a small set of tuples. This way, we can investigate how tuples evolve over time, e.g., for auditing or trend detection. This evolution can be considered along the system time, the application time(s) or both. Additional aspects to study are the effects of constraints on the version range (complete time range, some time period, some versions) and type of tuple selection, e.g., keys or predicates. In total, we specify 6 queries, each with small variants to account for the different time dimensions: K1 selects the tuple using a primary key, returns many columns and does not place any constraints on the temporal range. For key-based histories, this should provide the yardstick, and also offers clear insights into the organization of the storage of temporal data. The cost of this operation can also be compared against T5 and T6, which retrieve all versions of all tuples (for both dimensions or each time dimension, each). To allow easy comparison with the T queries, all queries are executed on the ORDERS relation. K2 alters K1 by placing a constraint on the temporal range. Compared to K1, this additional information should provide an optimization possibility. K3 alters K2 even further by only retrieving a single column, providing optimization potential for decomposition or column stores. K4 complements K2 by constraining not the temporal range (by a time interval), but the number of versions (by using TOP N). While the intent is quite similar to K2, the semantics and possible execution strategies are quite different. K5 constitutes a special case of K4 in which only the immediately preceding version is retrieved, employing no TOP N expression, but a timestamp correlation. From a technical point of view, this provides additional potential for optimization. From a language point of view, such an access is required for queries that perform change detection. K6 chooses the tuples not via a key of the underlying table, but using a range predicate on a value (`o_totalprice`). Besides a general comparison to key-based access, choosing the value of this parameters allows us to study the impact of the selectivity on the computation cost.

Range-Timeslice Queries. As the most general access pattern, range-timeslice queries permit any combination of constraints on both value and temporal aspects. As a result, a broad range of queries falls into that range. We will provide a set of application-derived workloads here, highlighting the variety and the different challenges it brings. As before, these queries contain variants which restrict one time dimension to a point, while varying the other.

R1 considers state change modeling by querying those customers who moved to the US at a particular point in time and still live there. The SQL expression involves two temporal evaluations on the same relation and a join of the results. R2 also handles state modeling, but instead of detecting changes, it computes state durations for LINEITEMs (the shipping time). Compared to R1, the intermediate results are much bigger, but no temporal filters are applied when combining them. R3 expresses temporal aggregation, i.e., computing aggregates for each version or time range of the database. At SAP, this turned out to be one of the most sought-after analyses of temporal data. However, SQL:2011 does not provide much support for this use case. The first query (R3.a) computes the greatest number of unshipped items in a time range. In SQL:2011, this requires a rather complex and costly join over the time interval boundaries to determine change points, followed by a grouping on these boundaries for the aggregates. The second query (R3.b) computes the maximum value of unshipped orders within one year. As before, interval joins and grouping are required. R4 computes the products with the smallest difference in stock levels over the history. While the temporal semantics are rather easy to express, the same tables need to be accessed multiple times, and 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 having orders with a price greater than 10. The join therefore not only includes value join criteria (on the respective keys), but also time correlation. R7 computes changes between versions over a full set, retrieving those suppliers who increased their prices by more than 7.5 percent in a single update. R7 thus generalizes K4/K5 by determining previous versions for all keys, not just specific ones.

Bitemporal Queries. Nearly all queries so far have treated the two temporal dimensions in the same way: Keeping one dimension fixed, while performing different operations types of operations on the other. While this is a fairly common pattern in real-life queries, we also want to gain a more thorough understanding of queries stressing both time dimensions. Snodgrass [11] provides a classification of bitemporal queries. Our first set of bitemporal queries follows this approach and creates 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 2 describes the semantics of each query.

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 2. Bitemporal Dimension Queries

5 Experiments

In order to validate the quality and usefulness of our benchmark, we carried out a number of preliminary performance experiments on systems supporting temporal data. In addition, we added baselines for systems without native temporal support by simulating temporal queries by means of additional columns which represent the time dimension. More specifically, we execute the experiments on System A, a relational DBMS supporting the temporal features of SQL:2011 as well as on System B, an in-memory column store with basic system time support. Since neither of these systems provides documentation on how to tune temporal tables, we utilize out-of-the-box settings. All experiments were carried out on a server with 192GB of DDR3-1066MHz RAM and 2 Intel Xeon X5675 processors with 6 cores at 3.06 GHz running a Linux operating system. The execution was staged and controlled using the Database Benchmarking Service [6].

Most of our experiments were performed on data sizes with an initial data scaling factor h of 1.0 and a history size m of 1.0. The most significant obstacle that currently prohibits us from reaching larger scale factors is the loading process into the database servers. In order provide a suitable system time history for our measurements, individual update cases need to be performed as individual update transactions. The loading process is therefore rather time-consuming, as it cannot benefit from bulk loading. With this loading approach, the timestamps of the system time history are compressed to the period of the loading time, thus requiring adaptations when correlating system and application times.

5.1 Pure Timeslice

Our first measurement concerns the pure-timeslice queries introduced in Section 4.3. For each query we measure a uniform distribution of timestamps over each temporal dimension while keeping the other dimension to the current time, when needed. Figure 2 shows the results for both systems. System A sees relatively little variance since almost all queries have similar cost as full-history retrieval. The only notable exceptions are T3 and T4, which seem to pay an additional cost for the second time travel. System B shows more varied results: Establishing two system times in the same query is not possible, and generally the cost of system time operations exceeds that of application time. Yet, in all cases, more restricted queries yield better response times. Somewhat surprising is the high cost of implicit time travel to the current version (T7).

In order to understand the impact of time travel on complex application queries, we compare the TPC-H benchmark queries on the latest version stored in a normal table with time travel on these queries over temporal tables. As we can see in Figure 3, the impact strongly depends on the individual queries. While the majority of queries only see a small effect, other queries suffer from an order-of-magnitude slowdown (H17, H20). The exact causes need further study, one potential factor being range comparisons with sub queries.

5.2 Key in Time

Figure 4 shows the results for the queries focusing on the evolution of tuples over time, as described in Section 4.3. The results for System A show that using all queries relying on a key over application time at the current system time perform very fast, since they can rely on the primary key of the current table. Any operation referring to historic data in system time is significantly slower (about two orders of magnitude). Yet this cost is still lower than the access to the entire history (T5). Among the variations of the workload, only history length (K1a) and selectivity of tuples choice (K6) have a clear (and plausible) effect. The reduction of the temporal range (K2), projection of attributes (K3), version count ranges (K4) and previous versions (K5) end up costing the same – regardless if the time dimension is application time or system time. System B also shows a performance gap between operations current system time and all

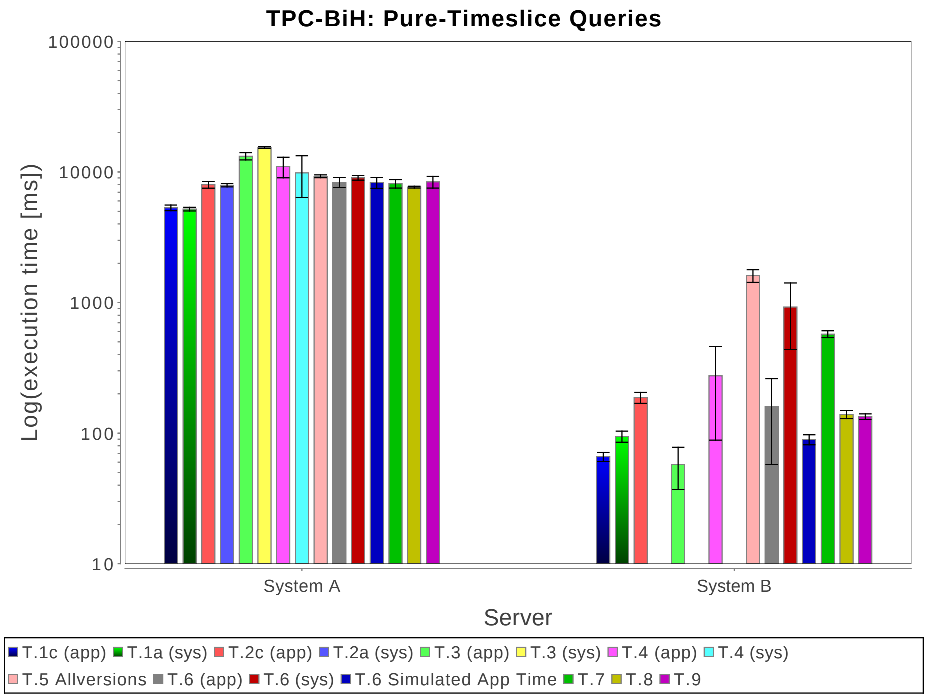

Fig. 2. Time Travel Operations

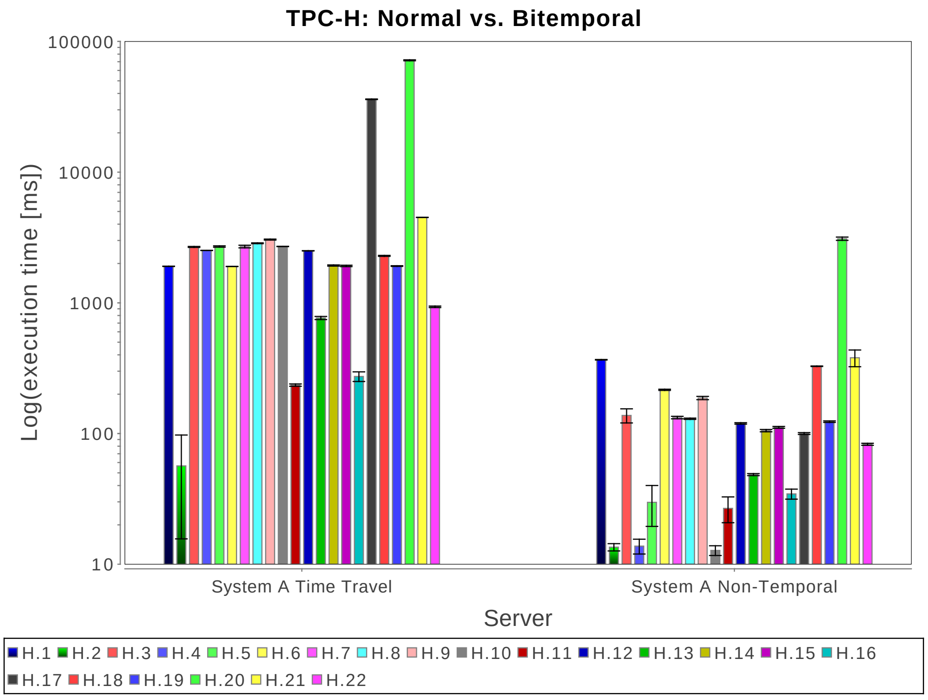

Fig. 3. Time Travel in Applications: TPC-H

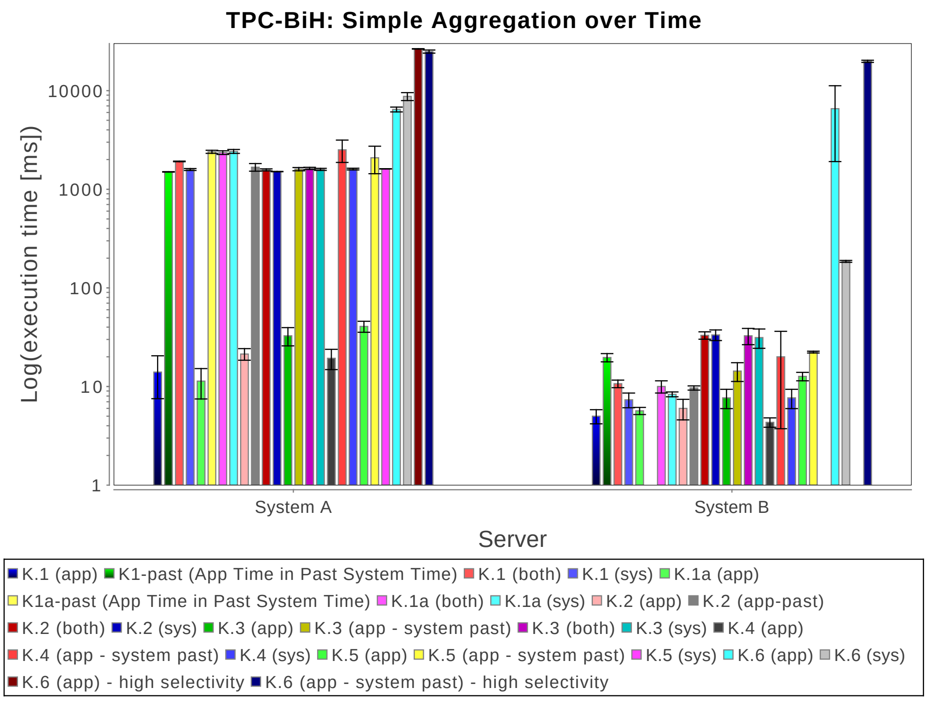

Fig. 4. Key in Time

other data, but it is much less pronounced. Similar effects exist for the value predicate selection.

5.3 Range-Timeslice

For the application-oriented queries in range-timeslice, we notice that the cost can become very significant (see Figure 5). To prevent very long experiment execution times, we measured this experiment on a smaller data set, containing data for h=0.1 and m=0.1. Nonetheless, we see that the more complex queries (R3 and R4) lead to serious problems: For System A, 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 T5). While System B fares somewhat better on the T3 queries, it runs into a timeout after 1000 seconds on R4. Generally speaking, the higher raw performance of System B does not translate into lower response times for the remaining queries.

5.4 Bitemporal Dimensions

The results for combining all temporal dimensions with different types of operations (Figure 6) show some similar patterns: Measurements running purely on current system time perform rather well (3.1, 3.3, 3.5), whereas history accesses are expensive. Application time operations are cheapest when time can be ignored (3.5/3.6 vs 3.1/3.2), followed by temporal joins which are more expensive.

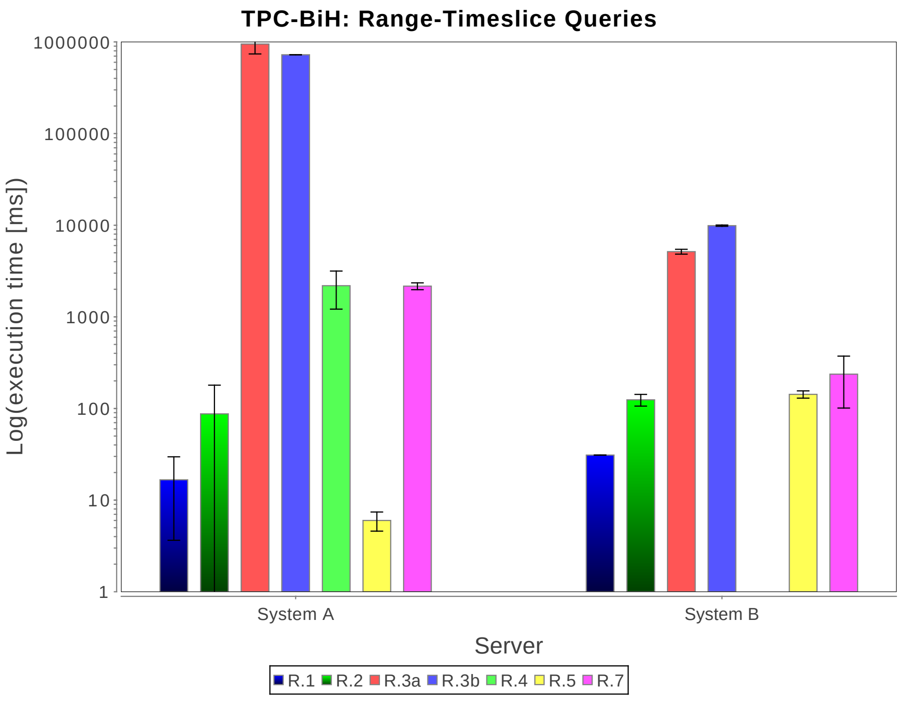

Fig. 5. Range-Timeslice

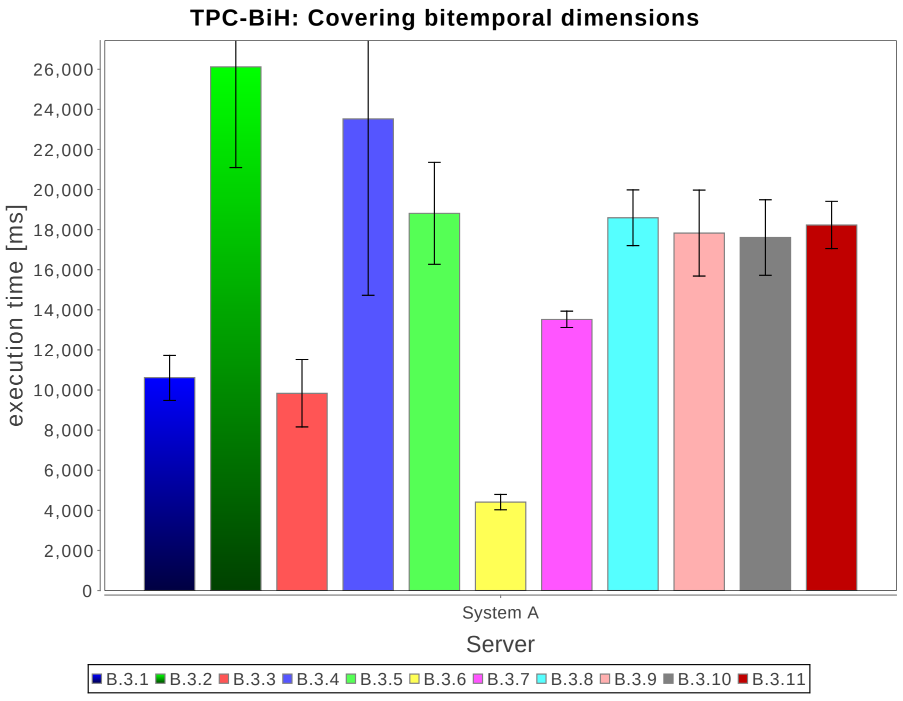

Fig. 6. Bitemporal dimensions

6 Conclusion

In this paper, we presented a benchmark for bitemporal databases which builds on existing benchmarks and presents a comprehensive coverage of temporal data and queries. Preliminary results on existing temporal database systems highlight significant optimization potential and insufficient support for common application use cases in the current SQL:2011 standard. We currently consider the following directions for future work: First, we want to broaden our evaluation, including larger data sets, DBMSs which we have not covered so far and possible tuning guidelines. This will give us further insights into which queries to add and possibly remove for a complete, yet concise coverage of the temporal DBMS workloads. Furthermore, we would like to incorporate explicit update queries, which we can evaluate in their performance characteristics.

References

1. M. Al-Kateb, A. Crolotte, A. Ghazal, and L. Rose. Adding a Temporal Dimension to the TPC-H Benchmark. In *TPCTC*, pages 51–59, 2012.
2. R. Cole et al. The Mixed Workload CH-benCHmark. In *DBTest*, page 8, 2011.
3. J. Gray. *Benchmark Handbook: For Database and Transaction Processing Systems.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.
4. C. S. Jensen et al. A consensus test suite of temporal database queries. Technical report, Department of Computer Science, Aarhus University, 1993.
5. P. P. Kalua and E. L. Robertson. Benchmarking Temporal Databases - A Research Agenda. Technical report, Indiana University, Computer Science Department, 1995.
6. M. Kaufmann, P. M. Fischer, D. Kossmann, and N. May. A Generic Database Benchmarking Service. In *ICDE*, 2013.
7. M. Kaufmann, D. Kossmann, N. May, and A. Tonder. Benchmarking Databases with History Support. Technical report, SAP AG, 2013.
8. 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.
9. K. G. Kulkarni and J.-E. Michels. Temporal Features in SQL: 2011. *SIGMOD Record*, 41(3), 2012.
10. B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2):158–221, 1999.
11. R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 1999.
12. R. T. Snodgrass et al. TSQL2 language specification. *SIGMOD Record*, 23(1), 1994.
13. P. Werstein. A Performance Benchmark for Spatiotemporal Databases. In *In: Proc. of the 10th Annual Colloquium of the Spatial Information Research Centre*, pages 365–373, 1998.