DISS. ETH NO. 21964

Storing and Processing Temporal Data in Main Memory Column Stores

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

MARTIN KAUFMANN

M.Sc. ETH in Computer Science, ETH Zurich
Dipl.-Ing., University of Stuttgart, Germany

born on 20.07.1978

citizen of Germany

accepted on the recommendation of

Prof. Dr. Donald Kossmann, examiner
Prof. Dr. Gustavo Alonso, co-examiner
Prof. Dr. Christian S. Jensen, co-examiner
Dr. Norman May, co-examiner

2014

Abstract

Managing and accessing temporal data is of increasing importance for many applications in industry. Yet, even though there was a significant amount of research in academia during the 1990's, temporal features were only recently included into the SQL:2011 standard. Therefore, only a few temporal operators and with rather poor performance are currently implemented by commercial database systems.

As several important use cases are currently not covered adequately by database systems, many developers model the time dimension on the application layer, rather than pushing down the operators to the database. The implementation of temporal features on the application layer leads to considerable performance overhead.

The goal of this dissertation is to develop native support of temporal features for SAP HANA, a commercial in-memory column store database system.

As no standard benchmark for temporal databases is available, we propose a new benchmark (TPC-BiH) which allows us to evaluate the performance of both commercial database systems and our own implementations.

We investigate different alternatives to store temporal data physically in main memory and analyze the trade-offs arising from different memory layouts that cluster the data either by time or by space dimension.

Taking into account the underlying physical representation, different temporal operators such as temporal aggregation, timeslice and temporal join have to be executed efficiently. We present a novel data structure called *Timeline Index* and algorithms based on this index, which have very competitive performance for all temporal operators. These algorithms beat existing best-of-breed approaches for each operator – in some cases by several orders of magnitude.

While analysing the requirements with clients, it appeared that many applications include more than one time dimension. User-defined time domains, such as the validity of a contract or the availability of a product, are modeled as an application-time domain, whereas the period when a fact was visible in the database is represented by the system-time. For this reason we provide a bitemporal extension of the Timeline Index for bitemporal data.

The Timeline Index is currently being integrated into SAP HANA.

Kurzfassung

Die Verwaltung und die Abfrage temporaler Daten ist für viele industrielle Anwendungen von zunehmender Wichtigkeit. Allerdings wurden temporale Funktionen trotz der umfangreichen wissenschaftlichen Arbeiten aus den 1990er Jahren erst vor kurzem in den SQL:2011 Standard übernommen. Aus diesem Grund sind temporale Operatoren noch nicht lange in kommerziellen Datenbanksystemen verfügbar und weisen zudem noch schlechte Performanz auf.

Da einige wichtige Anwendungsfälle derzeit nicht angemessen von den Datenbanksystemen unterstützt werden, modellieren viele Anwendungsentwickler die Zeitdimension auf Anwendungsebene, anstatt die Operatoren auf dem Datenbanksystem auszuführen, was zu deutlich erhöhten Kosten führt.

Das Ziel dieser Dissertation ist die Entwicklung einer nativen Auswertung temporaler Funktionen für das kommerzielle Datenbanksystem SAP HANA, welches auf Hauptspeicher und Spalten-orientierter Repräsentation der Daten basiert.

Da kein Standardbenchmark für temporale Datenbanken existiert, schlagen wir einen neuen Benchmark (TPC-BiH) vor, welcher uns ermöglicht, die Leistungsfähigkeit aktueller Datenbanksysteme sowie unserer eigenen Implementierungen zu bewerten.

Wir untersuchen verschiedene Alternativen, temporale Daten physisch im Hauptspeicher abzulegen und analysieren die Vor- und Nachteile, die aus der Anordnung der Daten entweder anhand der Zeit- oder der Raumdimension entstehen.

Unter Berücksichtigung der physikalischen Anordnung der Daten werden effiziente Algorithmen zur Ausführung temporaler Operatoren wie temporale Aggregation, Timeslice und temporaler Join benötigt. Dazu haben wir eine neuartige Datenstruktur entwickelt mit dem Namen *Timeline Index*. Algorithmen die auf diesem Index basieren haben sich als äußerst effizient erwiesen und übertreffen aktuelle Implementierungen teilweise um mehrere Größenordnungen.

Während der Analyse der Anforderungen in Zusammenarbeit mit Kunden von SAP stellte sich heraus, dass für viele Anwendungen mehr als eine Zeitdimension erforderlich ist. Benutzerdefinierte Zeitintervalle wie die Gültigkeit eines Vertrags oder die Verfügbarkeit eines Produkts werden als Anwendungszeit modelliert, während der Zeitraum in welchem eine Information in der Datenbank sichtbar war als Systemzeit repräsentiert wird. Wir stellen daher einen erweiterten Timeline Index vor, der bitemporale Daten unterstützt.

Der Timeline Index wird derzeit in SAP HANA integriert.

Acknowledgements

The work presented in this dissertation is the result of a very productive collaboration between people from ETH Zurich, SAP AG in Walldorf/Germany, SAP Waterloo/Canada, University of Freiburg and University of Waterloo. I had the honor to be part of the SAP HANA team while working on my PhD thesis at ETH Zurich. I therefore had the ability to work in a wonderful team of both skilled researchers and experienced developers. This dissertation would not have been possible without them.

First of all, I would like to express my sincere gratitude to Prof. Dr. Donald Kossmann, who gave me the chance to do a PhD as a part of the Systems Group. It was he who introduced me to SAP and allowed me to present my work in Walldorf after I had been working at ETH for six months.

I owe my deepest appreciation to my industrial supervisor Dr. Norman May from SAP AG for his constructive feedback and advice. He was always helpful and willing to share his valuable experience both as a developer and as a researcher in the area of in-memory databases.

My special thanks go to the SAP HANA team, especially Anil K. Goel, Franz Färber and Andreas Tonder. They introduced me to the special requirements of temporal operators in a real world commercial database system, and they even gave me the chance to discuss the use cases directly with SAP customers.

I warmly thank Prof. Dr. Peter Fischer for his valuable feedback and his support during the countless nights before paper deadlines. Peter shared his great experience in benchmarking database systems with me and engaged me in many critical discussions and thus helped me to look at my work from a different angle.

I want to express my gratitude to all the additional members of my committee; Prof. Dr. Christian S. Jensen and Prof. Dr. Gustavo Alonso. They provided me with interesting feedback and comments which helped me to improve this work.

I also want to thank all the Master students for the great collaboration. I am sincerely grateful to Chang Ge from the University of Waterloo (Canada) and Filip Curcic, Bojana Dimcheva, Georgios Gasparis, Sava Ilic, Alessandra Loro, Andreas Lüthi, Amin Amiri Manjili, Florian Köhl, Shanmuganathan Puspanantha, Panagiotis Vagenas and Alessandro Zala from ETH Zurich.

In addition, I thank Simonetta Zysset for proof-reading my paper drafts and this dissertation and thereby letting me benefit from her valuable experience in the English language.

Finally, I wish to thank my family for their love and help during my studies. I also want to thank my girlfriend Stephanie for her patience and support. This dissertation is dedicated to Stephanie and my family.

Contents

1 Introduction

1.1 Motivation

The management of temporal data is a critical feature in many database systems today. 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*). In contrast to "update-in-place", update operations in a system-time database result in creating a new version of an object rather than overwriting old information. Thus, analysts can keep track of modifications of the data and preserve previous states of the data for audits and legal aspects. Orthogonal to the system-time, a temporal table can include several application-time dimensions. Examples are the validity time of a contract or the availability time of a product. Within a temporal database single time dimensions can be relevant for particular tables or both system- and application-time can be combined.

Once the cost of keeping additional versions has been paid, users expect rich capabilities to query and process that data. For instance, users might wish to compare the current status of their investment "portfolio" with the status *AS OF* a year ago. Querying a previous version of a tuple is typically referred to as *timeslice* [55] (called *time travel* in [77]). Another example is the analysis of how many orders are delayed as a function of time in a quality assurance system, thereby querying all previous versions of the database over a certain time period. This application is called *temporal aggregation* [51]. Applications such as Facebook Timeline have brought temporal data and query operators to the limelight.

From an academic point of view, a large body of work in the area of temporal data management is available. The seminal work by Snodgrass et al. [75] introduces a new temporal data model and proposes TSQL2, which is an extension of SQL-92 for temporal data. In [15] Böhlen et al. describe a different temporal data model based on Statement Modifiers, which can be applied to an existing query language to add support for temporal data.

In addition, temporal data structures and algorithms have been the subject of extensive research, summarized, e.g., in [27, 70]. That work covers proposals for index structures (such as multi-version B-trees [7]) and algorithms for certain kinds of queries (e.g., temporal aggregation [14, 51] and temporal joins [27, 88]).

From an industrial perspective, the adoption of temporal database technology has been much slower. SQL has only recently included temporal features as part of the SQL:2011 standard [54]. Even that standard, however, lacks many important features such as temporal aggregation or temporal joins. Database vendors have also been rather hesitant to ship products with temporal features. IBM DB2, for instance, included bitemporal tables only with a recent version that was released in 2012 [71]. The only exception is Oracle, who has been supporting the *timeslice* operator using its Flashback technology for more than 10 years [68]. But even Oracle does not provide an implementation of temporal aggregates and joins.

1.2 Problem Statement

Market traction and lack of incentives are clearly not the problem: Customers are desperate to get rich temporal features. At SAP, for instance, application developers of the financial (FI) and sales & distribution (SD) modules implement temporal operators as part of the application logic because the relational database products do not support these features. Temporal operators are needed for these applications for legal, auditing, and reporting use cases (e.g., risk assessment). Implementing database functionality in the application is not only bad from a developer's productivity perspective, it also kills performance as large volumes of data need to be shipped from the database server to the application server.

The lack of temporal features in state-of-the-art database systems actually has technical reasons. Looking closely at the literature, it turns out that most of the work on temporal data management is highly specialized and proposes index structures and algorithms for a specific temporal function (e.g., temporal aggregation). While all of these functions are important and deserve special attention and tuning, even a global player like SAP cannot afford to implement a new data structure for each kind of temporal query. From a customer perspective, the operational cost of maintaining dedicated index structures on the same data for each kind of temporal query can also be prohibitive.

To our biggest surprise, the most significant knock-out criterion for the majority of the existing proposals from the research literature was performance. We did extensive experiments with the best-of-breed approaches from the literature and found out that the performance results were simply not acceptable for SAP HANA. Digging deeper, it turns out that many of these proposals do not parallelize well and do not work efficiently on modern hardware with many cores, large main memories, and non-uniform memory access (NUMA). For instance, all approaches that are based on tree structures (e.g., B-trees) showed poor performance in our experiments because, even in main memory, a sequential access pattern is essential in order to avoid contention in the memory system. Another problem with such tree-based structures is that they only work well for queries with high selectivity, i.e., queries that select a few tuples based on either a temporal or spatial criterion. As many of our customer use cases involve analysis over large volumes of data, including significant parts of the temporal data, no approach presented in literature so far was applicable for SAP HANA.

The implementation of the temporal features has to meet the following design goals:

In-Memory Column Store. Both current and past versions of the data are represented in a main memory column store. We assume that all data fits in memory of one or several machines.

Easy Integration. SAP HANA is a very large system with more than 6 Million lines of code. Therefore, we aim for a solution which can be integrated into the existing database system easily and which is able to reuse as much of the existing functionality as possible.

Good Performance. The performance of all temporal operators have to meet the requirements of a *real-time analysis*. I.e., the computation of all relevant temporal operations needs to be finished within less than 10 seconds for a common temporal dataset of 100 GB.

Modern Hardware. Our implementation has to consider the properties of modern hardware, that is NUMA awareness and fast scans.

Efficient Memory Consumption. In a commercial main memory column store such as SAP HANA, the storage consumption has to be optimized as memory is an important cost factor.

Low Update Cost. The cost of DML operations (insert, update, delete) has to be minimized and the cost for the maintenance of additional data structures needs to be limited.

Flexibility. As the number of use cases for temporal data is steadily increasing, a unified approach is required which supports many different temporal operators.

1.3 Contributions

The topic of this dissertation is to investigate how temporal features can be implemented natively in a commercial main memory column store, such as SAP HANA. The challenge is to find a unified solution which takes advantage of modern hardware and provides optimal query execution times for the three most important temporal operators at the same time: 1) temporal aggregation 2) timeslice and 3) temporal join.

The technical contributions of this dissertation include:

The TPC-BiH Benchmark. During our work, the performance evaluation of our operators turned out to be a problem as there are no standard benchmarks for temporal databases. The cost of keeping and querying previous versions of the data with temporal operations (such as timeslice, temporal joins or temporal aggregations) is not adequately reflected in any existing benchmark. As a part of this dissertation, we present the TPC-BiH benchmark [45, 46], 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 implemented our workload on a framework based on the Benchmarking Service [43], which includes an abstract and generic model of benchmarks. The results of our benchmark for a number of temporal database systems has been presented in [44], also highlighting the need for certain language extensions [47].

Layouts for Temporal Data in In-Memory Column Stores. The first step is to study alternative approaches to represent temporal data in a main memory column store, as published in [48]. Here we focus on the physical storage of temporal data and scan-based algorithms rather than index data structures. The experiments which we run on the different memory layouts give insight into the fundamental space-time tradeoffs of versioned column stores. A hybrid approach, which partially clusters the data per time and space, shows the most balanced performance for our use cases.

Timeline Index for System-Time. Whereas in the previous approach we considered scan-based solutions only, we now present a novel index data structure called *Timeline Index* [49, 42] and algorithms for processing a large variety of temporal queries for the system-time dimension based on this index. Only one instance of a Timeline Index is required per table, with memory consumption linear with respect to the table size and often only about 30% of the original

data. As demonstrated in [50], the performance results for the temporal operators are very competitive - up to three orders of magnitude faster than current results from related work in literature in the case of temporal aggregation.

Bitemporal Timeline Index. Many applications rely on bitemporal data, 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 for in the real world (called *application-time*). We therefore propose the *Bitemporal Timeline Index*, which extends the basic ideas of the Timeline Index in order to support the full bitemporal data model as covered by the SQL:2011 standard. Comprehensive performance experiments with the TPC-BiH benchmark show that the Bitemporal Timeline Index significantly outperforms all existing commercial database systems and all previous approaches that have been proposed in the research literature to process queries on bitemporal data.

1.4 Overview

This dissertation is organized as follows: In Chapter 2 we explain some use-cases as a motivation for our work and introduce terms and definitions for temporal data. In Chapter 3 we describe the database systems and the prototype which we compared for this work. Chapter 4 defines a benchmark which evaluates the performance of temporal operators and gives an overview of how state-of-the art database systems perform for workloads. Chapter 5 gives a survey of alternative layouts to store temporal data in main memory. In Chapter 6 we introduce the Timeline Index as a unified index data structure which allows for efficient implementations of temporal operators for the system-time dimension. In Chapter 7 we propose an extension of the Timeline Index for bitemporal data. Chapter 8 concludes this dissertation and gives some avenues for future work.

2 Temporal Data

In the first section of this chapter we describe the use cases that motivated our work. These use cases originate from SAP customers and have been collected for analyzing the requirements of the application developers. We describe these use cases in an informal way and show possible implementations in the subsequent chapters.

Furthermore, this chapter includes an introduction of the data model and the theoretical basics for temporal data. We also revisit existing models from literature that are relevant for our work.

In the third section we give an introduction of the temporal operators which are relevant for our use cases.

In the final section of this chapter we define the terminology and notations we will use for the remainder of this dissertation.

2.1 Use Cases

2.1.1 Liquidity Risk Management

The SAP Bank Analyzer is a software which supports the evaluation and analysis of financial products and has been on the market for a long time.

The input and result data of the Bank Analyzer are versioned with respect to two dimensions. These dimensions are called *technical timestamp* and *business key date* which corresponds to system-time and application-time. More concretely, the business key date is the validity time of a contract (in the granularity of days) whereas the technical timestamp is the time (in granularity of minutes) when the record was inserted into the database.

U1a) Business Risk Estimation. A frequent use case in the banking and insurance business is to evaluate the risk of a business to fail before the contract is approved. In this scenario the past is analyzed to derive predictions for the future. For example, an account manager might be interested in the number of open bills within the previous year that exceeded the monthly income of the customer. As the information is stored in a temporal database, the previous value of all unpaid orders can be computed by restoring the state of the database for each point in time.

The parameters for this calculation, such as the time interval or the group of customers, are often adapted manually by the analyst. As the user expects the result to be reported immediately, the computation must be efficient such that the result can be reported within a few seconds.

2.1.2 Business ByDesign

SAP Business ByDesign (ByD) is an ERP system for small and medium sized companies that is hosted by SAP or partners as a service. This system involves several use cases for temporal data.

For analytics and audit purposes, the ByD application requires a database system which stores the data in an *'Insert Only'* manner. In this context 'Insert Only' means that information is only added to the database system, but previous versions of the data are only accessed read-only and never deleted or overwritten. Whereas this technical visibility represents the fact that an information is stored in the database system, the validity time means that a fact is valid in real world. Thus, a separation of these different domains is necessary.

The 'Insert Only' capabilities can be activated for each table individually, i.e., the application can decide if the previous content is stored or not. In any case, the access to the currently visible version must not be influenced by the presence of the past state. The access of previous tuples should not be significantly slower.

The temporal features need to be accessible by means of a generic interface, i.e., a query language such as an extended SQL. It should be possible to query both current and previous tuples within the same query.

The basic assumption is all data is stored in the main memory of a single or a cluster of machines. Thus, the memory consumption has to be optimized as this is an important cost factor for the customer.

The following query scenarios have to be considered:

U2a) Reconstruction Scenario. A previous state of a data set can be retrieved. The user is able to select the point in time and the result is reported in real-time. This feature is used for audits and in case of disputes where it is important to prove the correctness of a previous action.

U2b) Aggregation on Temporal Data. Compute an aggregated value on each point in time for analytical purposes. A frequent use case is maximum value of a stock inventory last year.

U2c) Change Events. A list of change events that have been applied to a data set has to be retrieved in a chronological order.

2.1.3 Automotive Part Manufacturer

A large German company producing automotive parts and household appliances keeps the information of its supply chain in a huge data warehouse which is stored in a temporal database. The schema includes tables with several time domains, i.e., tables with system-time and multiple application-time domains. The current size of the database is 100 GB. Analysts frequently run queries which include the following operations:

U3a) Moving back in Time. An analyst should be able to travel in time to access a previous state of the database system. This use case often has legal aspects, for instance, to proof that a product had certain properties at the time it was ordered.

U3b) Aggregation by Time. It is necessary to compute values aggregated "by time" within a defined time interval. For example, an analyst might be interested in the value of all unshipped items for each day in the previous year for optimizing the supply chain of a mail order business.

U3c) Combination in Time. Information from several tables including the time dimension have to be combined. An example are the total manufacturing costs of a

car based on the planned configuration for a car model and the prices of the parts that have been agreed upon with the suppliers.

As current database systems do not adequately support the operations for U3b) and U3c), all information is currently stored in a non-temporal database system with all temporal queries implemented completely on the application layer. As confirmed by the customer, the performance of many temporal queries is currently not acceptable and sometimes involves more than one hour of computation time.

2.1.4 100 Most Expensive Queries in SAP ERP

SAP ERP (Enterprise Resource Planing) is one of the most important products by SAP. This system allows handling all processes which are relevant in a company such as accounting, human resources and logistics. SAP gathers statistics of all queries generated by the ERP system. The accumulated runtime of all queries in a huge SAP HANA system were evaluated and the top 100 most expensive ERP queries were collected.

Several queries among the top 100 list are temporal in nature. Examples are:

U4a) System-Time Query (Rank 14). The following query implements a simple timeslice (as described in Section 2.3.2) query with respect to the system-time dimension that is defined by the attributes *date_from* and *date_to*.

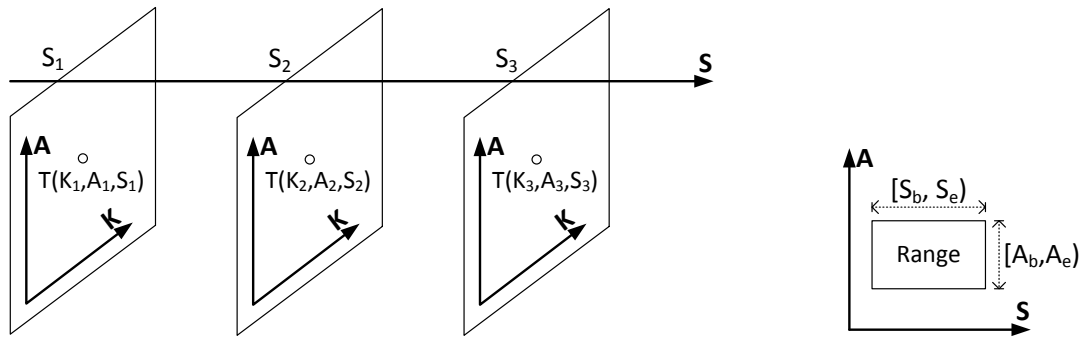
```
SELECT PARTNER1, PARTNER2, RELTYP, DATE_FROM , DATE_TO
FROM BUT050
WHERE CLIENT = ? AND PARTNER1 = ? AND RELTYP = ?
  AND DATE_TO >= ? AND DATE_FROM <= ?
```

U4b) Application-Time Query (Rank 23). This query is similar to the previous use case but involves a timeslice based on application-time rather than on system-time. The attributes of the time interval is given by *begda* and and *endda*.

```
SELECT *
FROM HRP1001
WHERE MANDT = ? AND PLVAR = ? AND SCLAS = ? AND SOBID = ?
  AND BEGDA <= ? AND ENDDA >= ? AND OTYPE = ? AND SUBTY = ?
```

These queries to retrieve the state of the database at a previous point in time are rather simple and include both non-temporal and a temporal conditions. More complex temporal queries such as temporal aggregation, which are even more expensive,

do not appear in this list as they can only be expressed with difficulty in standard SQL at the moment and are therefore modeled on the application domain. This leads to code which is very inefficient, hard to maintain and error-prone. For this reason an important requirement by SAP ERP is to make the execution of temporal queries more efficient.



(a) Time Dimensions (b) Temporal Ranges

Figure 2.1: Bitemporal Data Model

2.2 Bitemporal Data Model

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. However, many database applications require more advanced support for temporal data.

These requirements lead to the development of TSQL2 [75], which is based on the Bitemporal Conceptual Data Model (BCDM). For a single row the *system-time* period – in [75] called *transaction time* – states when this row was visible in the system. DML Statements on a row such as INSERT, UPDATE, DELETE create a new version in system-time. This time dimension is immutable, and the values are implicitly generated during transaction commit. Orthogonal to that, validity intervals can be defined on the application level – in [75] called *valid time*, which states when a row was valid in the real word. An example is the specification of the visibility of a marketing campaign to customers. Unlike the system-time, the *application-time* can be updated at any time, and both interval boundaries may refer to times in the past, present, or future.

According to the definition of a bitemporal relation in [37], one possible interpretation of "bi" refers to exactly one system-time and one application-time per table. An alternative definition given in [37] allows for two different *types* of time dimensions, i.e., there can be one or several system-times and one or several application-times. Driven by the use cases we observed at SAP customers, we stick to the second definition for this dissertation in general: We consider one or several application time dimensions, but one system-time dimension only. We show most examples by means of one time dimension of each type, as additional application-time dimensions can be added analogously. The most frequent use case at SAP is one system-time and several application-time dimensions, which are usually different for each table.

The relationship between the system- and application-time dimension is visualized in Figure 2.1(a), which has been motivated by [55]. In this figure, the system-time is represented by the S axis, the application-time by A and the space-dimension (i.e., non-temporal primary key) is shown as K. The entire (A, K) plain is the data which was current and visible at a particular point in system-time S_1. This information can be interpreted as the current knowledge at time S_1, which includes both previous and future values with respect to application-time A. An example are the marketing campaigns of a product K that have been scheduled within the current business year, which corresponds to a time interval in A. Whenever a row that is contained in (A,K) is changed (i.e., inserted, updated or deleted), the updated information becomes the new *current* information represented by a new (A, K) plain visible at S_2. Therefore, each updated application-time creates a new version in system-time. The opposite is not necessarily the case.

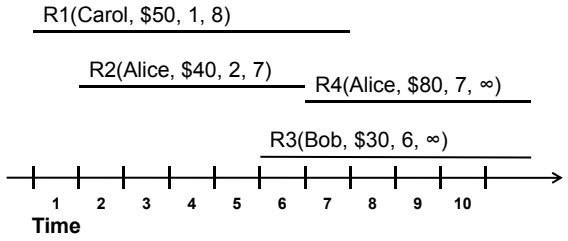
The value for a given (non-temporal) key K is uniquely indicated if a point in time is selected for each time dimension. Thus, (A, K, S) uniquely defines the value of each attribute for this point in time.

For analytical queries, often time intervals are selected in order to investigate how the data evolves with respect to a particular time dimension. Such a time range is selected by means of an interval which is defined by a lower bound b and an upper bound e. In the notation we chose for this dissertation, all intervals are *half open*. More precisely, we represent the validity interval of a tuple by 1) the point in time when the tuple became valid and 2) the time it was invalidated. As illustrated by Figure 2.1(b), an interval can be selected both for the system-time as $[T_b, T_e)$ and the application-time dimension as $[A_b, A_e)$. If a query selects the whole temporal range of one time dimension, we say this query is *agnostic* with respect to that dimension.

The concept of the bitemporal model is now also applied in the SQL:2011 standard [54]. This standard focuses on basic operations like timeslice 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.

	Name	Balance	Begin	End
R1	Carol	$50	1	8
R2	Alice	$40	2	7
R3	Bob	$30	6	∞
R4	Alice	$80	7	∞

(a) Temporal Table



(b) Validity Intervals

Figure 2.2: Temporal Data

SUM (Balance)	Begin	End
$50	1	2
$90	2	6
$120	6	7
$160	7	8
$110	8	∞

Figure 2.3: Instantaneous Temporal Aggregation

2.3 Temporal Operators

In this section we describe the temporal operators which are relevant for the use cases described in Section 2.1. These operators can be evaluated both for system-time and application-time of the bitemporal data model, which we introduced in the previous Section 2.2. For this reason, we just define the time interval as *begin* and *end* in this section without any restriction for either system- or application time.

2.3.1 Temporal Aggregation

Temporal aggregation considers a sequence of versions to compute an aggregated value for a *temporal range*. This range can be evaluated using different *aggregation functions*.

Temporal Range Selection. Many queries require repeated aggregations over time. Examples are the value of an inventory for each point in time or the average number of pending orders per week. Different types of temporal aggregation have been summarized in [26]. Given the task of defining ranges over data ordered by time, we utilize both [26] and the Window concepts from data stream systems [28] and indentify six types of temporal ranges:

1. **Point in Time.** The aggregate is computed and returned as a result for each version or point in time.
2. **Instantaneous Temporal Aggregation.** This range type is similar as point in time, but the result is returned only for each constant interval, i.e., each time interval the aggregated value remains unchanged.
3. **Tumbling Window.** The time-intervals are non-overlapping (e.g., the number of all orders shipped per calendar week).
4. **Sliding Window.** The intervals are overlapping (e.g., the value of all orders shipped within the previous 7 days, are computed each day).
5. **Landmark Window.** The windows are overlapping, but have the same start point (e.g., the number of orders shipped up to each day of this year).
6. **User-defined.** The period for the computation of the aggregated value is defined by the user (e.g., the number of phone calls since the last order has been placed by a customer).

For the remainder of this dissertation, we will refer to *instantaneous temporal aggregation*, as this is the most common use case at SAP. Figure 2.2(a) shows an example of a temporal table from a simple banking application with customer names and their account balance. For each tuple the validity interval is represented by the attributes *Begin* and *End*. The validity intervals of all tuples from Figure 2.2(a) are visualized in Figure 2.2(b).

As an example for an instantaneous temporal aggregation, we compute the *sum* of all account balances at each point in time. This sum can be computed in an incremental way starting from Time 1 in Figure 2.2(b): The balance of a tuple is added to the sum when it becomes visible and subtracted again when the visibility of this tuple ends. In this example, only R1 is visible in the time interval [1,2) leading to total balance of $50. R2 becomes visible at time 2, which increases the sum by $40 leading to a total balance of $90 in the interval [2, 6). A new value is reported for each time interval for which the sum remains constant. The result (shown in Figure 2.3) can again be represented as a temporal table.

Aggregation Functions. On top of these temporal ranges we can apply a number of aggregation functions. The first, most common approach of aggregate computation, only depends on versions within a specified range. In this type, which we call *range-local*, we compute aspects such as the number of unshipped orders or the maximum inventory of a specific item during a period of time. We distinguish *cumulative* aggregation functions (such as `SUM` and `COUNT`) and *selective* aggregation functions (`MIN` and `MAX`).

The second approach of aggregate computation is specific to temporal workloads, but has so far not been investigated much. Instead of considering only the version within the target range, the "history" of a row up to the end of the target temporal range is required. We refer to this use-case as a *non-local aggregate*. This concept is best explained with examples: a) We are interested in the point in time at which a state change occurs (i.e., an order is marked as shipped) and b) we want to measure the state duration (i.e., how long the order was in the state "unshipped"). Since for many typical applications, states of data items are stored (like "unshipped" or "shipped"), but not state changes, we need to inspect previous versions for both cases.

Some aspects of temporal aggregation are well-explored in terms of dedicated data structures and algorithms [14, 87]. The support in SQL:2011 is limited to simple cases in which a single aggregation period is explicitly specified by selection predicates on the *begin* and *end* columns. Given this limited functionality and the large number of use cases requiring more expressive semantics, we have opted for an extension of the SQL syntax (sketched in Section 3.1.3) and semantics, providing explicit, higher-level control over the aggregation parameters. For this extension, we need to consider two orthogonal aspects: defining the *temporal range* and the *aggregation locality*.

The temporal aggregation operator covers the use case U2b) and U3b). It can also be applied for risk prediction use case in U1a), but requires a dedicated aggregation function for computing the risk based on previous values.

2.3.2 Timeslice

The timeslice operator establishes a consistent view of a (past) state of a temporal table. It allows the user to perform regular value queries on a single, usually older version of the data returning the tuples which were visible in the system at a given point in time with respect a certain time dimension. The result of the timeslice operator is a non-temporal table.

As an example, an analyst might be interested in the value of his stock portfolio as of August 1st, 2012. Timeslice is currently the most widespread use case for temporal queries and is supported by several commercial database management systems such as Oracle and DB2. Within SQL:2011, the timeslice operator has been standardized by means of the *AS OF* clause.

In the example shown in Figure 2.2(b), a timeslice to a given point in time can be visualized by slicing all visibility intervals. In this example, at time 9 the tuples R3 and R4 are visible.

Timeslice is the corresponding operation for use cases U2a), U3a), U4a) and U4b). Many practical use cases include several time dimensions (system-time and application-time), which can be combined in the same query.

2.3.3 Temporal Join

A temporal join of two temporal tables returns a new temporal table as a result which includes the tuples for which predicates in both value and time domain are satisfied. In addition to the value condition of a traditional join, the time dimension is added to a temporal join: Tuples match if 1) their value predicate is fulfilled and 2) their time intervals overlap. The semantics of this temporal condition is that the tuples were valid at the same time. Thus, the temporal attributes of each tuple in the result are adapted such that they correspond to the time interval for which the input tuples overlap. In addition, other join conditions are possible, such as as adjacent tuples or outer joins.

For example, we retrieve all orders that customers placed last year while living in New York (customers may have lived at another place earlier or later). SQL:2011 provides rudimentary support for such joins by placing an explicit join condition on the temporal columns, but does not offer any dedicated syntax for temporal joins.

The temporal join is the relevant temporal operator in use case U3c).

2.4 Terminology and Notations

In this section we give a summary of the terms and notations used for the remainder of this dissertation.

Terminology:

- **Dataset:** All facts that are stored in the database, i.e., the tuples in all tables, are referred to as a dataset.
- **System-Time** vs. **Application-Time:** In a temporal database the system-time represents the time at which a certain fact has been *current* (i.e., "visible") in the database. A new version in system-time is created by a committed transaction. In contrast to this, the application-time is maintained by the user and tells when the fact has been "valid" in real world. A **bitemporal** table contains both system- and application-time.

  The SQL:2011 standard includes the definition of *system-versioned tables* and *application-time period tables*. Details are available in ISO/IEC 9075, Database Language SQL:2011 Part 2: SQL/Foundation. For the remainder of this thesis, we will just use the terms *system-time* and *application-time*.
- **Current** vs. **previous version:** A tuple is referred to as *current* with respect to system-time if it has not been invalidated yet, i.e., if it has not been deleted or overwritten by a later version. A tuple becomes *previous* as soon as it has been deleted or a new tuple with the same user-defined primary-key has been inserted. Standard (non-temporal) database systems only store the current version of the information.
- **History:** We call all previous versions of the tuples stored in a temporal database its *history*. We use this term both for system- and application-time as several vendors of temporal database systems (i.e., IBM und SAP) also adopted this term.

Terms and Notations Used for this Dissertation:

In this disseration we use the current standard SQL:2011 [54] (rather than T-SQL2) and adopt its syntax and terms whenever it is possible. For many SQL examples we adopt the syntax of IBM DB2 [71] as it comes closest to SQL:2011. For the operators (such as temporal aggregation and temporal join) which cannot be expressed in SQL:2011 concisely, we use pseudo code.

3 System

In this section we describe the temporal database systems we evaluated in this work. These systems include:

SAP HANA. The main memory column store database system by SAP is our target system.

TimelineDB. We implemented our operators in a prototype for evaluating different design alternatives.

Other Systems. Several commercial database systems are available which support a limited subset of temporal operators. We used these systems as a baselines for our measurements.

3.1 The SAP HANA Database System

SAP HANA [24] is a commercial database system which employs both a column store and a row store for in-memory data processing.

3.1.1 Architecture of SAP HANA

SAP HANA was designed to exploit the properties of modern hardware such as multi-core systems and large main memories. Especially fast full column scans and customized operators as well as massive intra- and inter-operator parallelism contribute to its performance characteristics. Column stores are well suited for analytic queries on big amounts of data, which originally was the core business of SAP HANA. The system is able to handle both OLAP and OLTP workloads efficiently in one system.

For reducing the main memory consumption and improving query execution times, SAP HANA makes use of multiple compression schemes. To achieve high insert/update performance, updates and inserts are first applied to one or multiple non-compressed *delta stores* which are specifically tuned for high volumes of updates. SAP HANA periodically merges the new data from the delta stores into the *main store* which is tuned for efficient reads. In order to guarantee consistency, all operations (in particular queries) take delta and main stores into account.

SAP HANA is a distributed database system which allows the deployment of multiple servers for a single database. In our lab, the biggest installation so far consists of 250 nodes with 1 TB each, which sums up to 250 TB of main memory. With an average compression ratio of 5, this installation can load up to 1.25 PB of raw data. SAP HANA includes multiple engine types such as a text engine, a graph engine, an OLTP engine, and others. Concurrency control is implemented in the OLTP engine using Snapshot Isolation, which is an important prerequisite to implement clear semantics for temporal data management.

Basic support for temporal data (system-time) is already available natively in SAP HANA, but only a subset of possible operators are currently implemented. The most prominent examples for temporal data structures are the data store objects in the SAP Business Warehouse product (BW, DSO) and the so-called “change documents” in the SAP ERP system, where applications store previous versions of business objects.

3.1.2 Temporal Features of SAP HANA

Storage. To implement the system-time dimension, the SAP HANA database offers the *history table* [73]. 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 session-level timeslice is available in the current release version of SAP HANA. That is, the AS OF operator for timeslice can be used only globally in a query. An implementation which includes the combination of multiple timeslice operations in one query is described in the Master's thesis by Zala [86].

Time Dimensions. SAP 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 includes a timeslice 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. The timeslice operator 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.

3.1.3 SQL Extension

With SQL:2011 temporal features [54] were added to the SQL standard. Yet, the syntax is limited to the timeslice operator. More complex temporal operators such as temporal aggregation or temporal join are not supported. Even if it is possible to express and "simulate" these operators with SQL:2011, the resulting code is not concise, is error-prone and the resulting pattern is hard for the SQL optimizer to detect. For this reason we decided to define an extension to the standard SQL:2011 and submitted a draft to the SQL committee of SAP.

In the following, we sketch the SQL syntax for the temporal operators in SAP HANA in an informal way:

Temporal Aggregation. This temporal operator computes an aggregated value for each point in time a table has been updated with respect to a given time dimension. Thus, in a bitemporal database system, the aggregation can be computed for any time dimension, i.e., system-time or application-time. This 'grouping by time' is indicated by a *group by* clause including the name of the time dimension followed by brackets, as the time dimension is not a physical attribute. The temporal aggregation can be computed for a given aggregation function such as a cumulative aggregation like `SUM` and `COUNT` or a selective aggregation like `MIN` and `MAX`.

In the following example, the value of all unshipped orders is computed for each point in the application-time dimension `receivable_time` as it is currently known:

```
SELECT SUM(o.totalprice) AS total, o.receivable_time()
FROM orders o
WHERE o.orderstatus = 'O'
GROUP BY o.receivable_time()
```

Timeslice. For the timeslice operator we adopt the syntax introduced in SQL:2011 by means of the AS OF clause. For example, the following query selects the value of all unshipped orders that were visible in the database on December 31th, 2013:

```
SELECT SUM(o_totalprice) AS revenue, count(*)
FROM orders
FOR SYSTEM_TIME AS OF TIMESTAMP '2013-12-31 12:00:00'
WHERE o_orderstatus = 'O'
```

This syntax is also implemented in DB2 10.5 [71]. Yet, DB2 implements only the system-time and one application-time dimension. The name of the application-time is hard-coded as `BUSINESS_TIME`. With the SQL syntax for SAP HANA we allow the user to define an arbitrary number of time dimensions.

Temporal Join. The temporal join operator is not supported by the SQL:2011 standard. We therefore propose the following SQL extension:

A temporal join of two tables returns only tuples for which the visibility (or validity, respectively) interval overlaps, i.e., the tuples are visible at the same time. The temporal criterion is usually used in conjunction to a non-temporal join condition.

The following SQL query computes a temporal join of two tables:

```
SELECT COUNT(*)
FROM customer c TEMPORAL JOIN orders o
  ON c.visible_time() OVERLAPS o.active_time()
WHERE o_orderstatus = 'O' AND c_acctbal < 5000
   AND o_totalprice > 10
   AND c_custkey = o_custkey
```

In this example the `OVERLAPS` keyword indicates that the two time dimensions `visible_time` and `active_time` need to be valid at the same time for two tuples from the partner tables. For this work we consider the `OVERLAPS` condition only, but in theory any other comparison condition defined in [75] such as `MEETS`, `PRECEEDS`, `CONTAINS` can be used as a join criterion.

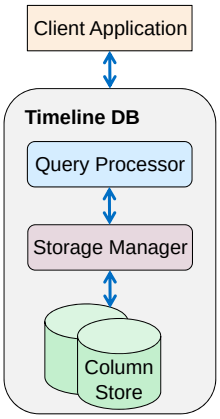


Figure 3.1: TimelineDB Architecture

3.2 Prototype: TimelineDB

For the evaluation of alternative implementations of temporal operators, we developed a prototype based on the architecture of SAP HANA. We designed the data structures and algorithms to fit the properties of modern hardware and with the goal to be implemented in the SAP HANA product. Given this perspective of a real system integration, all aspects of productive software like performance, memory consumption, parallelism, and complexity of algorithms had to be taken into account. Therefore, the data structures had to be simple, the memory overhead had to be low, incremental updates had to be supported, delta structures as well as fast index reconstruction had to be available.

Figure 3.1 depicts the architecture of the TimelineDB prototype. The TimelineDB prototype provides the basic functionality of a database system to a client application using a well-defined API. For example, the client can be part of a test framework to run benchmarks or a web-interface as it was used in [50]. The architecure of the prototype contains several layers: The physical storage layer is located at the lowest layer and currently includes a column store only.

The Storage Manager is located on top of the column store and encapsulates low-level access methods such as insert and scan operations. In addition, the Storage Manager offers a feature for bulk-loading data. Our TimelineDB prototype currently does not contain any implementation of data logging, transactions or recovery. The Query Processor accesses the Storage Manager and implements all temporal and non-temporal operations that are exposed to the client. All algorithms described in Section 6.3 and 7.3 of this dissertation are implemented within the Query Processor.

More information about the implementation details of the TimelineDB prototype are described in the Master's theses by Manjili [60] and Vagenas [82].

3.3 Other 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 features 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 only 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 4.5. We were able to get plausible results for four systems, which we refer to as Systems A to D in all experiments of this dissertation. Other database systems also provide 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.

3.3.1 Teradata

Architecture. Teradata processes temporal queries by compiling them into more generic, non-temporal operations [6]. 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.

Teradata implements the Temporal Statement Modifier approach presented in [15] by Böhlen et al., which describes an extension of an existing query language with temporal features.

SQL Syntax. As disscussed below, Teradata includes 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 with the type definition `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 features, i.e., table partitions or indexes can be defined in the same way as other columns.

Temporal Operators. The timeslice operator on system time is formulated via the syntax `TRANSACTIONTIME AS OF TIMESTAMP <date-or-timestamp>` and using `CURRENT TRANSACTIONTIME` before the query or after a table reference. Similarly, timeslice on application-time is supported by `VALIDTIME` instead of using

`TRANSACTIONTIME`. It is possible to specify the semantics of joins and DML operations as `SEQUENCED VALIDTIME` or `NON-SEQUENCED VALIDTIME` as defined in [74]. We did not find examples for a temporal aggregation operation as defined in [49].

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 [6]. As temporal queries are compiled into standard database operations, no specific query processing features for temporal queries seem to be exploited.

3.3.2 IBM DB2

Architecture. Only recently, IBM announced the introduction of temporal features in DB2, see [71] for an overview. In order to use system-time in a temporal table, one has to create a base table and a so called *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 comprises bitemporal tables: The application-time dimension 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 timeslice 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. An implementation of the temporal aggregation operator is not provided. Queries referencing a temporal table can use additional temporal predicates to filter for periods of the application-time. Following the SQL standard, timeslice on both time dimensions is possible to filter ranges of system or application-times. DML statements are based on the SEQUENCED model of Snodgrass [74], 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. Moreover, DB2 does not automatically create any index on the temporal table.

3.3.3 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 [68] 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 system-time was complemented by application-time.

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

Time Dimensions. As mentioned above, Oracle 12c includes the application-time dimension, which can be combined with Flashback to create a bitemporal table [65]. 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 timeslice 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 timeslice 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. The 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.

3.3.4 PostgreSQL

Architecture. The standard distribution of PostgreSQL does not provide a native implementation for temporal data beyond SQL:2003 features. However, patches [84, 20] for temporal features are available. Consequently, it is currently not possible to create, query and modify temporal tables in PostgreSQL.

Temporal Algorithms and Indexes. PostgreSQL implements the GiST data structure [33]. Therefore, 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 [32] or Quad-Tree [25], we are able to analyze this advanced index type without having to deal with the extension packs of the commercial alternatives.

System	SQL Syntax	Storage	Type	Sys.Time	Appl.Time
SAP HANA	proprietary	column	memory	yes	no
Oracle	proprietary	row	disk	yes	yes
DB2	SQL:2011	row	disk	yes	yes
Teradata	T-SQL2	row	disk	yes	yes
Postgres	non-temporal	row	disk	no	no

Table 3.1: Temporal Features of Commercial Database Systems

3.4 Concluding Remarks

In this chapter we described the database systems we compared in our experiments. SAP HANA is our target system in which we integrate our data structure and algorithms. SAP HANA is an in-memory column store which is optimized for very fast scan operations.

All other commercial systems we tested are disk based row stores. Even if the architecture of the systems is very different in many aspects, we can observe some commonalities: SAP HANA, Oracle and DB2 use a horizontal partitioning approach to separate current and previous versions of the data in order to ensure an efficient access to current tuples. All disk-based row stores make use of indexes to achieve a good performance.

For the evaluation of our algorithms we also use a protoype called TimelineDB, which is implemented based on the architecture of SAP HANA and allows for an efficient implementation and comparison of different design alternatives.

4

Benchmark for Bitemporal Databases

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 [54], 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 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 database (i.e., a certain system-time) but they also examine time intervals defined in the business domain (i.e., application-time). 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 [5] and [17]) 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 allow for temporal operations accessing previous states of the system. For evaluating the time dimension we define additional queries which retrieve data at different points in time.

The remainder of this chapter is structured as follows: In Section 4.1 we summarize the design goals for our proposed benchmark TPC-BiH. We survey related work on benchmarking temporal databases in Section 4.2. In the core part of the section (Section 4.3), we define the schema, the data generator for temporal data, and the queries comprising the benchmark. We analyze several systems that support temporal queries, and we present performance measurements for our benchmark (Section 4.5).

4.1 Goals and Methodology

The goal of this chapter 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 features, 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 for many different scenarios.

Our query workload provides a coverage of common temporal DB requirements. It covers operations such as *timeslice*, *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 [29], 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 correspond to the SQL:2011 standard.
- *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".

4.2 Related Work on Temporal Benchmarks

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 [17] 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 1993, a research proposal [21] 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 [41] 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 [83] 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 [5]. 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 a 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 [5] 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.

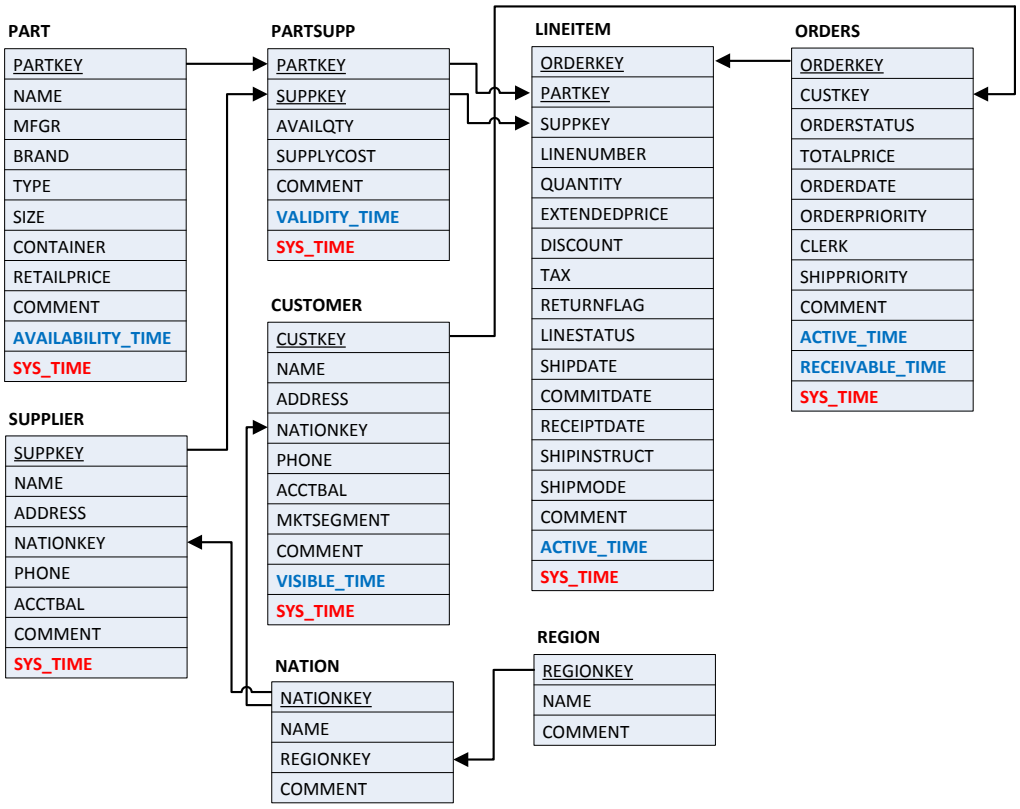


Figure 4.1: Schema

4.3 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. Thus, we describe representative examples in this chapter and refer to a technical report [46] which includes all queries and definitions in detail.

4.3.1 Schema

The schema we use in our benchmark is shown in Figure 4.1. 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 allow for 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.

4.3.2 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 [17]. We also used a similar approach (with additional timestamp assignment) in a previous version of the benchmark [49], 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 SF_0 for TPC-H) and the length of the history (called SF_H) separately, thus permitting cases like large initial data with a short history ($SF_0 \gg SF_H$), small initial data with a long history ($SF_0 \ll SF_H$) or any other combination. Similarly to the scaling settings in TPC-H, where $SF_0 = 1.0$ corresponds to 1 GB of data, we normalize $SF_H = 1.0$ to the same size, and use the same (linear) scaling rules.

Table 4.1 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 $SF_0 = SF_H$. As we can see, CUSTOMER and SUPPLIER get a fairly high number of history entries per tuple, while ORDERS and LINEITEM see proportionally fewer temporal operations. When taking the sizes of the initial relations into account, the bulk of temporal operations is still performed on LINEITEM and ORDER. A second aspect on which the tables differ is the kind of temporal 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 a bitemporal data model, as described in Chapter 2.2.

Table	**Growth Ratio**	**Dominant**	**Specialization**	**Overwrite App**
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 4.1: 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.

4.3.3 Queries

Given the multi-dimensional space of possible temporal query classification, we cluster the queries among common dimensions: Data access [59], temporal operators and specific temporal correlations.

Pure-Timeslice Queries.

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 *timeslice*, 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 timeslice 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 timeslice 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 timeslice operations on significantly different histories. T3 and T4 correlate data from two timeslice operations within the same table. Comparing their results with T2 (very selective) and T5 (entire history) gives an insight into whether any sharing of temporal 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 timeslice 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 4.3.1. 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 timeslice 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 [5] 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 4.3.2).

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 previous versions 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 in time. 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,

Name	Application-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 4.2: Bitemporal Dimension Queries

we also want to gain a more thorough understanding of queries stressing both time dimensions. Snodgrass [74] 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 4.2 describes the semantics of each query.

4.4 Implementation

In this section we describe the implementation of the TPC-BiH benchmark introduced in Section 4.3. 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 [43]. 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).

4.4.1 Benchmarking Framework

The goals we have defined for the Benchmarking Service require several aspects of conceptual underpinning. First we describe how a benchmark is modeled in our service. Next, we sketch the architecture of the system. We continue with a description of the web-based user interface of the benchmarking service and explain how a benchmark can be executed.

Modeling a Benchmark

Since our benchmarking service aims to combine flexibility with rich data operations and user guidance, a comprehensive and expressive model is required. The key benefit of this meta model is that artifacts (i.e., components of a benchmark) can be parameterized, stored and reused. The intuitive definition of these artifacts is achieved by a web-based UI, which also supports archiving and comparing results.

As visualized by the abstract data model (Figure 4.2), a *benchmark definition* is a combination of several artifact types: *Schema Definition, DDL Tuning, Data Generators, Database Servers, Query Set, Execution Order*. A simplified example of such a benchmark definition is ("TPC-H schema", "Index on L_SHIPDATE", "TPC-H dbgen: Scale 100", "SAP HANA, PostgreSQL", "Q1,Q5", "uniform mix"). In addition to general artifact selection, most of these artifacts can be parameterized.

Generally speaking, a *benchmark* can be seen as a subset of the cross-product of all the artifact types and parameters. Given the possibly large design space, we introduced two means of structuring: 1) *Templates* define the type of a benchmark. Examples of such templates include "a parameterized query on a server (one curve per parameter)" or "several grouped generator runs (one curve per server and query)". 2) *Measurements* are a grouping of artifacts along particular aspects

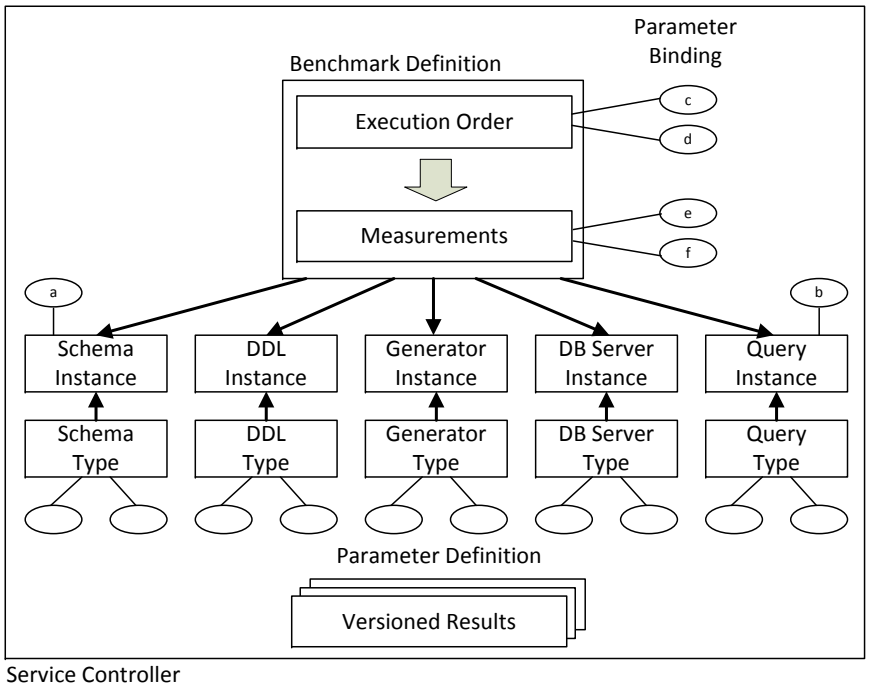


Figure 4.2: Data Model

(yielding, for instance, a line in a graph for a query, scaled over the database size). The known set of artifacts, possible parameters and templates provide information to the GUI to let the user intuitively design and run benchmarks.

The artifacts of a benchmark are described as follows:

Schema Definition. The aim of the schema meta model is to provide abstract information on the data model of individual benchmarks (such as TPC-H), in particular on tables, columns, data types and constraints. This information can be exploited in various ways, among them: 1) Generating DDL statements for creating tables (with metadata specific for a database server type) 2) Generating consistent data-preserving constraints and relationships. In terms of parameterization, we allow the user to choose which columns are being used for each experiment.

DDL Tuning. Beside the schema, there are many aspects in DDLs which can affect performance. We separate them from the schema, so as to provide more flexibility in benchmark design and execution. Typical "tuning" DDL aspects include index creation, materialized views and partitioning. Given the abstract modeling of the schema and the tuning, the system can create both combined and incremental DDL statements at different states within a running experiment.

Data Generator. A data generator can be applied before the execution of an SQL statement in order to populate the database instance with an experimental data set. Different types of data generators are supported and may be combined:

1. *Predefined generators* for common benchmarks (e.g., all the TPC benchmarks), including the parameters given in the benchmark specification.

2. *Generic user-defined generators*: a built-in generator using information from data definition and database server information, covering common aspects such as size, value distribution and correlation between the tables. Furthermore, referential integrity constraints and arbitrary join paths with a chosen selectivity can be defined. All these aspects are exposed as parameters.

3. *Custom generators*: Specific requirements can be expressed in the service as custom classes or by calling an external tool (such as [31]). Parameters of these tools need to be specified for the integration into the service.

Database Servers. Since our benchmarking service aims for a multitude of different database servers, the meta model needs to cover three aspects:

1. Capabilities of the database systems involved such as data types, DML expressions, etc. This information can be utilized to tailor DDL and DML statements.

2. Operational information on how to perform operations on the actual server instances using standard call-level interfaces like JDBC, e.g., establishing a connection, executing a query, interpreting the results, all of which will be relevant when running a benchmark.

3. Tunable parameters that are not reachable via normal DDL statements, such as the "merge interval" of SAP HANA or memory/disk settings of Oracle. Besides custom call-level statements, this may involve a collection of scripts at OS-level access.

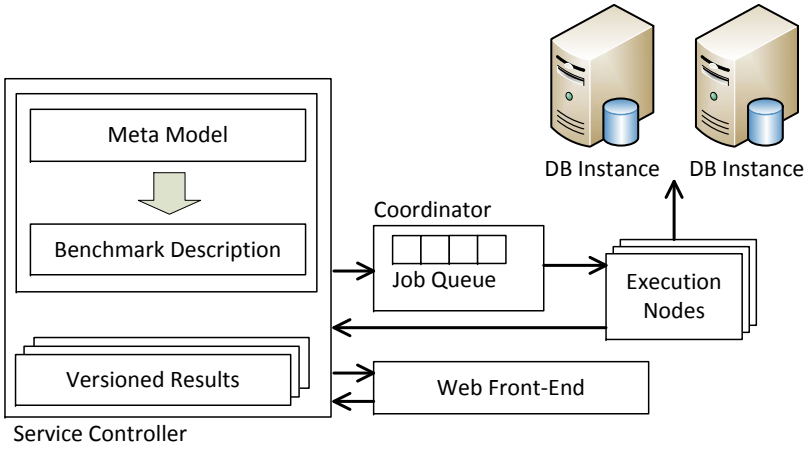


Figure 4.3: System Architecture of the Benchmarking Service

Query Set. The set of queries to be executed in an experiment can consist of arbitrary DML statements in their textual form. This includes standard SQL statements like queries, insert, update and delete operations, but also stored procedures or scripts in languages such as PL/SQL or T-SQL. Each statement has a possibly empty set of parameters (including type information) for input and output values, allowing for parameterized queries and reusing the output of one query as input for another. Depending on the specification, these parameters may be applied by text replacement or as invocation-time arguments.

Execution Order. Many benchmarks do not consider individual queries in isolation; instead, queries are combined at varying levels of complexity. The meta model of the benchmarking service provides two means to express such interactions: 1) For workloads that consider state changes explicitly, an ordering of the query set may be given. 2) For workloads which combine multiple queries with different cost or characteristics, a query mix can be specified. Once more, a built-in model and driver provide the means to define common aspects like the distribution of query types or their timing. Custom query mix drivers may be included to manage those requirements which are not expressible by standard settings.

The entire meta model (artifacts and benchmark specifications) as well as results are stored in a *versioned* database. With this versioning we can track how interactions among artifacts have developed. Furthermore, artifacts can have *variants*, e.g., custom queries for specific DBMSs if automatic tailoring from meta model data is not sufficient.

System Architecture

A distributed architecture, as shown in Figure 4.3, was chosen for the service: A central *Service Controller* keeps track of the meta model instances, which includes both the actual artifacts and the results. The process of running an experiment is controlled by a *Coordinator Node*, which contains a queue of benchmarks that are about to be executed, distributes jobs and detects node failures. The benchmarks are run on several *Execution Nodes* in parallel to simulate a multi-user workload or speed up measurements. Each execution node in turn may distribute the measurements over several database servers. Database servers can be accessed at different levels, mainly using call-level interfaces such as JDBC with queries and statements derived from workloads and DBMS metadata. And, when necessary, at the OS level using scripts to start/stop databases and perform external tuning. Clearly, more access rights provide more precise control of the execution.

The usage model assumes a benchmark cluster or a "private" cloud setting. Using it in a public cloud is possible, but problematic due to the legal and financial constraints of benchmarking (commercial) DBMS: Benchmarking results must not be published without explicit permissions by the vendors. Running DBMS instances in the cloud incurs additional licensing fees, while running DBMSs on customer premises and accessing them from the cloud is often prohibited for security reasons.

Web-Based User Interface

The service controller provides a web front-end for the definition of artifacts and visualization of results. This GUI leverages the powerful meta model introduced in Section 4.4.1 by exposing the various kinds of artifact types.

For the definition of a benchmark, the web front-end allows the user to combine artifacts and specify parameters. For instance, in the TPC-H data definition, the configuration of a benchmark is done in multiple steps. It includes the known DDL tuning options, the parameters for generating input data and the queries with their parameters. Once an experiment has finished, its results can be compared to similar experiments.

The web front-end provides a comprehensive access to features, models and results of the experiments. Yet, the system supports including custom code and classes for special problems such as specific parameter distributions or complex and state-dependent conditional execution orders.

Running Benchmarks

Benchmarking is inherently expensive, as it involves computation- and data-intensive tasks like running input data generators and loading the generated data into databases. Furthermore, our definition of a benchmark as a cross-product of its contributing artifacts and their parameters, provides great flexibility, but can possibly entail high cost. The benchmarking service contains several strategies to cope with these costs: Users can specify directly or implicitly (using a template) which execution flow to follow. We apply a number of optimizations: The sequence of steps can be modified to reuse previous, costly stages (like dataset creation or DB loading). In addition, the data generator performs caching and pipelining (depending on the setting) to reduce memory and/or CPU costs. Whenever possible, the controller distributes and parallelizes steps as to take advantage of available nodes. Correctness of the results and precision of measurements can be ensured. Within an experiment, measurements are performed on a "hot" database and repeated several times to achieve stable results. Users may specify reference results against which the output values of queries are to be compared.

4.4.2 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 SF_0 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, $SF_H * 1\text{Mio}$ update scenarios are executed and the data is updated in-memory.

The data generator keeps its state in a lightweight in-memory database and includes both the 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.

4.4.3 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.

4.5 Evaluation and Results

This section presents the results of the benchmark described in Section 4.3 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 system 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 [10].

4.5.1 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 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, application and 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.

4.5.2 Architecture Analysis

Our first evaluation consists of an analysis of how the individual systems store the temporal data physically; we derived this information from the documentation, the system catalogs, an analysis of query plans and the 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 history 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.
- System D stores all information in a single non-temporal table. All other sytems use horizontal paritioning to separate current from previous versions of the data.
- Besides B-Tree indexes, System D additionally includes indexes based on GiST [33].

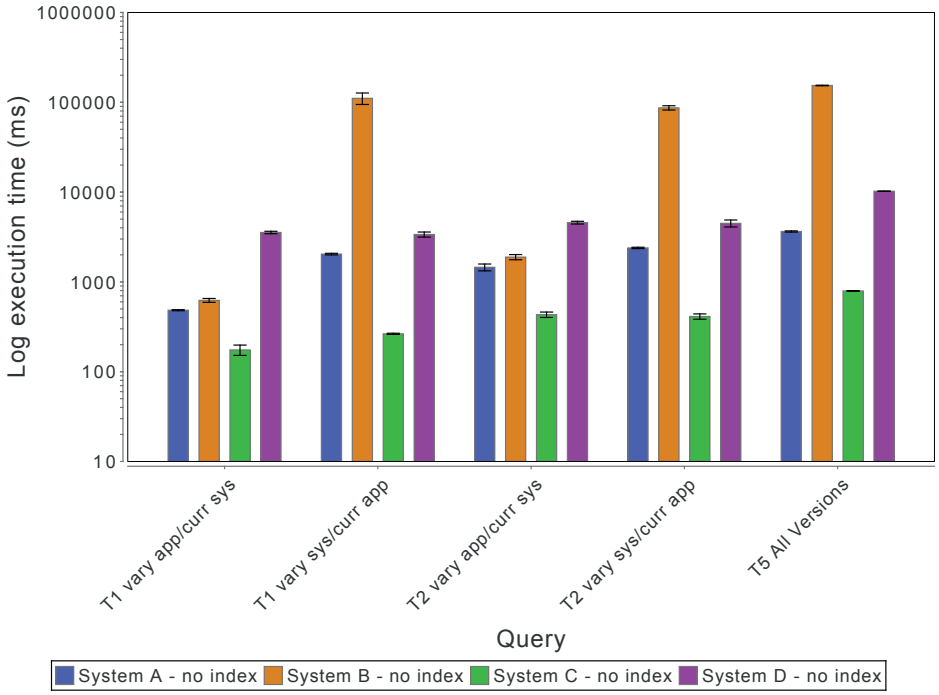


Figure 4.4: Basic Timeslice (Scaling 1.0/10.0)

4.5.3 Timeslice Operations

Point Timeslice

Our evaluation of temporal queries starts with timeslice, since this is the most commonly supported and used temporal operation. We utilize the full range of of queries specified in Section 4.3.3 to stress various aspects of timeslice.

Our first experiment focuses on point-point timeslice, 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 timeslice on a stable (non-growing current data) relation, namely PARTSUPP. T2 is a point-point timeslice 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 4.4 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 previous versionf of the data on System B. ALL is most expensive, since it not only needs to scan all tables, but also process all tuples.

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 4.5, 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

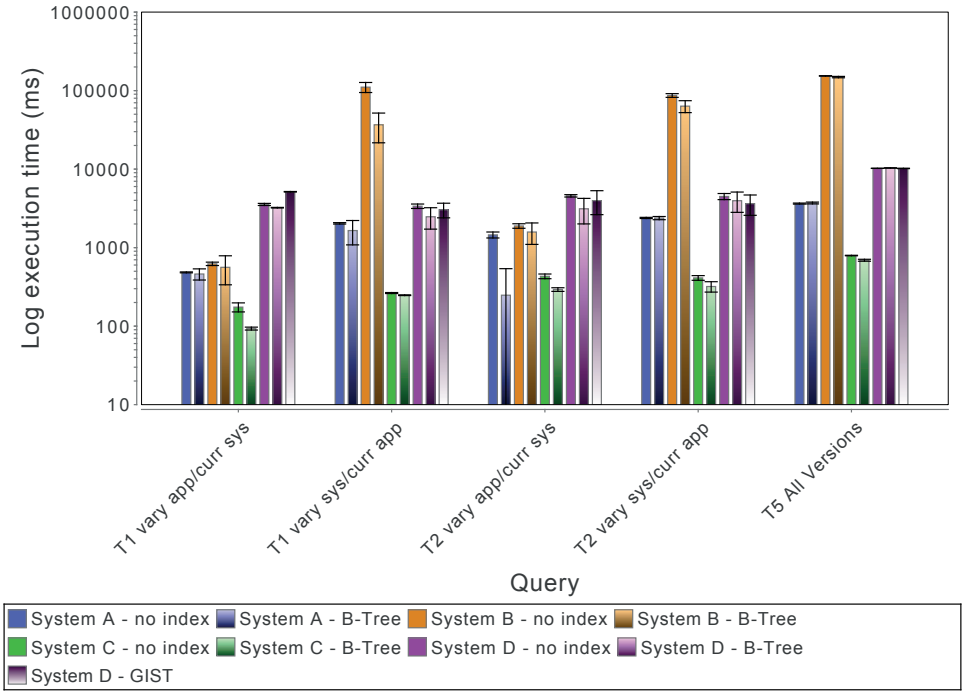


Figure 4.5: Index Impact for Basic Timeslice (Scaling 1.0/10.0)

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.

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 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 4.6 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

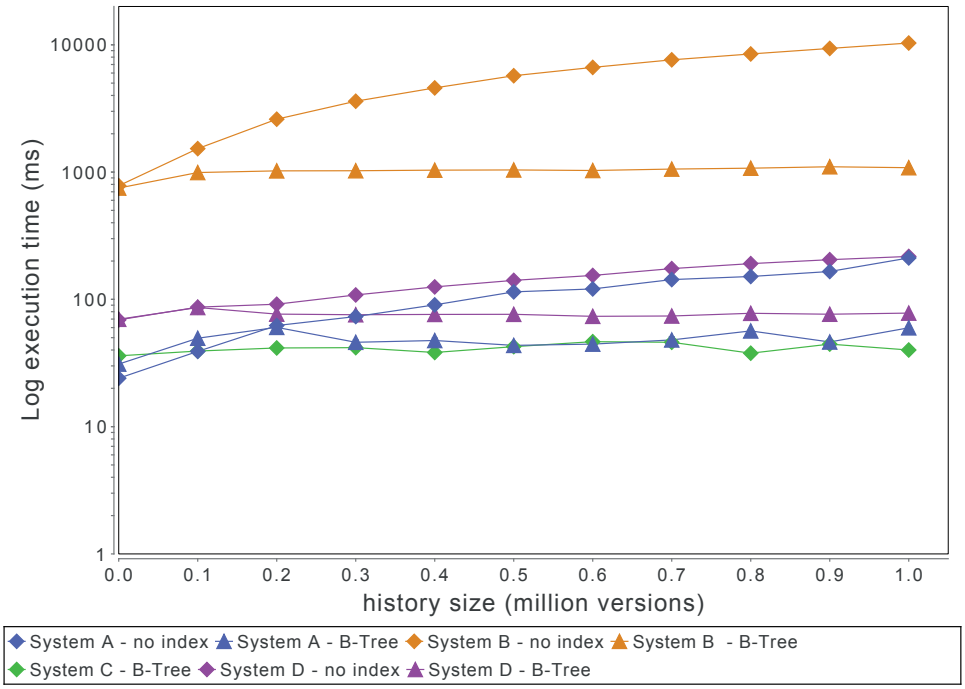


Figure 4.6: T1 for Variable History Size (Scaling 0.1/1.0)

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.

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 4.7 contains no results for application-time slicing for system B due to the bug mentioned in Section 4.5.3. 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 timeslice, in particular for System C due to somewhat lower complexity of the query.

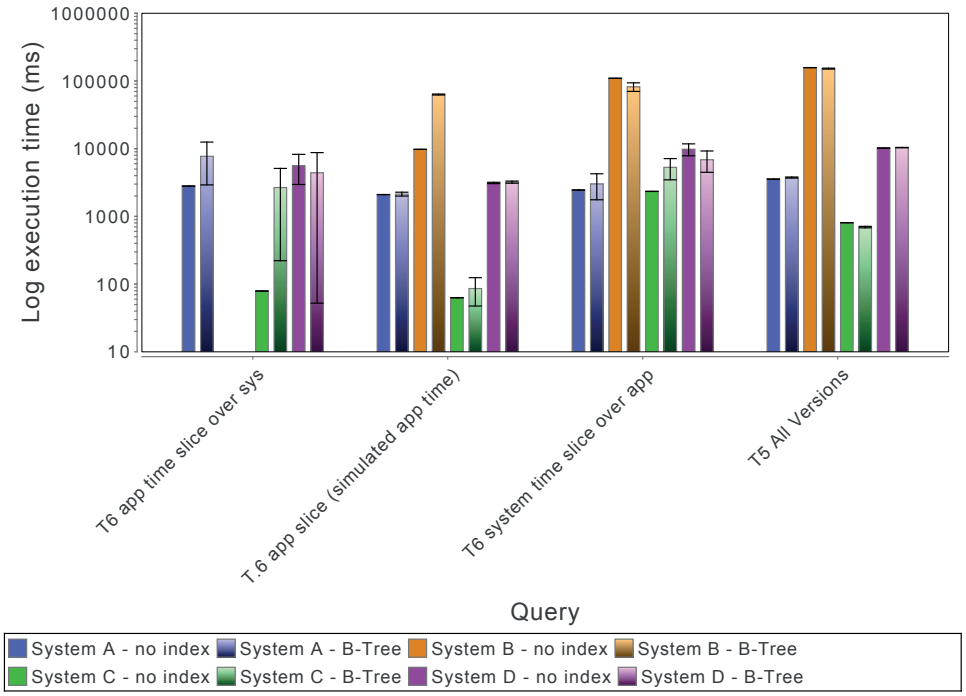


Figure 4.7: Temporal Slicing (Scaling 1.0/10.0)

Implicit vs. Explicit Timeslice

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 4.8 and the query plans show, all three system access the history table when using the explicit version as none of them recognizes this optimization.

4.5.4 Timeslice for Complex Analysis

Timeslice measurements so far focused on the behavior of individual timeslice 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

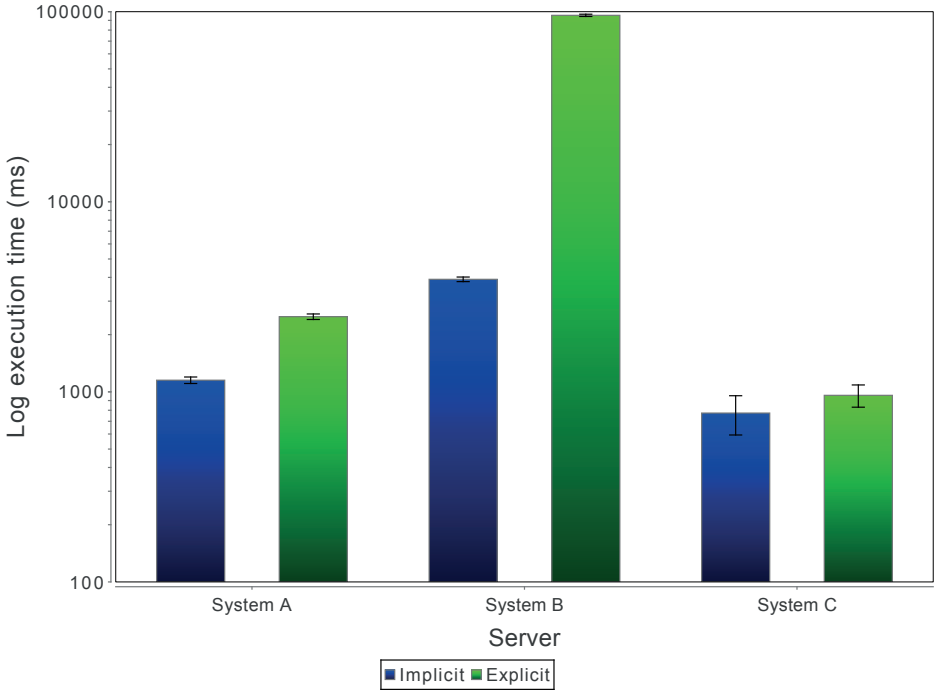


Figure 4.8: Current TT Implicit vs. Explicit (Scaling 1.0/10.0)

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 [10]) 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 resulted in 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 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.

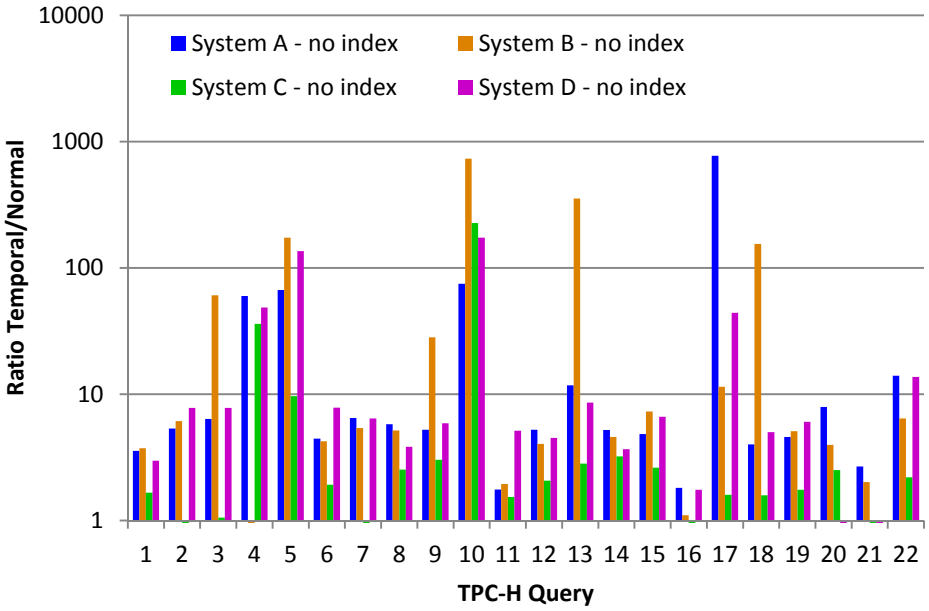


Figure 4.9: TPC-H with Application-Timeslice (Scaling 1.0/10.0)

Application-Timeslice 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 4.9 shows the slowdown factor between the queries on the non-temporal tables and the timeslice 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

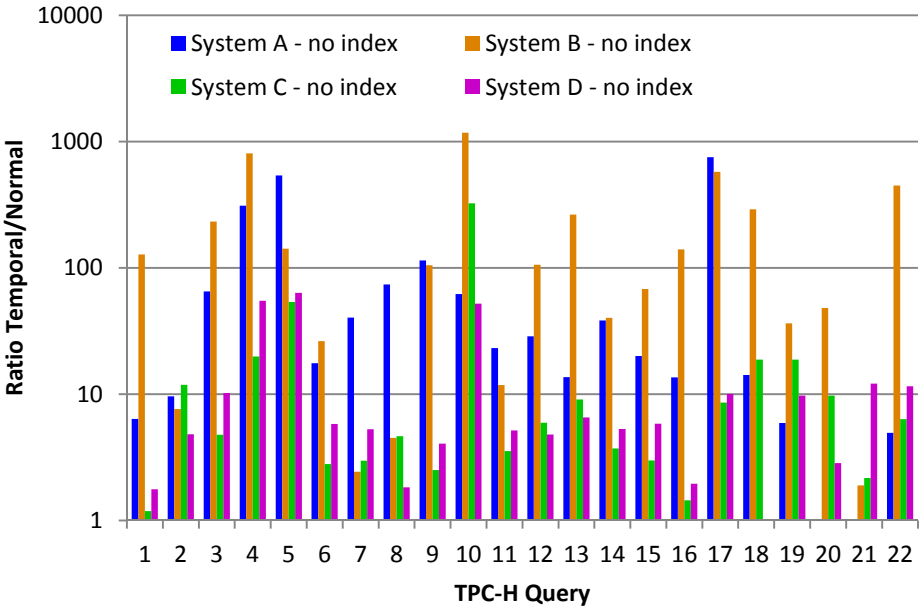


Figure 4.10: TPC-H with System-Timeslice (Scaling 1.0/10.0)

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.

System-Timeslice

Given our experience with significant performance overheads when accessing history tables (as in Section 4.5.3), our second experiment with complex analytical queries performs a timeslice 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 4.10 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-timeslice. 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.

4.5.5 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 4.3.3. 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.

Complete Time Range

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

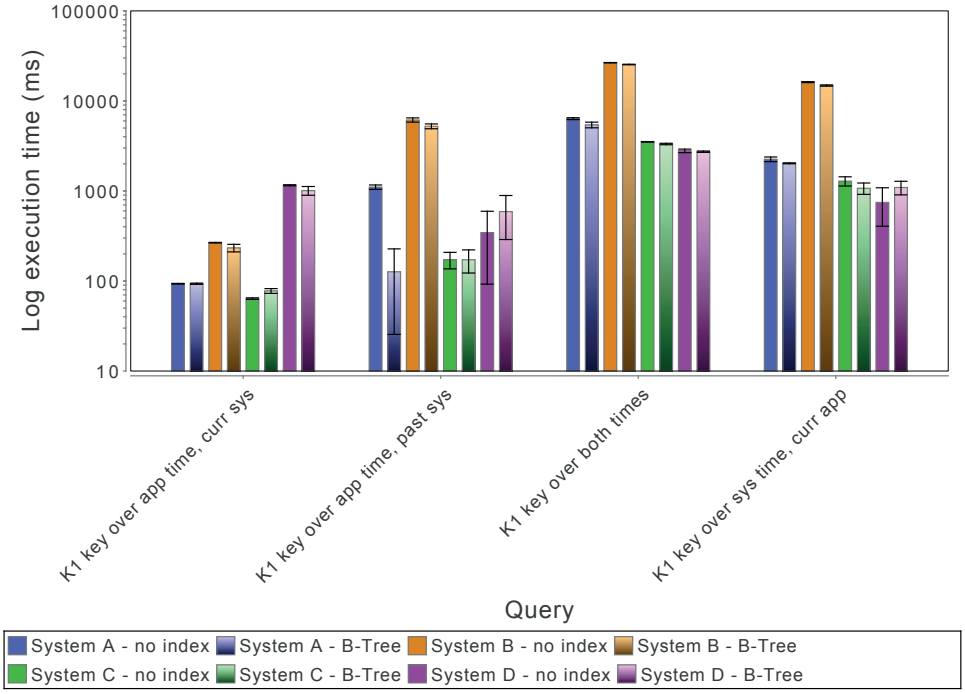


Figure 4.11: Key in Time - Full Range (Scaling 1.0/10.0)

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 4.11 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

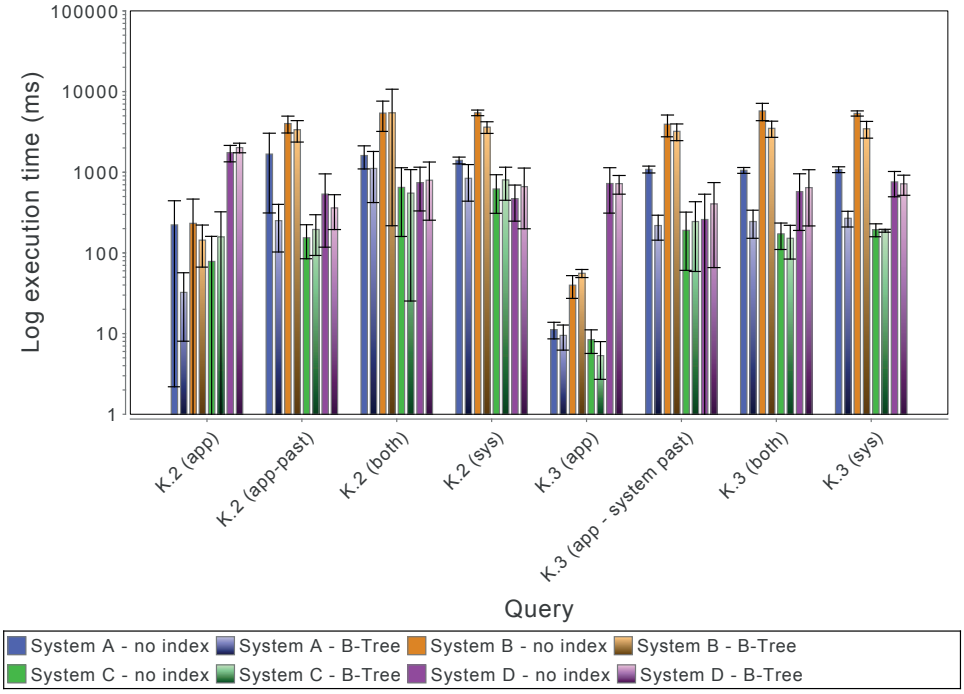


Figure 4.12: Key in Time - Time Restriction (Scaling 1.0/10.0)

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 accessing previous application-time versions at current system-time more expensive.

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 4.12 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 4.13 performs a complementary restriction, based not on time but on version count. As a result, we only consider only each individual time dimension, 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.

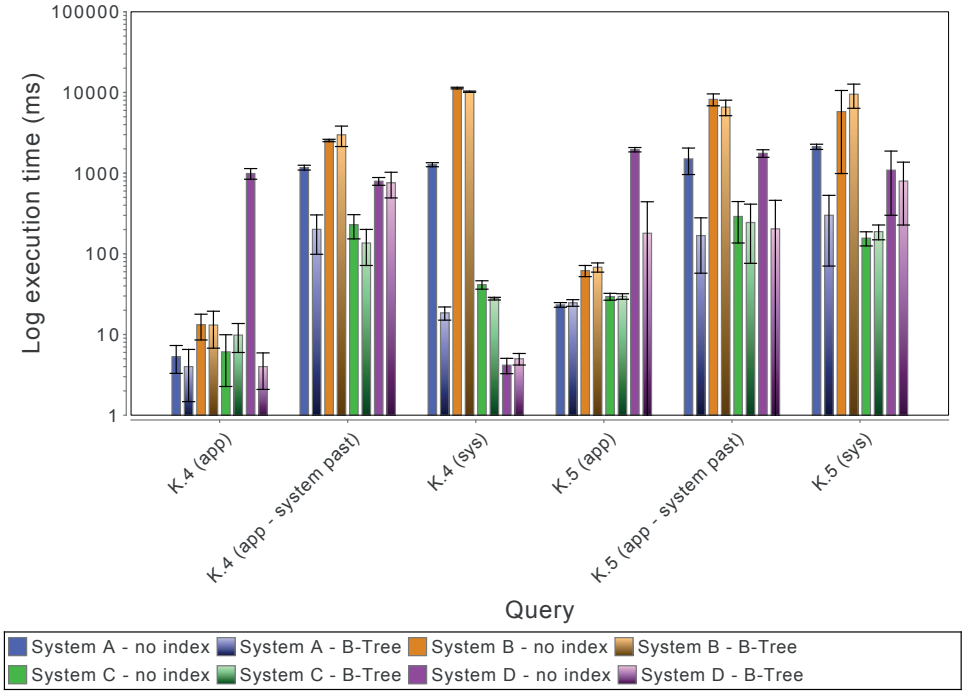


Figure 4.13: Key in Time - Version Restriction (Scaling 1.0/10.0)

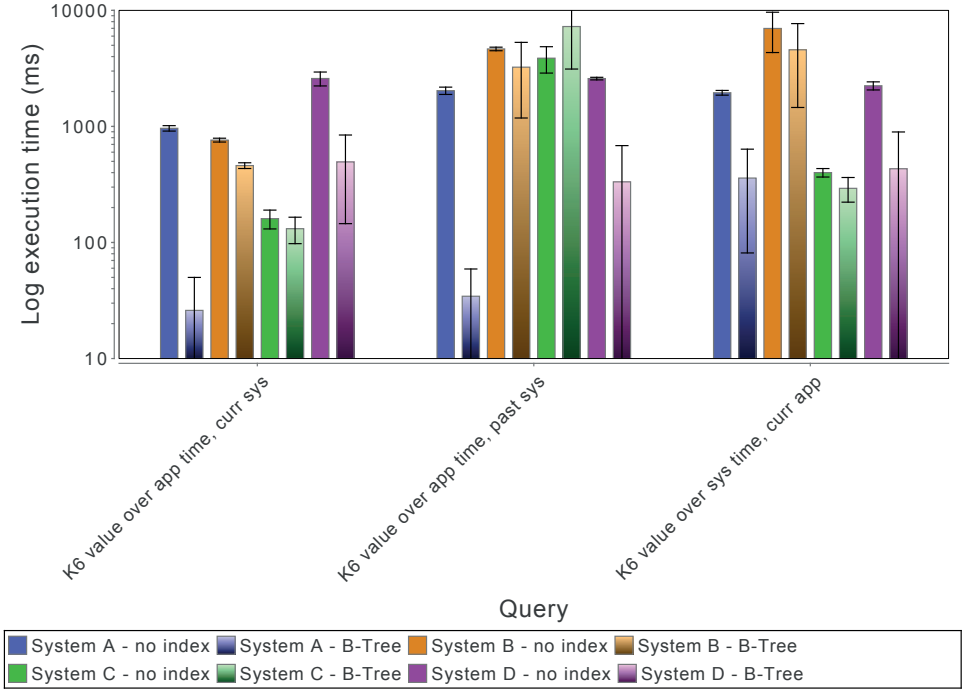


Figure 4.14: Value in Time (Scaling 1.0/10.0)

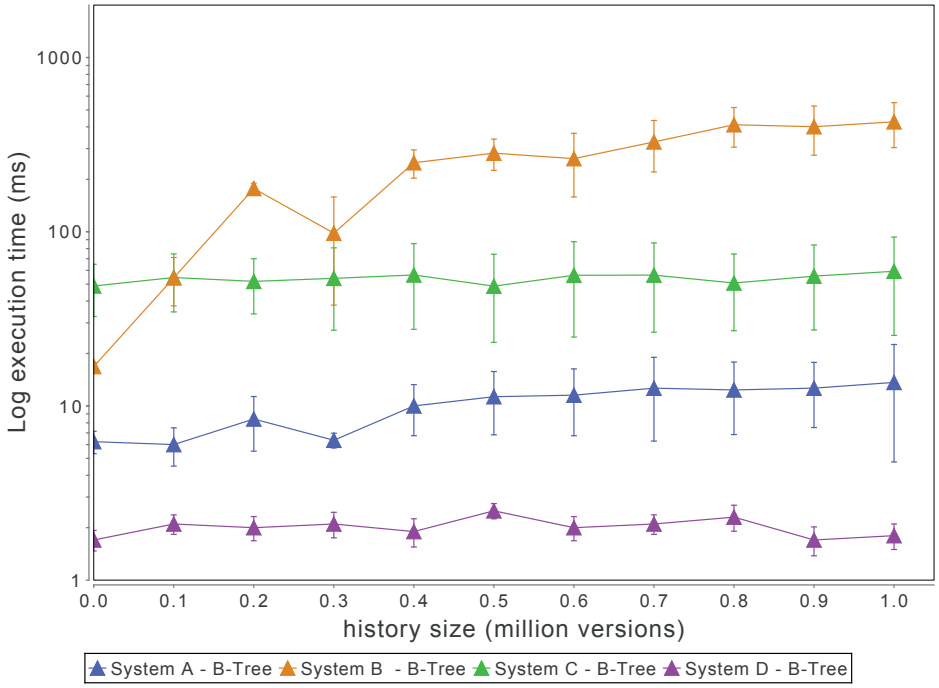


Figure 4.15: Key-Range Variable History Size (Scaling 0.1/1.0)

History of Non-Key Attributes

Beyond analyzing previous versions of tuples identified by their keys, we also investigated the cost of choosing tuples by non-key values. Figure 4.14 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.

Sensitivity Experiments

Similar to experiment in Section 4.5.3, we want to understand how data set changes affect the results. In Figure 4.15 we vary the history length for the query that investigates the application-time evolution at a fixed system-time (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 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

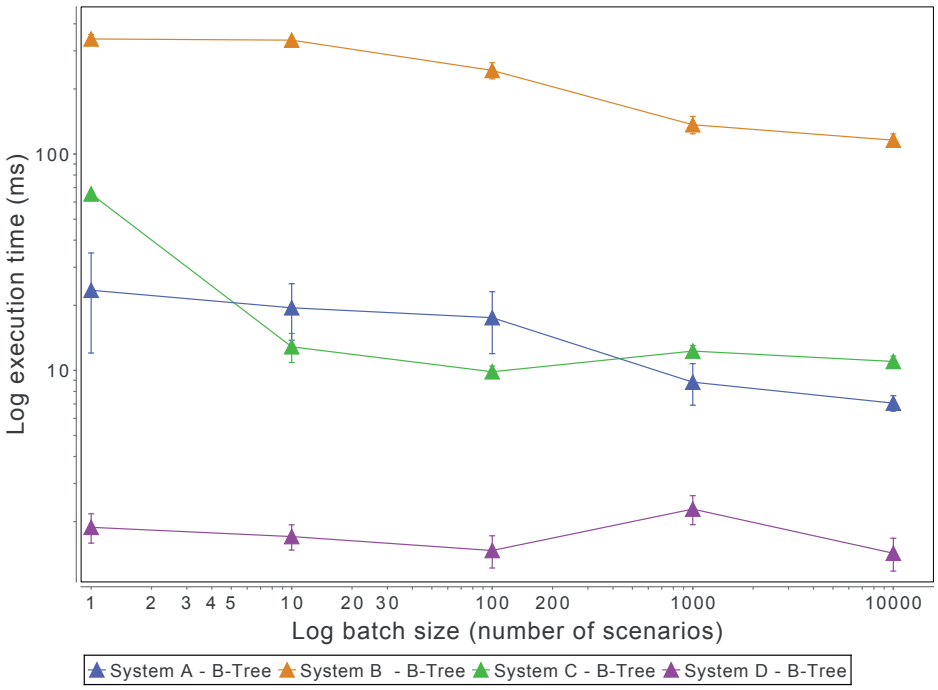


Figure 4.16: Key-Range for Variable Batch Size (Scaling 0.1/1.0)

are combined into a transactions as to understand if the number of transactions has an impact. As we can see in Figure 4.16, 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.

4.5.6 Range-Timeslice

For the application-oriented queries in range-timeslice, we notice that the cost can become very significant (see Figure 4.17). 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.

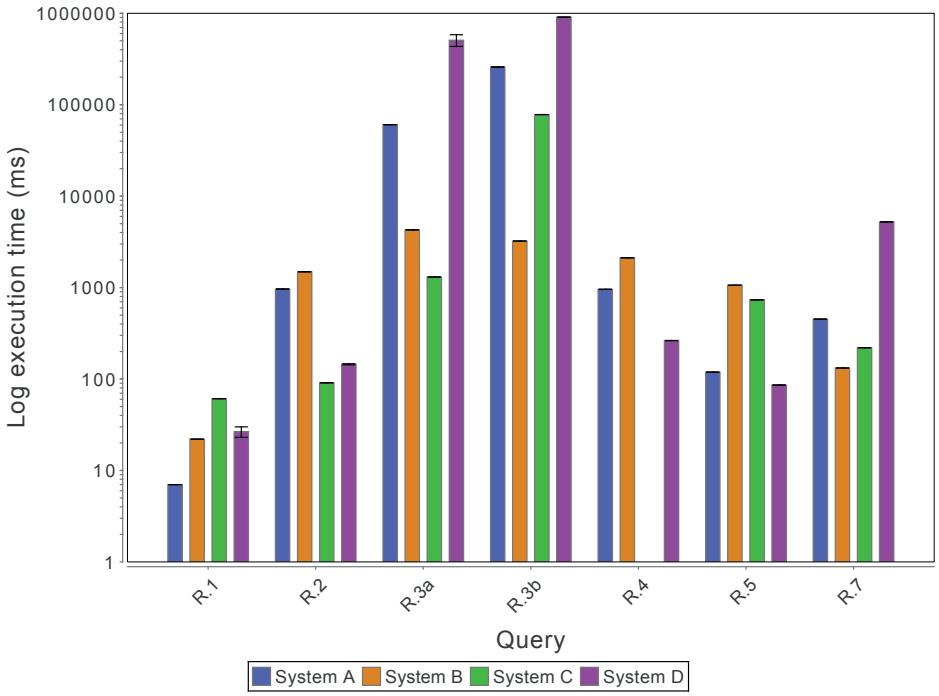


Figure 4.17: Range Timeslice (Scaling 0.01/0.1)

4.5.7 Bitemporal Coverage

We measured bitemporal coverage on the 1.0/10.0 data set. As shown in Figure 4.18, without indexes, most queries turn into table scans 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. The absence of any temporal join operators lead to rather very slow operations when performing correlations.

4.5.8 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 4.19 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 magnitude 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.

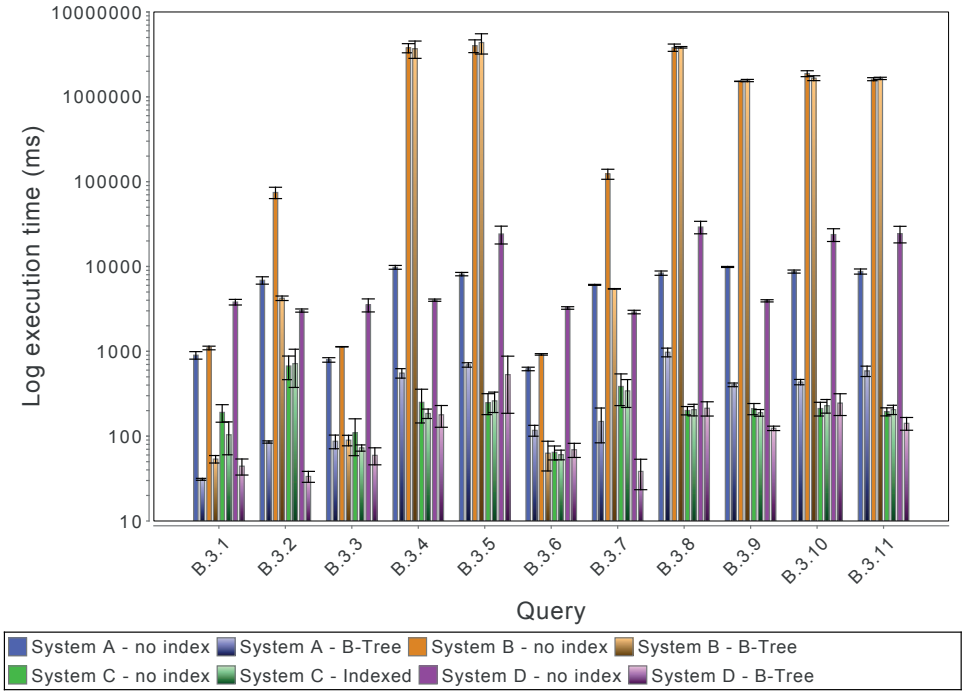


Figure 4.18: Bitemporal dimensions (Scaling 1.0/10.0)

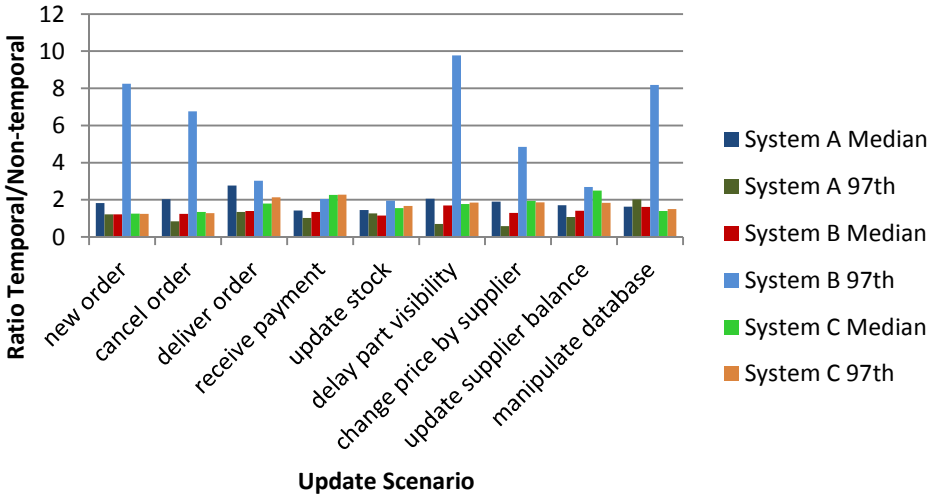


Figure 4.19: Loading Time per Scenario (Scaling 0.1/1.0)

4.5.9 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 those 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 remain 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.

4.6 Concluding Remarks

In this chapter, we presented a benchmark for bitemporal databases which builds on existing benchmarks and presents a comprehensive coverage of temporal access patterns and queries.

We performed a thorough analysis 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 management often performs worst in this area. The usefulness of tuning the systems with conventional indexes varies a lot and depends mostly on the overall selectivity of the queries. Other temporal operators that are not directly supported in the current language standards (such as temporal aggregation and temporal join) fare even worse.

Our experiments based on the TPC-BiH benchmark showed that state-of-the-art commercial database systems do not support complex temporal operators such as temporal join and temporal aggregation efficiently. As these systems do not contain any dedicated implementation of temporal operators, all operations are mapped to standard (non-temporal) data structures.

As shown by the experiments in this chapter, in-memory column stores outperform disk-based row stores for analytical workloads. This is the motivation for choosing an in-memory column store for the analysis of temporal data. In the next chapters we investigate how the performance of temporal operators can be improved by exploiting the properties of such a database system. To this end, in Chapter 5 we first evaluate alternative memory layouts for temporal data in main memory. In Chapters 6 and 7 we propose a unified index data structure for the system-time dimension and bitemporal data, and we describe efficient implementations of all important temporal operators based on this index.

5

Physical Storage of Temporal Data

In one of his last talks, Jim Gray postulated that "update in place" was dead [30]. Storage is becoming so abundant that it is cheaper to keep all data, rather than thinking about which data to delete. Instead of overwriting updated data, it is better to create a new version of the data.

There are a number of database products that support versioning. Correspondingly, these systems also allow the user to compose so-called *timeslice* queries that allow for the navigation to old versions of the data. Oracle has pioneered these ideas with its Flashback feature [68], which is integrated into the Oracle database product. Flashback extends SQL's `FROM` clause with an optional `AS OF` construct assigned to each table: `AS OF` specifies a version number or a timestamp that indicates which version of the table should be used. By default and in the absence of an `AS OF`, the latest version is accessed. In such a system, updates can only be applied to the latest version so that all previous versions are immutable. PostgreSQL had a similar feature based on the append-only design of the PostgreSQL storage manager [77]. ImmortalDB by Microsoft Research is a row store system that supports versioning and timeslice queries [56].

So far, most work on versioning and temporal queries has been carried out in the context of a row store. Lately, however, it has become clear in numerous studies [2, 34] that *column stores* outperform row stores. In particular, column stores show superior performance for read-mostly and OLAP workloads.

Temporal queries are particularly crucial in OLAP applications. For example, an analyst might be interested in the value of his portfolio today if he had left it unchanged since the beginning of the financial crisis in September 2008. This

query involves a timeslice to the state of the portfolio *as of* September 2008 and a reassessment of the value of that portfolio with current prices and stock quotes.

This chapter presents alternative approaches to implement versioning and temporal operators in a main memory column store. The work was motivated by the timeslice feature of the in-memory column store database system SAP HANA [24], which is designed to accelerate OLAP queries. The goal of this work was to find the best design for the timeslice component of this system.

Implementing versioning and temporal operators in a column store is not trivial. The state-of-the-art implementation of versioning in row stores is based on chaining the versions of a record using pointers [56]. If versions are held in the granularity of individual fields as part of a column store, then the storage overhead of keeping such pointers can be prohibitive. Furthermore, a lot of optimizations carried out for column stores are based on a predictable sequential access pattern while processing the data; this optimization may become less effective if pointers are chased.

Another issue is the organization of the column store. Typically, the relative positions of the attributes of a row are identical in all columns in order to make inner joins of columns within the same table fast. The question is how different versions of an attribute can be stored and which memory layout is most attractive for temporal queries.

The main contribution of this chapter is to study alternative approaches to represent temporal data physically in a main memory column store. We present three memory layouts which differ in the way they encode versioning information and how they cluster the data. The first approach clusters by *row* (as in traditional column stores). The second approach clusters by *version-ID*. The third approach is a *hybrid* between the first two approaches. For each layout, the basic data structures, query processing and update algorithms are shown. Furthermore, this chapter presents the results of a comprehensive performance study that assesses the tradeoffs of the alternative approaches and compares them to a state-of-the-art row store implementation. These experiments also give insight into the fundamental space-time tradeoffs of versioned column stores.

In this chapter we focus on the physical storage of temporal data and scan-based approaches rather than index data structures. Many traditional index data structures do not work well on modern hardware and in-memory database systems as they are designed for efficient operations on hard disks, optimizing the number of I/O operations for updates and queries. For instance, tree based approaches show a poor performance as they lead to a high synchronization overhead on many-core systems and contention of the memory.

For this chapter we assume that all temporal data fits into main memory, which is legitimate because a real world main memory column store such as SAP HANA can be operated in a distributed environment. For instance, the biggest installation

in our lab so far (including several hundred nodes) supports up to 1 PB of raw data.

The remainder of this chapter is structured as follows: Section 5.1 discusses related work. Section 5.2 presents use cases which are relevant for accessing previous versions of the data. Section 5.3 sketches which update granularities can be implemented in a column store. Sections 5.4 to 5.6 describe the three alternative approaches to implement temporal operator in column stores. Section 5.7 discusses the results of the performance experiments. Section 5.8 contains conclusions and possible avenues for future work.

5.1 Related Work

Temporal and versioned databases have been subject to extensive research. A survey on the fundamental work on temporal databases is given by [66]. More recent related work and an overview of indexing techniques on temporal databases is presented in [57].

Management of temporal data has been implemented in well known DBMS: [77] describes the implementation of an archive in *PostgreSQL*.

Oracle provides a similar feature called Flashback [68, 39] that allows going back in time. This Flashback feature comes in different variants: a) short time row level restore using UNDO information, b) restoring deleted tables using a recycle bin and c) restoring a state of the whole database by storing previous images of entire data blocks. A newer variant of Flashback introduced in version 11g is called Flashback Data Archive and stores the entire data augmented with the required meta-data in a dedicated archive using background processes.

Another database system that manages temporal data is *ImmortalDB* [56, 57, 58]. It is built into a row-oriented system and timestamps the data in the granularity of records. Their chained representation of records (i.e., every record has a pointer to the next older version of itself) does not work in column stores as the overhead of pointers is significantly higher.

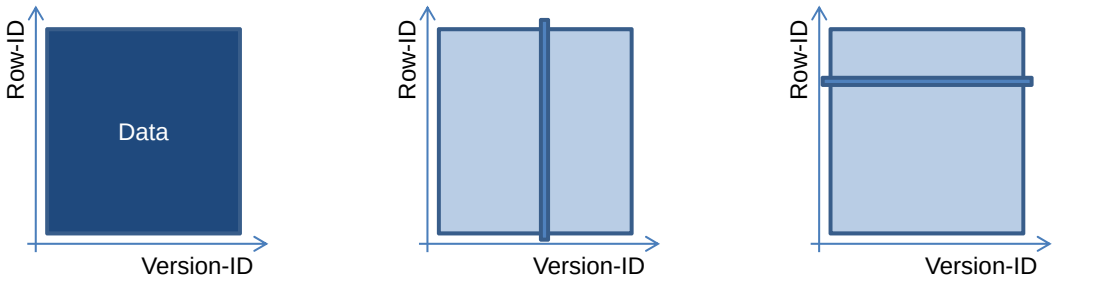
C-Store has support for versioning using a multi-version storage that enables snapshot isolation and the concept of a writable and read-optimized storage [78]. However, this feature is limited in C-Store to short-term timeslice for the implementation of snapshot isolation. C-Store does not support timeslice queries.

Vertica [35] is the commercial successor of C-Store and (similarly to C-Store) provides only very limited versioning features by means of snapshot isolation. In the area of processing temporal data there have been surveys like [59] and [38] which present the foundations of temporal data management and access methods. As in [59], we will discuss physical layouts where the data is clustered by *row* (called key-only in [59]), by *version-ID* (called time-only) and one approach that combines the two (called time-key).

We investigate the area of processing temporal data in the context of column stores: The idea of storing data in columns instead of rows dates back to [18]. The advantage is clear: Only the required columns are brought to the CPU via the memory hierarchy. However, the data needs to be reconstructed from the different columns in order to return the resulting row. This can usually be done quite efficiently if the data in the different columns is stored in the same order. Adding temporal data to a column store imposes the following design problem: Either we supplement the storage with data that is not required to answer the query (if we insert an entry in every column for each update), or we cannot have simple offset

access to all columns as they might have received a different number of updates. This leads to the question to what extent versioning affects the advantages of a column store. To the best of our knowledge, this question has not been answered yet.

Furthermore, the high similarity of adjacently located data in a column store can be exploited for compression, which both reduces memory consumption and improves query performance. An overview of different compression schemes in column stores is presented in [1].



(a) Whole Database (b) Slice at a Point in Time (c) Evolution of a Certain Row

Figure 5.1: Different Dimensions of a Relation with Temporal Data

5.2 Use Cases

In this section we describe the use cases we are considering when designing our implementation. In this chapter we investigate the low-level representation of the temporal data in main memory only and neglect complex operations such as temporal join or temporal aggregation. We therefore simplify the use cases introduced in Section 2.1 for this chapter.

There are basically two dimensions relevant to a relation that contains temporal data: the time dimension (i.e., slice the relation to show the state at a given point in time) and the row dimension (i.e., slice the relation to show the changes made to a certain row) as shown in Figure 5.1. In addition, combinations and aggregations are possible. The data can be clustered along no more than one of these two dimensions. Depending on this decision, different costs have to be paid for different access patterns. The use cases presented in this section are selected to expose these tradeoffs.

In general, a *version-ID* represents a unique system-time timestamp in the database. In the remainder of this chapter, we will focus on system-time only. For simplicity, we will not distinguish a *version-ID* from a date in real world. We will explain the use cases and our proposed memory layouts in examples based on versioned tables from the TPC-H schema.

5.2.1 Timeslice

One application of temporal data is the possibility to "travel" in time: The recording of temporal data enables the user to see the database at a certain point in time. An example query could be: "What was the maximum ordered quantity in all *lineitems* at the end of last year?" This can be formulated as an SQL query. The SQL syntax is along the lines of the temporal features of SQL:2011 [54]:

```
SELECT MAX(l_quantity)
FROM lineitem
FOR SYSTEM_TIME AS OF '2012-12-31'
```

This use case corresponds to the timeslice operator which has been introduced in Section 2.3.2.

5.2.2 Evolution of Data (Audit)

The other application of temporal data slices the data along the other dimension, meaning that we query the changes of a specific value over time. An example for such a query is "What was the maximum quantity of a specific *lineitem* over the last five years?" This type of query is important to satisfy audit requirements (e.g. showing that the data was always within a certain range). `BETWEEN` returns a row for each version of the specified data item. The attributes *linenumber* and *orderkey* are the compound primary key for *lineitem*.

```
SELECT MAX(l_quantity)
FROM lineitem
FOR SYSTEM_TIME
  BETWEEN '2008-01-01' AND '2012-12-31'
WHERE l_linenumber='3' AND l_orderkey = '1'
```

In contrast to the temporal aggregation operator which has been desribed in Section 2.3.1, only a single aggregated value is computed as a result of this query rather than one aggregate per each point in time.

5.2.3 Record Reconstruction

Since accessing multiple attributes is different in row and column stores, we consider a query which returns the value of different attributes of a table at a certain time. In addition, a condition is defined. Thus, only a subset of all rows which existed in the database at that time is retrieved. The following SQL code gives an example of such a query:

```
SELECT availqty, supplycost
FROM partsupp
FOR SYSTEM_TIME AS OF '2012-12-01'
WHERE suppkey < 10
```

5.2.4 Processing Inserts and Updates

There are three relevant additional use cases which are related to changing the information that is stored in the database. An *insert* operation adds an additional record (e.g., a new *lineitem* record). An *update* modifies an existing record and adds a new version without removing the old information from the storage. The *delete* operation is special in this scenario since the database is required to keep track of the history of the deleted rows in order to answer queries about the past consistently.

With the objective of supporting the use cases shown in this section, Sections 5.4, 5.5 and 5.6 present different approaches to store a table in main memory with versioning-support in the granularity of a single column (attribute). Our design space for the memory layouts contains several dimensions. First, the data can be clustered either by row or by version. Second, replication of data improves query response time, but introduces a tradeoff between query execution time, update costs and memory consumption. Third, depending on the storage layout, different compression methods can be applied. In addition, dictionary encoding [1] and dictionary compression [11] are general compression methods which work both in row and column stores. Compression not only reduces the consumed amount of memory but can also improve the execution time of queries which are executed over compressed data [1]. Furthermore, in case of archiving (moving old versions to harddisk), compressed data reduces the required space on harddisk and increases the speed of transferring data from main memory to disk and vice versa.

5.3 Update Granularity

This section investigates in which granularity updates of single attributes can be applied to a versioned table. As in such a temporal table an update is implemented as an insert of a new version, the question is whether the entire affected row should be stored as a new version or if only the modified attributes should be preserved.

5.3.1 Asynchronous Columns

In this approach, updates are only applied to the columns where the value has changed. Thus, the relative position of values for a given row and version is independent in different columns. For example, if in a row of the customer table only the address is updated, a new version is only written to the address column, and the other columns are not affected. The *asynchronous columns* approach is described in [40] and referred to as Temporal Decomposition Storage Model (TDSM).

The advantage of *asynchronous columns* is the efficient memory consumption. Because no data has to be replicated, read operations are fast for single columns. In addition, updates can be executed very quickly by simply inserting a new version.

On the negative side, performance of tuple reconstruction decreases with the number of columns which have to be joined. Since different columns have different sizes (due to the different number of updates modifying them), it is not trivial how to efficiently find the corresponding value for a row in all columns.

5.3.2 Synchronous Columns

In this approach, each version of a row is stored at the same relative position *synchronously* in all columns. In the case of an update, the previous value is replicated for unchanged columns, and therefore the relative position of a value for a given row and version is identical in all columns, which results in an efficient tuple reconstruction. However, space consumption increases due to the replication of data and redundant storage of unchanged attributes of a tuple. The *synchronous columns* approach has been chosen in most commercial systems so far.

Very fast record reconstruction is the main advantage of synchronized columns. This benefit can be achieved by preserving the same relative position of all values for a given row and version in all columns. In addition, the *version-ID* has to be stored only once for one tuple as all columns are always updated simultaneously.

On the negative side, *synchronous columns* lead to an increased update execution time and memory consumption due to replication of data. However, compression can be applied; this benefits from the high similarity of values in columns which are not updated frequently.

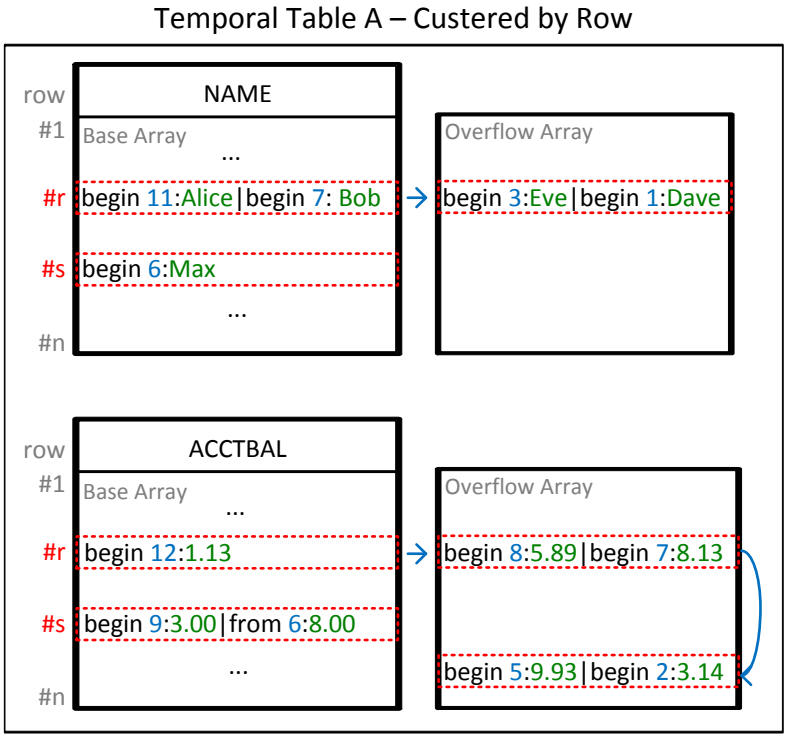


Figure 5.2: Clustering by Row with 2 Versions per Row Segment

5.4 Clustering by Row

This section introduces a memory layout in which the data is clustered by *row-ID*.

5.4.1 Storage Layout

In the *clustering by row* approach, space for a fixed number of versions is reserved for each row. The memory layout contains a *base array* of *segments*. Each position in the *base array* corresponds to a row in the table. A *segment* contains $width_{row}$ pairs of (val_{im}, ver_m) as a payload rather than an atomic value as it is the case in a traditional column store. val_{im} is the value of row i which has been valid since version ver_m.

If the number of updates of one row in the *base array* exceeds $width_{row}$, the data of the *segment* is copied to the next available position in an *overflow array* and a reference is stored. Within this *overflow array*, the *segments* of each row are chained and referenced by their array position.

In the example shown in Figure 5.2 we consider a versioned *customer* table with two attributes. If the account balance of customer s decreases to '\$3.00' at version number '9' a new (val_{sm}, ver_m) pair with val_{sm}='3.00' and ver_m='9' is prepended

to the segment of this customer. The former value is moved to the next available position in this *segment*.

5.4.2 Query and Update Processing

In this subsection we describe the operators needed to support the use cases which were introduced in Section 5.2.4. We show how these operators can be implemented efficiently for this layout. First, we describe operators to alter the stored data. Next, we will continue with operators to retrieve data from memory.

Insert. For insertion of a new row to a column, we append a *segment* to the *base array*. If the current number of *segments* exceeds the maximum number of rows $MaxSize_{byrow}$, space for a new column with size $2 * MaxSize_{byrow}$ has to be allocated and the data from the old column is copied. Next, a new *segment* is appended and the new (val_{im}, ver_m) pair is written to the leftmost position of the *segment*.

Update. As already shown in the motivation of this chapter, in temporal tables old versions are never modified. Therefore, an update operation in a temporal database can be translated to inserting a new version.

If there is space for a new version in the corresponding *segment* of a row, the previous (val, ver) pair is moved from the leftmost position to the next unoccupied position within the *segment* and the new pair is written to the leftmost position. Thus, the highest version in each segment is always on the left, and the remaining pairs are in ascending order starting from the second position. By this means, shifting all previous versions can be prevented. If the *segment* is full, the content of this *segment* has to be copied to an available position in the *overflow array*. The position within the *overflow array* is used as a reference to chain the *segments* as shown in Figure 5.2. Thus, references to previous *segments* never have to be updated because the position within the *overflow array* remains unchanged. Again, allocated space for the *overflow array* is doubled if the maximum size $MaxOv$ is exceeded.

In the *clustering by row* layout, the implementation of *synchronous columns* is difficult because for referencing (val, ver) pairs, it is necessary to add the information if the pair is located in the *base* or *overflow array*.

Delete Operation. For simplification, we assume that a deleted row is never re-inserted again with the same ID. In order to keep track of deleted rows, we choose a similar approach as the one presented in [78]. We introduce a bitmap in which a *true* bit at position i indicates that row i has been deleted. In this case, the last entry is a dummy update used to keep track of the version in which this row was deleted.

Next, to efficiently access data in memory, we define a set of scan operators retrieving values which fulfill a given condition. Optimal performance of each scan operator can be achieved by exploiting the clustering characteristics of the underlying layout. In the following, we describe how the value for a specific version can be retrieved. We will continue with describing how an aggregation over a subset of versions for a row can be implemented.

Select Value for a Given Version and Row-ID. This operator retrieves a single value for a given row i and version ver. In this layout, the data is primarily clustered by row and secondarily clustered by version. Thus, the position of the row has to be determined in the array of *segments* first, which can easily be achieved by a simple array-lookup based on *row-ID* i. In a second step, the value which is valid for a given version ver has to be retrieved by sequentially scanning the (val_{im}, ver_m) pairs. If ver in located within an overflow *segment*, preceding *segments* can be skipped by looking at the smallest version per *segment* first. A value val_{im} is valid for ver, if $ver_m \leq ver$ and no other ver_q exists with $ver_m < ver_q \leq ver$.

Algorithm 1 Cluster by Row: Get Value for ID and Version

```
function GetValue(id, version, base, overflow)
    segment ← base[id]                          ▷ get segment for the given ID
    repeat
        if segment.pair[0].version <= version then
            return segment.pair[0].value
        end if
        for i = colWidth − 1; i >= 1; i − − do
            if segment.pair[i].version <= version then
                return segment.pair[i].value
            end if
        end for
        segment ← overflow[segment.nextOverflow]
    until segment == empty
    return NOT_FOUND
end function
```

The pseudo code in Algorithm 1 shows the algorithm to access the value for a given ID and version. Note that always the first element in the segment shows the latest version, whereas other versions are stored in increasing order (in order to prevent shifting the array while prepending).

Select Value for a Given Version and a Group of Rows. This operator repeats the operation mentioned above for all rows in a given group.

Aggregation over a Version-Interval for a Single Row. This operation reads the values of a given row for all versions in a time interval $[ver_{start}, ver_{stop}]$ and calculates a single aggregated value. In a first step, the *row segment* for the given *row-ID* has to be located. Secondly, all values for versions within the time interval have to be read in the *segment* to calculate the aggregated result. For this purpose, all *segments* connected to this row in the *overflow array* have to be traversed successively. Scanning (val, ver) pairs can be stopped when it holds $ver < ver_{stop}$ as ver decreases steadily for one row.

5.4.3 Uncompressed Memory Consumption

Let $size(T)$ be the size of data type T (e.g., 4 bytes for Integer), $size(ver)$ the size of the version number (e.g., 8 bytes for Long), $size(pos)$ (e.g., 4 bytes for Unsigned Integer) the size of position of the next segment in the array. Then the size of a *segment* is given by

$$size_{segment} = width_{row} * (size(ver) + size(T)) + size(pos)$$

Correspondingly, the total size of a column with $MaxSize_{byrow}$ rows and $MaxOv$ overflow segments is

$$size_{total,byrow} = (MaxSize_{byrow} + MaxOv) * size_{segment}$$

The most important parameter for this layout is the width of a segment $width_{row}$. On the one hand, a larger number of versions per *segment* provides faster access to temporal data because fewer jumps in memory are required. On the other hand, a lot of memory is wasted if only a few rows are updated frequently. For the experiments, 10 versions per *segment* were chosen as a reasonable compromise.

5.4.4 Archiving

An archive allows storing parts of the table on harddisk, e.g., in case not all data fits in main memory. To create an archive, a version $ver_{archive}$ is chosen as a threshold. All versions which are older than $ver_{archive}$ are stored on harddisk, newer versions are kept in main-memory. Archiving can be implemented for the *clustering by row* approach by moving all *segments* to harddisk for which the validity interval of all (val, ver) pairs is strictly smaller than $ver_{archive}$. Yet, the segment containing the value valid at $ver_{archive}$ must reside in the main memory to reconstruct the value for this version without accessing the harddisk. The space of the released *segments* within the *overflow array* can be re-used for future *overflow segments*.

5.4.5 Compression

The *clustering by row* layout is a compact representation of updates per row. Therefore, it is hard to achieve an additional reduction of memory consumption. However, dictionary encoding [1] and dictionary compression [11] can be applied to exploit the characteristics of the stored values.

5.4.6 Discussion

The *clustering by row* layout performs best for queries which access a large number of versions of the same row. This is the case for the evolution of data use case in subsection 5.2.2. It is, however, expensive to retrieve a very early version of a row if it has been updated many times because a large number of positions of overflow pages has to be accessed. Memory consumption is optimal only if all *segments* are fully occupied, which is the case when the number of updates per row corresponds to the width of a *segment*. A lot of space is wasted if rows are never updated.

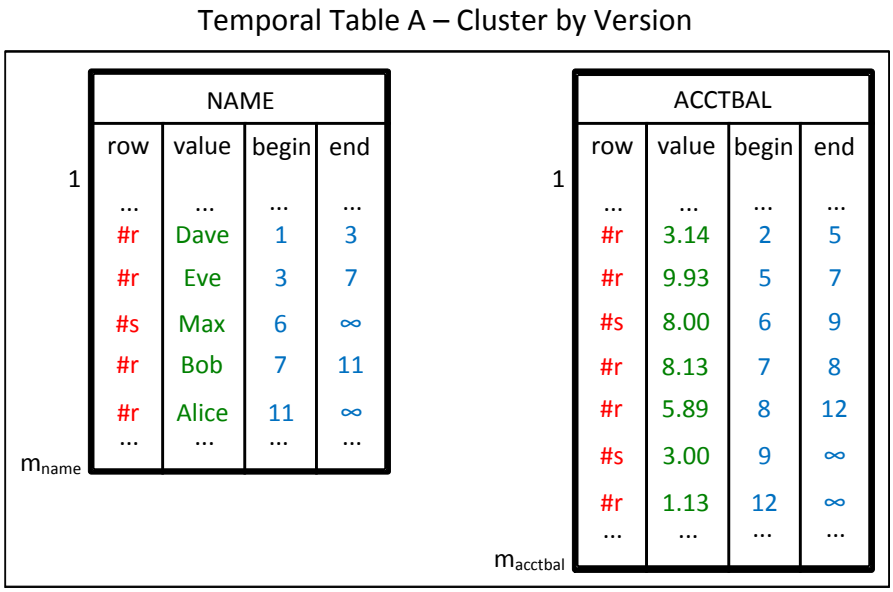


Figure 5.3: Clustering by Version

5.5 Clustering by Version

In this layout the data is clustered by insertion-order, i.e., by *version-ID*.

5.5.1 Storage Layout

In the *clustering by version* approach visualized by Figure 5.3, for each version of a row four values are stored in an array: The *row-ID* i, the value val and a version interval given by the version ver_{begin} for which this value becomes valid and the version ver_{end} when it is invalidated. The version interval simplifies determining if a value is valid for a given version without having to scan all data to check if it has been invalidated within another update.

For example, the fact that the customer with *row-ID* #r had a balance of '$8.13' from '7' to '8' can be represented by $\langle r, 8.13 \| 7 - 8 \rangle$.

5.5.2 Query and Update Processing

Insert. In the *clustering by version* approach, if a new tuple with *row-ID* i is inserted at version ver, the tuple $\langle i, val \| ver - \infty \rangle$ is appended to the array. If the number of tuples in the column exceeds its maximum size $MaxSize_{byversion}$, space is doubled and values are copied as in the previous approach.

Update. As we have to store a version interval for each update, the end interval of the previous version of this row has to be set first. Finding the latest version can be done in constant time by looking up the position in a *latest version array*. This array of size $count_{row}$ stores the position of the latest tuple for each *row-ID*. An alternative to the *latest version array* is a backwards-scan to retrieve the latest value which is valid for a given *row-ID*.

In the next step, the new version is appended similarly to the insert operation described in the previous section. This results in all tuples being sorted by ver_{begin}.

The *clustering by version* layout can support both the *asynchronous* and *synchronous columns* update granularities introduced in Section 5.3, because it is possible to reference a tuple efficiently by its array position.

Delete Operation. We give two alternative implementations of the delete operation.

First, the latest version can be invalidated with ver_{end} being set to the deletion time and no new tuple being inserted. Second, a bitmap marking deleted rows can be kept within the *latest version array*.

Select Value for a Given Version and Row-ID. As the data is clustered by version in this layout, the operator is implemented as a scan with the version as the primary search criteria. In a first step, the position has to be found for which the values are valid with respect to the given version *ver*. This is the case when it holds $ver_{begin} \leq ver < ver_{end}$. Next, for each valid tuple the ID has to be compared to the given *row-ID* and the value is read and returned as a result when the ID matches. Note that a backward scan would be more efficient if *ver* was closer to the latest version.

Algorithm 2 shows a scan for the retrieval of a given version and *row-ID*. The matching value was found with the occurrence of the first tuple for which the *row-ID* is equal to the given *row-ID* and the requested version is in the interval of versions when the value is valid for this row. The scan direction is chosen heuristically: a backward scan is performed when the given version is closer to the maximum available version $version_{max}$.

The latest version can be found more efficiently with the first occurrence of a given *row-ID* i in a backwards-scan. Alternatively, the position of the latest value for each *row-ID* can be retrieved in constant time from the *latest version array* without having to scan the whole column.

Select Value for a Given Version and a Group of Rows. In contrast to the previous layout, in this layout there is no need to repeat the above operation for each row. One scan is enough to retrieve values for a group of rows.

Algorithm 2 Cluster by Version: Get Value for ID and Version

```
function GetValue(id, version, data)
    if version < version_max/2 then                    ▷ do a forward scan
        for i = 0; i < data.size; i++ do
            if data.begin[i] <= version < data.end[i] then
                if data.id[i] == id then
                    return data.val[i]
                end if
            end if
        end for
    else                                               ▷ do a backward scan
        for i = data.size − 1; i >= 0; i−− do
            if data.begin[i] <= version < data.end[i] then
                if data.id[i] == id then
                    return data.val[i]
                end if
            end if
        end for
    end if
    return NOT_FOUND
end function
```

Aggregation over a Version-Interval for a Single Row. This operation has to be implemented with a linear table scan. For each tuple, the ID is compared to the given *row-ID i*. If the IDs are equal, the time intervals are compared, the corresponding value is read, and the aggregation can be calculated. As the data is sorted by ver_{begin}, the calculation can be aborted when it holds $ver_{stop} < ver_{begin}$.

5.5.3 Uncompressed Memory Consumption

For this memory layout, the size of one $(rowID, val, ver_{begin}, ver_{end})$ tuple can be calculated by

$$size_{tuple} = size(rowID) + size(T) + 2 * size(ver)$$

The total size of the column is

$$size_{total,byversion} = MaxSize_{byversion} * size_{tuple}$$

The total memory consumption $size_{total,byrow}$ of the *clustering by row* approach is higher than $size_{total,byversion}$ if a lot of *segments* are left unoccupied. This depends on the workload and number of updates for each row.

5.5.4 Archiving

In order to archive previous versions on harddisk which are older than $ver_{archive}$, a scan of the column is necessary. A tuple can be moved from the column to disk if its validity interval is completed before $ver_{archive}$ which is fulfilled when it holds $ver_{end} < ver_{archive}$. After transferring the old tuples to the harddisk, the column has to be rewritten in order to free the memory of the tuples which have been moved to the archive.

5.5.5 Compression

In this layout the representation of data is very similar to a traditional column store layout. Therefore, almost all the compression schemes for column stores presented in [1] can be applied for this layout as well.

5.5.6 Discussion

In the *clustering by version* approach, both timeslice and evolution of data queries are expected to be expensive for a large number of updates because a lot of tuples have to be scanned. Yet, insert, update and delete operations are simple look-ups and appends and therefore very efficient (constant time).

5.6 Hybrid

This section describes a layout representing a hybrid approach with two different types of clustering. The goal of this layout is to limit the amount of data which needs to be scanned to retrieve a given version.

5.6.1 Storage Layout

The layout of the *hybrid* approach illustrated by Figure 5.4 is similar to *clustering by version* layout in Section 5.5, but it includes additional *checkpoints*, each containing the latest version for all rows at the time that the checkpoint has been computed.

Again, if a row with ID i is inserted or updated at version ver, the tuple $\langle i, val \| ver - \infty \rangle$ is appended to a data structure called *delta array* according to the *clustering by version* approach in Section 5.5. The tuples are therefore clustered by version. After a fixed number of updates (defined by the *checkpoint interval* parameter $delta_{max}$) a consistent view of the entire column for the current version is serialized and stored in a checkpoint. In such a checkpoint, the value val and the latest version ver are stored for each row. The ID of a row is represented implicitly by the position in the checkpoint. Hence, the data within a checkpoint is clustered by row.

For keeping track of the versions for which a checkpoint is available, an index is introduced. By means of this information, the last checkpoint before a given *version-ID* can be determined efficiently in $O(log(s))$ with s being the number of checkpoints for this column. As the current version is accessed most frequently, a checkpoint of the latest version of each row is always maintained and called *current checkpoint*. When the current version of a row is updated, the old value is removed from the *current checkpoint* and appended to the delta array. The new value is now written to *current checkpoint*.

For this layout, $delta_{max}$ is the most important parameter which defines the maximum number of tuples to be stored before a full checkpoint is computed. A discussion on how to choose this parameter can be found in Section 5.7.7.

5.6.2 Query and Update Processing

Insert. In the hybrid approach, the insert operation is a simple append to the *current checkpoint*. If, due to insertion, the size of the table exceeds the maximum size of the *current checkpoint*, the data will be copied into an array with double size. For the sake of simplicity, this case is not considered when calculating the memory consumption.

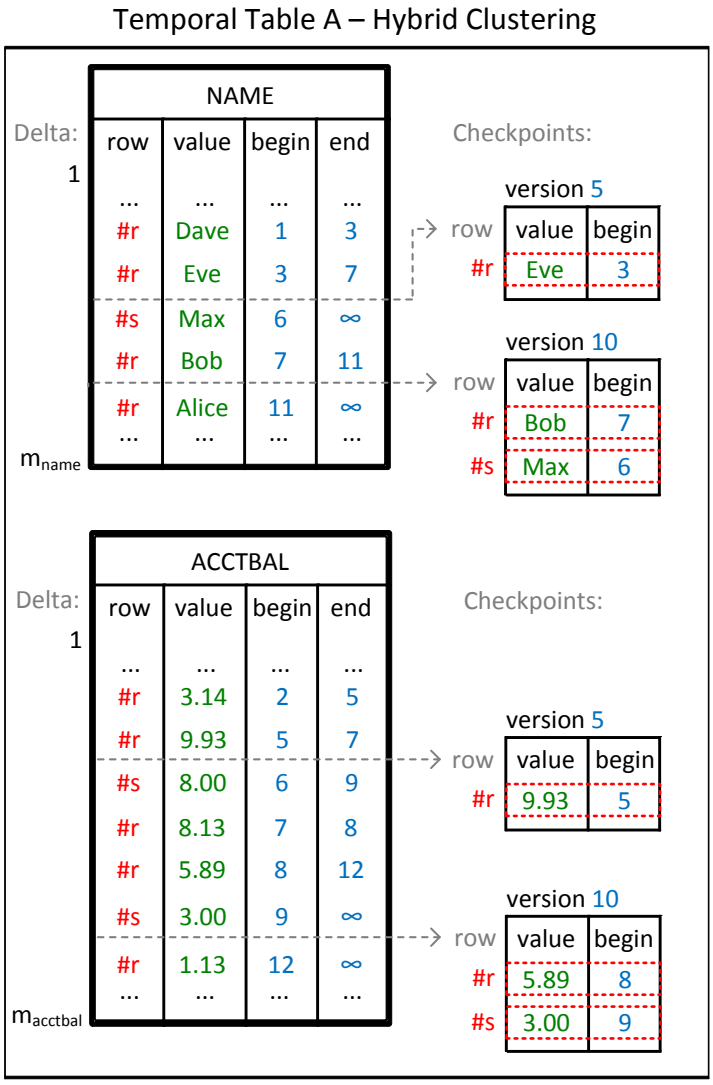


Figure 5.4: Hybrid Layout with 2 Checkpoints

Update. Before the new value is written to the *current segment*, the previous value of the updated row has to be written to the *delta array*. If the number of updates exceeds the limit defined by $delta_{max}$, a new checkpoint is built and a full serialization is written to a newly allocated checkpoint. This can be achieved efficiently by a copy operation from *current* to the new checkpoint. The position and the *version-ID* that lead to a full serialization are appended to the *checkpoints index* to keep track of the position of all *checkpoints*.

In the *hybrid* layout, similarly to the *clustering by version* approach, both *asynchronous columns* and *synchronous columns* update granularities are feasible. In addition, the synchronization of columns can be achieved based on checkpoints, this limits the number of tuples that have to be scanned, to retrieve the value in each column.

Delete Operation. The implementation of the delete operation is similar to the *clustering by version* approach described in Section 5.5.2.

Select Value for a Given Version and Row-ID. For retrieving the value for a given *row-ID* i and version ver, both the *clustering by row* and *by version* can be exploited in the *hybrid* approach. First, the position of the latest previous checkpoint is retrieved using the *checkpoints index* by means of a binary search in $O(log(s))$. The value for the given *row-ID* can be retrieved from this checkpoint by a simple array lookup. Next, the *delta array* is scanned as long as $ver_{begin} \leq ver$ to check if the row has been updated.

Algorithm 3 Hybrid: Get Value for ID and Version

```
function GetValue(id, version, delta, checkpoints, index)
    indexPos ← binarySearch(index.versions, version)
    checkptOffset ← index.checkptOffsets[indexPos]
    deltaPosition ← index.deltaPositions[indexPos]
    value ← checkpoints[checkptOffset + id]
    i ← deltaPosition                          ▷ read delta as long as version is valid
    while i < delta.size and delta.begin[i] <= version do
        if delta.id[i] == id then
            value ← delta.val[i]
        end if
        i ← i + 1
    end while
    return value
end function
```

In Algorithm 3, first a pointer to the nearest checkpoint older or equal to the given version is retrieved by applying a binary search in the *index*. This search

retrieves the position of the largest checkpoint smaller or equal to the given version. If the given version is equal to the version at which the checkpoint was built, the value for the *row* can be obtained by a single array-lookup. Otherwise, the delta has to be read additionally until the version at the current position becomes larger than the requested version.

As in the *hybrid* layout, the current version is always contained in a checkpoint. This layout provides the fastest possible access to the latest version by means of a simple array lookup.

Select Value for a Given Version and a Group of Rows. Similar to the approach mentioned above, values for a group of rows can be selected within one single scan.

Aggregation over a Version-Interval for a Single Row. As the aggregation has to be computed for a time interval $[ver_{start}, ver_{stop}]$, the index can be exploited to find the latest checkpoint before ver_{start}. Next, the table has to be scanned in a similar way as described in Section 5.5 as long as the current version is in the time interval.

5.6.3 Uncompressed Memory Consumption

In this layout, memory consumption is calculated similarly to the *clustering by version* approach. In addition, the space for checkpoints and their index has to be considered. The size of each checkpoint can be calculated by:

$$size_{checkpoint} = size(pos) + count_{row} * (size(T) + size(ver))$$

Where *pos* is the latest position in the *delta array* at the time of construction of the checkpoint. The *checkpoint index* is an array constituted of pointers to checkpoints and the versions at which the checkpoints were built. So the index size is:

$$size_{index} = s * (size(ptr) + size(ver))$$

For the sake of simplicity we assume a constant size for all checkpoints. Finally, the total memory consumption is derived by the following formula:

$$size_{total,hybrid} = size_{total,byversion} + s * (size_{checkpoint}) + size_{index}$$

5.6.4 Archiving

For storing old versions on harddisk, a checkpoint taken at $ver_{archive}$ is chosen as a threshold. All previous checkpoints and tuples in the *delta array* before that checkpoint are moved on harddisk and deleted from main memory. Again, the checkpoint at $ver_{archive}$ has to be preserved in main memory to allow the reconstruction of the values of all rows. This method prevents the execution of a full table scan operation as described in Section 5.5.

5.6.5 Compression

The *hybrid* layout can be understood as a *clustering by version* approach with additional checkpoints. Therefore, the compression of the *clustering by version* layout can be applied as described in Section 5.5. In addition, checkpoints can be compressed by exploiting the similarity of adjacent checkpoints. We consider a specific number of checkpoints as reference checkpoints and the rest as intermediate ones. Each intermediate checkpoint can now be represented based on the differences compared to its previous reference. Such a representation results in a sparse matrix, which can be compressed by means of the Yale format [81]. This representation both leads to a reduced memory consumption and a fast reconstruction of checkpoints by only one reference comparison. Again, dictionary encoding [1] and dictionary compression [11] can be applied in addition.

5.6.6 Discussion

The advantage of the *hybrid* layout is the speed up for timeslice queries for a given row. The execution time of this query is limited and shorter for smaller checkpoint intervals. However, this involves a time-space tradeoff because memory consumption increases for a larger number of checkpoints.

5.7 Experiments and Results

This section describes the results of performance experiments motivated by use cases from SAP (introcuded in Sections 2.1 and 5.2) involving analytical queries on large data-warehouses containing temporal data. In our experiments, we will study two metrics: memory consumption and the response time of queries and update operations.

5.7.1 Software and Hardware Used

The experiments were measured on a server with 24 GB RAM and one Intel Xeon L5520 CPU with 2.26 GHz. We implemented our memory layouts (*clustering by row*, *clustering by version*, *hybrid*) as an experimental database system whose design closely resembles the architecture of SAP HANA [24]. This prototype is a main memory column store which is used at SAP for the development of new data structures and query processing algorithms. The *clustering by version* memory layout with *synchronous columns* corresponds to the physical storage of the current release version of SAP HANA. Yet, we do not consider compression in our experiments.

The results were compared to a prototype of an in-memory row-oriented database system as a baseline. In this row store, versioning support is implemented by chaining previous versions similar to [56]. The row store consists of an array of tuples each containing the values of different attributes. For each of these tuples a reference to a chain of previous versions is stored. To insert a new row, a new tuple is appended to the array. Correspondingly, the update operation adds a new tuple to the chain of versions of a row.

5.7.2 Benchmark

For the experiments in this chapter we used a simplified version of the benchmark described in Chapter 4 by adopting the general access patterns of TPC-BiH. Our benchmark is based on TPC-H with additional update scenarios to generate realistic temporal data. For our measurements we chose the *lineitem* table from the TPC-H benchmark because this table is updated frequently; it was populated by an initial load of 10 million rows followed by 200 million updates of rows, which were chosen based on a Zipf distribution with skew parameter $s = 1.5$. For each update, the updated attribute was chosen randomly, again, according to a Zipf distribution. In approximately 50% of the updates the value of the *l_quantity* attribute was updated.

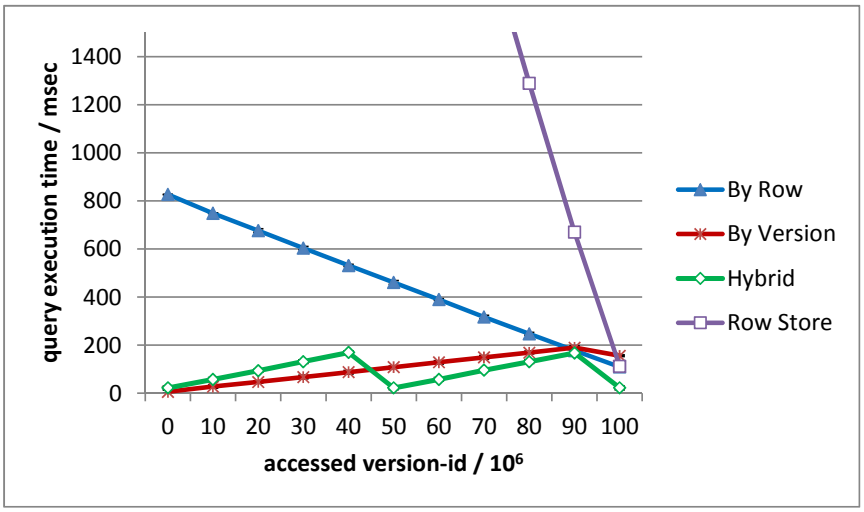


Figure 5.5: Time to Select a Given Version from One Column

5.7.3 Query Response Time Experiments

The queries in this subsection are evaluated on one column only. For the measurements we chose the *l_quantity* from the *lineitem* table.

Timeslice to a Previous Version

The measurement results shown in Figure 5.5 refer to the timeslice use case (Section 5.2.1) in which the maximum value of the *l_quantity* attribute for all rows at a given version is calculated. In this diagram the execution time of a query which performs a timeslice to a variable previous version is measured.

For the *clustering by row* layout, the query execution time decreases for higher *version-IDs*. This is due to the fact that the newest versions are stored in the leftmost *segment*. The performance decreases linearly for older versions because an increasing number of *segments* in the *overflow array* have to be read.

In contrast to *clustering by row*, the execution time increases for later versions in the *clustering by version* approach because more tuples have to be scanned for a higher *version-ID*.

The performance of the *hybrid* approach decreases faster than the *clustering by version* layout because there is an additional overhead caused by searching for the nearest checkpoint in the index. The sawtooth shape of the line is caused by the execution time increasing linearly to the distance to the nearest checkpoint. In addition, the latest version can always be retrieved in constant time from the *current* checkpoint. For a better visualization of the effects, only one checkpoint was created for the measurements.

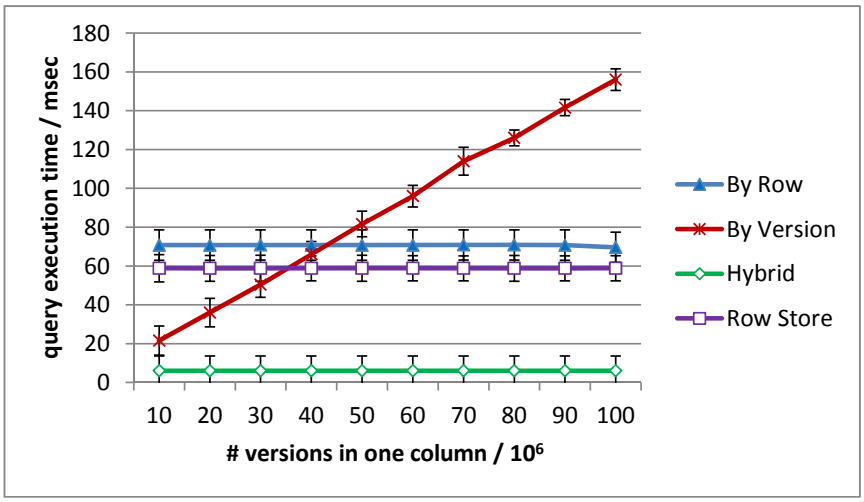


Figure 5.6: Time to Select the Latest Version

The performance of the *row store* decreases significantly for lower *version-IDs* because a pointer has to be followed for each version.

Select Value for the Latest Version

As a special case of the timeslice use case (5.2.1), Figure 5.6 shows the execution time of the query that retrieves the latest version of the *l_quantity* attribute for all rows. The query execution time is measured for a variable number of versions which are stored in this column.

For the *clustering by row* layout, the execution time is independent of the number of versions because the latest versions are always stored in the leftmost *segment* and can therefore be accessed directly. In the *clustering by version* layout, the performance decreases steadily with the number of updates because the number of tuples to scan increases. The execution time of accessing the latest version in the *hybrid* layout is constant and lower than with *clustering by row* because all data can be read from the *current* checkpoint. Within the checkpoints, the accessed values are located closer together, which results in smaller jumps in memory and better cache efficiency compared to the wider segments of the *clustering by row* layout. In this experiment, the performance of the row store is similar to *clustering by row* because both approaches store the latest version at the leftmost position. Yet, accessing the data from the row store is slightly faster because the distance of the values is smaller for this schema, thus resulting in a better cache efficiency.

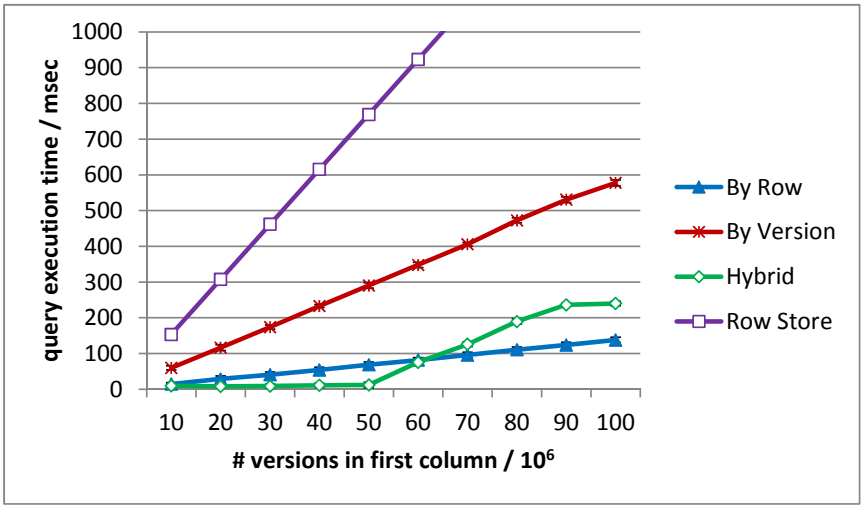


Figure 5.7: Time to Aggregate over a Time Interval

Aggregation over a Version-Interval for a Single Row

The next measurement refers to the audit use case introduced in Section 5.2.2. Figure 5.7 shows the results of calculating the maximum value of the *l_quantity* column within the version interval [60 M, 90 M] for a given row. Again, the execution time of the query is measured for a variable number of versions in this column.

The execution time of the *clustering by row* layout increases linearly with the number of versions because all versions of a row are clustered together and can be read sequentially. For the *clustering by version* approach, a full table scan is required to retrieve all versions of a row. This full traversal of all data leads to a worse performance compared to *clustering by row* because an additional comparison with the *row-ID* is required. The *hybrid* approach benefits from the checkpoint at version 50 M, which results in a linear scan within the interval [50 M, 90 M] only. The query execution time for the row store is the worst because of the pointer chasing for each version.

5.7.4 Record Reconstruction

Up to now, the performance for executing queries was shown for a single column only. In this section the record reconstruction is investigated by measuring the execution time for different numbers of selected attributes. As an example, Figure 5.8 shows the execution time of a query which selects the latest value for each row in a variable number of columns.

For the *hybrid* layouts and *clustering by row*, the latest version is directly accessible from the *current checkpoint* and the leftmost position, respectively. Therefore,

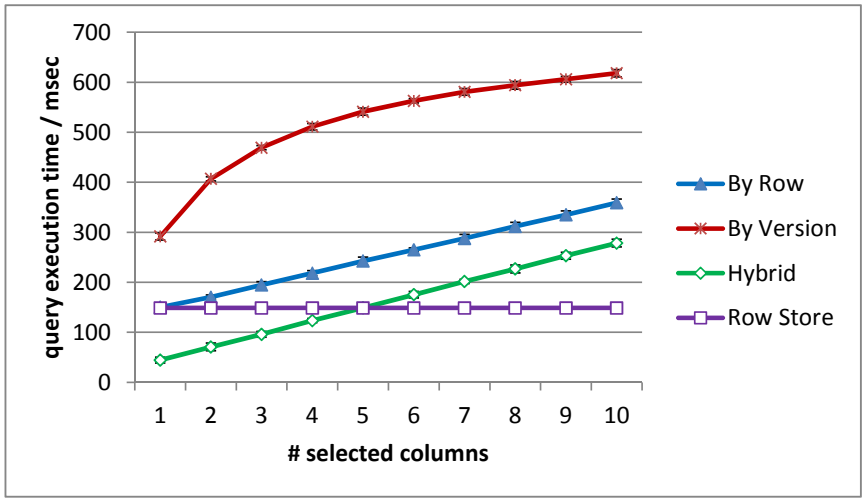


Figure 5.8: Record Reconstruction for Variable # Columns

the performance of the record reconstruction join operator is independent of the number of updates per column, which leads to a linear increase with the number of columns. In *clustering by version*, a scan has to be performed for each column. Since we are using asynchronous columns, the number of tuples in different columns are not equal. Thus, scanning columns with fewer tuples takes less time and as a result we see slower growth in the right part of the curve. Since in the row store the values for all columns are located in the same record, the execution time does not depend on the number of retrieved columns.

5.7.5 Processing Inserts and Updates

Inserts

Figure 5.9 shows the time required for inserting a variable number of values into 10 columns of the *lineitem* table. Since inserting a new row is a simple append operation to the end of array in each layout, we can see that the execution time increases linearly with the number of new rows, and the measurement results remain in the same order of magnitude for all layouts.

Updates

A value in the database can be updated by inserting a value with a new version for an existing row. Figure 5.10 visualizes the time needed to execute a variable number of updates on the *l_quantity* column for the different layouts. The update execution time is approximately in the same order of magnitude for all column store layouts because a new version has to be appended for each approach. In contrast to this,

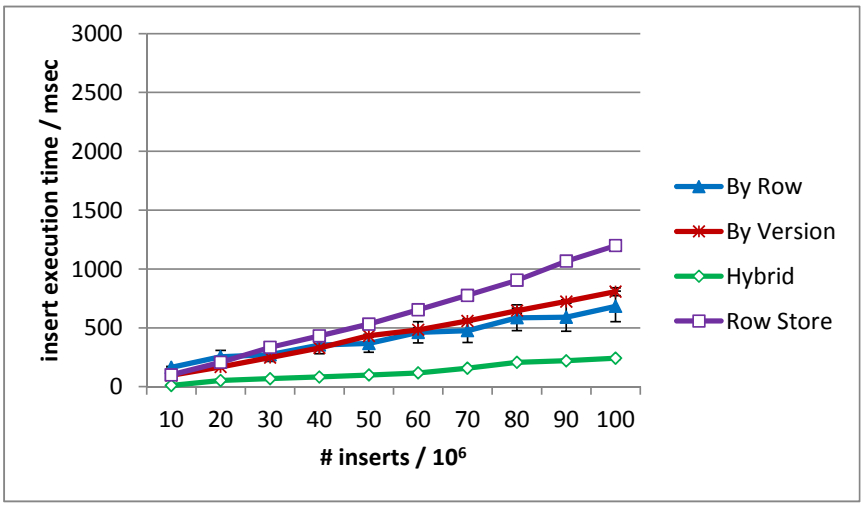


Figure 5.9: Insert Execution Time for Variable # Inserts

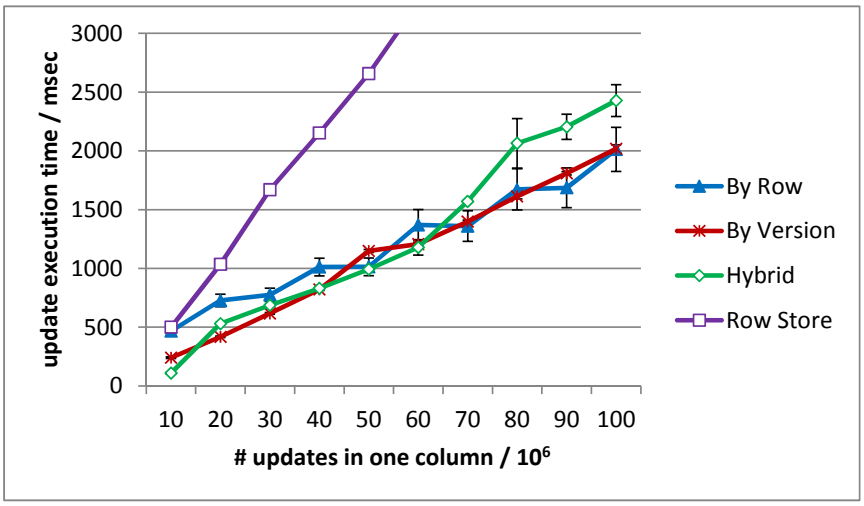


Figure 5.10: Update Execution Time for Variable # Updates

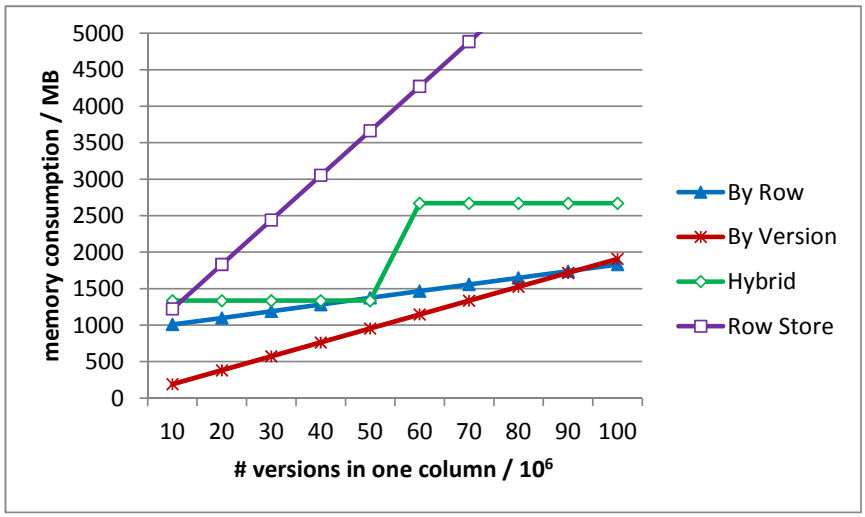


Figure 5.11: Memory for one Column with 10 Million Rows

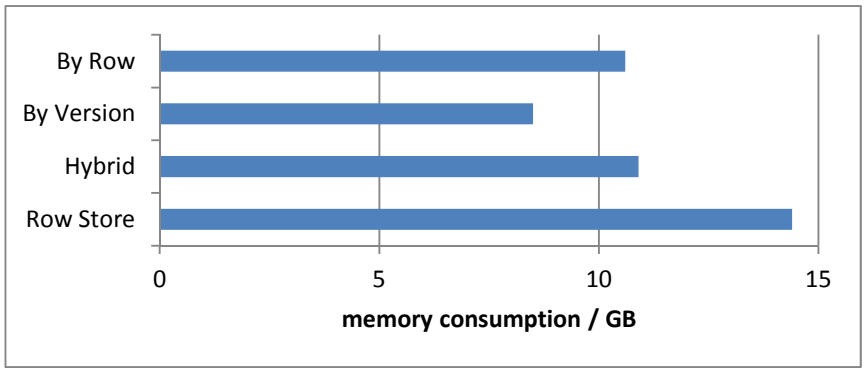


Figure 5.12: Memory of 10 Columns with 200 Million Updates

the update performance of the row store is much worse because all attributes of a row have to be replicated for each update.

5.7.6 Memory Consumption

Figure 5.11 shows how the memory consumption for the column *l_quantity* scales in all layouts with the number of updates. Compression is disabled for this measurement.

The *clustering by row* layout consumes more memory than the *clustering by version* layout because of the constant width of 10 versions per *segment*. The most memory-efficient layout is *clustering by version* since the amount of unused memory is minimal. The memory consumption of the *hybrid* approach is higher than

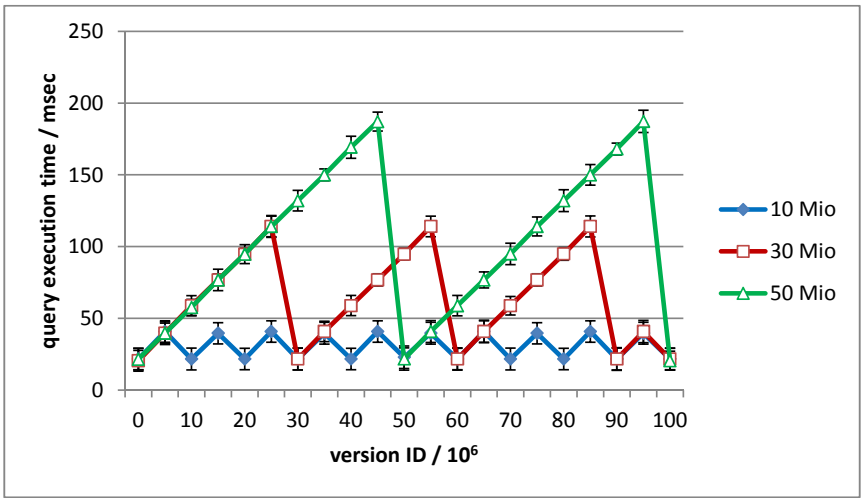


Figure 5.13: Serialization Interval for the Hybrid Approach

clustering by version due to the existence of checkpoints. The row store is the most memory-inefficient layout because of replicated data.

Figure 5.12 shows the total memory consumption of a subset of 10 columns from the *lineitem* table. The *clustering by column* approach is the most memory-efficient for 10 columns because all columns are updated independently. The memory consumption of the *hybrid* approach depends on the number of checkpoints. For our experiments, only one checkpoint was created. Again, the row store has the highest memory consumption because it has to replicate the information of all 10 columns even if only a subset of the attributes are updated.

5.7.7 Serialization Interval for the Hybrid Approach

In the *hybrid* layout described in Section 5.6, the *checkpoint interval* $delta_{max}$ defines the maximum number of updates that are stored in the *delta array* before a *checkpoint* is generated. This involves a time-space tradeoff. Figure 5.13 shows the query performance for accessing a *version-id* for three different *checkpoint intervals* in one column. A smaller *checkpoint interval* leads to a smaller amount of data to be scanned in the *delta* and a smaller maximum execution time on one hand, but causes increased memory consumption due to the larger number of checkpoints on the other hand. For our experiments with the *hybrid* approach, we chose 50 million as the *checkpoint interval* ($delta_{max} = 50\ M$), meaning that after every 50 million updates we build a new checkpoint.

5.8 Concluding Remarks

This chapter presented three alternative memory layouts to implement versioning and temporal queries in column store database systems. The first approach clusters the data by *row-ID* as a natural extension of the current design of column stores. The second layout clusters the data by *version-ID* similarly to the approach taken in log-structured file systems and PostgreSQL. Third, a hybrid approach, which is similar to the way that document versioning systems such as RCS and SVN keep versions, was presented.

Comprehensive performance experiments studied the tradeoffs of all three approaches. As a baseline for the comparisons, an implementation of versioning and temporal queries in a row store was used. The experiments showed that, overall, all three approaches to implement the temporal operators in a column store outperform the row store, with respect to both query response times and storage overhead. In terms of query response times the *hybrid* approach is the overall winner. The number of checkpoints for the *hybrid* approach involves a space-time tradeoff, and this parameter can be chosen automatically by the database system based on the expected workload.

The basic assumption for the experiments in this chapter was that it is feasible to adapt the physical location and the ordering of the data in a way which that is most appropriate for our specific temporal workload. Yet, in a commercial main memory database system such as SAP HANA the tuples are frequently repositioned in memory to achieve an optimal compressibility with the goal to both reduce memory consumption and achieve a more efficient data access. In the following Chapters 6 and 7 we will therefore not rely on the physical order of the data any more, but restore the order and the properties of the *hybrid* approach logically by means of an index data structure.

6

Timeline Index for Queries on System-Time

In this chapter we present a novel index structure called *Timeline Index* and the algorithms to process different kinds of temporal queries on this index structure. The Timeline Index we present in this chapter covers the system-time dimension only. An index data structure for bitemporal data will be introduced in Chapter 7.

In a nutshell, the Timeline Index has the following advantages:

- *Generality:* The Timeline Index is a single data structure that can be used to process a large variety of different temporal queries. In particular, we can address all our customer use cases. Besides, only a single Timeline Index per table is needed.
- *Performance:* As shown in Section 6.4, the Timeline Index outperforms the best known approaches for each kind of temporal query in our context (i.e., main memory column stores). In some cases, the Timeline Index beats the best known existing algorithms by orders of magnitude.
- *Memory efficiency:* The Timeline Index is space-efficient. There are space/-time tradeoffs, but even in a space-consuming variant, the storage overhead is usually only 15 percent of the size of the uncompressed temporal table.
- *Flexibility:* Many traditional techniques require that tables are ordered by *system-time*. This requirement limits the physical design and can result in

poor compression. The Timeline Index sheds this limitation and can be used independently of other decisions for physical database design. We therefore leverage the insights of the previous Chapter 5 in a logical way without having the reshuffle the data physically.

- *Applicability:* The Timeline Index can be integrated naturally into the SAP HANA system. It can be implemented as a normal table, thereby reusing the same structures and algorithms that are already in place to efficiently maintain and process tables.

The main ideas of the Timeline Index and the algorithms presented in this chapter are general: In principle, they can be applied to both row and column stores. Nevertheless, the focus of our work has been on main memory column stores because a core part of SAP HANA is exactly that kind of system. One may argue that temporal databases grow so large that it is not economic to keep all information in main memory. Still, utilizing compression, as well as the continuing trend of larger main memories, and distribution over clusters of machines, SAP HANA is already able to handle (temporal) queries on hundreds of terabytes of data in main memory.

The remainder of this chapter is organized as follows. Section 6.1 gives an overview of existing work on temporal data management. Section 6.2 presents the Timeline Index. Section 6.3 introduces algorithms on how to process different kinds of temporal operators using the Timeline Index. Section 6.4 gives the results of a comprehensive performance study that compares the Timeline Index with existing approaches for a variety of temporal queries. Section 6.5 contains conclusions and possible avenues for future research.

6.1 Related Work

Following Snodgrass' work on defining the foundations of the Bitemporal Conceptual Data Model in the early 1990s and the resulting TSQL2 standards proposal [75], a large body of research on temporal data has been established. Various algorithms and data structures have been proposed for different temporal operators. We will provide a brief overview, covering the parts which are relevant for our general design and our specific use cases.

6.1.1 General Temporal Access Patterns

In [20] Dignös et al. present a unified approach to map all temporal operators to the corresponding non-temporal operators which are defined by reduction rules. This mapping to non-temporal operators is achieved by 1) adjusting the time intervals of the argument tuples such that the intervals of all considered tuples are either identical or disjunct and 2) applying the non-temporal operators to the tuples with adjusted timestamps. The adjustment of the time intervals is called *temporal alignment*. This approach is generic and allows for user-defined functions and specific predicates on the resulting matching tuples by exposing the time intervals of the argument tuples as explicit attributes. In [20], the authors describe an implementation based on PostgreSQL.

An important direction of work is given by general methods to model and organize temporal data. A survey by Salzberg et al. [70] lists the typical access patterns (*Timeslice*, *Key in Time* and *Key/Time range*) and provides an overview of how well various index structures support these operations. Since most of these index structures were developed in the mid-to-late '90s, they are designed for hard-disk efficiency, optimizing the number of I/O operations for updates and queries. Tree indexes over intervals or versions are used, relying on various clustering strategies for time and key values, and partial replication for efficiency. Furthermore, some index types were designed to "truncate" history with the goal of moving it to other storage media ([59]), but distribution and parallelization have not been researched widely. Given the design goals, some proposals (such as [7]) have been proven to have (asymptotically) optimal I/O behavior for a range of temporal queries. However, given the different tradeoffs between access time, transfer speeds and CPU cost, these structures will not necessarily perform best in a main-memory setting.

Next, we will briefly discuss those indexes that are most relevant to us: The Time Index [23] (from now on referred to as *Elmasri 1990* for clarity) is one of the earliest temporal indexing methods, and provides explicit support for all our use cases. It is directly comparable to our proposed *Timeline Index* because it indexes only the time

dimension. Technically, the Time Index is a B^+-Tree over versions, in which each leaf page contains all active versions at the beginning, and the changes afterwards. The multi-version B-tree [7] (mentioned as *MVBT*) is one of the most advanced temporal indexing methods. It provides an index for both key- and time-dimensions with optimal I/O behavior. As a result, it is able to support many query classes and exploit clustering over the time and key space. Its implementation is based on a (logical) forest of B-Trees sharing pages. In contrast, [69] provides a much simpler index structure, based on a single B^+-Tree and encoding of windows over intervals, making it the closest match to the Timeline Index. The query performance of [69] is also optimal; the complexity of updates has not been studied yet.

6.1.2 Temporal Aggregation

A challenging temporal operator is temporal aggregation, in particular *temporal grouping*. In contrast to non-temporal, traditional aggregates, temporal grouping computes the aggregates as running values for time points or time intervals; e.g., the number of sales that occurred for each point in time. Temporal grouping and aggregation are well-researched topics. Kline et al. [51] introduced the first algorithm for computing temporal aggregation on constant intervals. In this algorithm, for each aggregation function and each attribute a separate data structure called *Aggregation Tree* is built. Since this tree is not guaranteed to be balanced, it may degrade into a linked list. In order to overcome the worst case, several variants of the Aggregation Tree have been proposed. However, these variants usually make special assumptions about the distribution of intervals or suffer from the drawbacks of the original design.

Böhlen et al. [14] introduced an algorithm for temporal aggregation based on AVL Trees for *begin* and *end point* values. The temporal aggregation is performed by traversing the *begin* index and inserting the tuples that are activated into the *End Point Tree*. The tuples which expire are removed from the *End Point Tree* (tuples are removed in their *end* time order), and the aggregate is returned as a result. The actual cost depends on the type aggregate: While cumulative aggregates like `SUM` and `COUNT` require little storage and cost, certain aggregates such as `MAX` drive up the resource requirements. We will discuss these issues in more detail in Section 6.3.1 because they are relevant for the design of how to process temporal aggregates with the Timeline Index, too.

Some of the general-purpose temporal index structures (e.g., [23]) are useful to compute temporal aggregates. Nevertheless, it is worth noticing that some of them, as for instance the MVBT tree, do not support temporal grouping and are limited to certain aggregates like `SUM` and `COUNT` (e.g, the MVSB tree [87] which is based on the SB-Tree [85]).

Method	Timeslice	Temporal Aggregation	Temporal Join
SB Tree	no	yes ([85])	no
Böhlen 2006	no	yes([14])	no
TSB-Tree	yes ([59])	no	no
Elmasri 1990	yes ([23])	yes ([23])	yes ([23])
MVBT	yes ([7])	(MVSB) ([87])	yes ([88])
Timeline Index	yes	yes	yes

Table 6.1: Supported Operations for different methods

6.1.3 Timeslice Operator

Establishing a consistent view of a (past) version of a database is currently the most widespread use case of temporal operations. Several database management systems provide support for this operation, which is typically called *timeslice*.

Oracle pioneered the timeslice operation with its Flashback feature [68], which is integrated into the Oracle database product. In IBM DB2 also features are available for timeslice [71] operations and the management of temporal data. PostgreSQL offers a similar functionality based on the append-only design of the PostgreSQL storage manager [77].

SAP HANA [24] (which is described in more detail in Section 3.1) provides a basic form of timeslice queries based on restoring a snapshot of a past transaction. ImmortalDB [56] by Microsoft Research is another system that supports versioning and timeslice queries by chaining versions of records and navigating to the appropriate version of a record. The indexing data structure used in ImmortalDB is a TSB-Tree [59] which defines a time range for each page in memory and keeps the data for the versions related to this time range in the corresponding page. It is therefore expected that the timeslice operator performs quite well by accessing exactly the pages that contain the data related to the target version.

Furthermore, the timeslice operator can be implemented based on general-purpose temporal index structures such as *Elmasri 1990* [23] and MVBT [7].

6.1.4 Temporal Join

Temporal joins contain predicates on both key and time domains. Typically, two tuples are considered to be join candidates on the temporal domain if their version ranges overlap.

There are two classes of algorithms for temporal joins: 1) Index-based algorithms that use extra data structures for identifying tuples or their locations, either based on their join-attribute or on their temporal properties. 2) Non-index algorithms that directly work on the temporal tables. A comprehensive survey on existing join

algorithms with support for temporal tables is provided by [27]. According to this survey, a valid-time natural join [76] can be evaluated in three different ways [27]: Using nested-loop-based, sort-merge-based and partition-based algorithms.

Elmasri 1990 [23] implements temporal joins by building a two-level index which combines a B^+-Tree index over the join attribute with a B^+-Tree index over the time dimension. The idea is that each leaf node of the top-level index (B^+-Tree) includes a value of the search attribute and a pointer to a separate index. Hence, there is an index for each attribute value. This method suffers from high memory consumption. As an alternative, [88] proposes several join algorithms which exploit MVBTs, for instance clustering in space and time, replication of records and linkage.

In summary, we can make three observations that motivated our work on a new, uniform and general-purpose data structure to support temporal queries: First, several general-purpose temporal index structures exist, but they are not tuned for large main memories and modern hardware. Second, production systems only allow for one particular kind of temporal query (i.e., timeslice), even though there is demand for all kinds of operators. Third, there is significant work on the other two types of queries (i.e., temporal joins and temporal aggregation), but all these approaches have shortcomings which limit adoption in production systems. Table 6.1 summarizes these findings on operators and specific access methods.

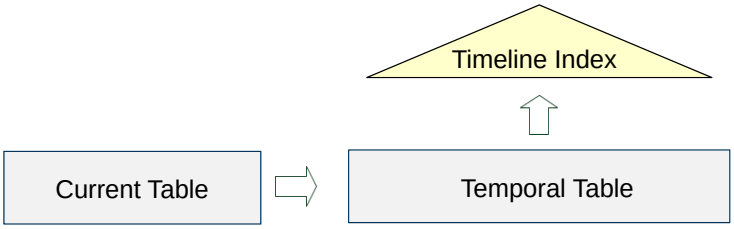


Figure 6.1: Architecture Overview

6.2 Timeline Index

This section describes the data structures and basic principles of the *Timeline Index*. Based on these data structures, Section 6.3 describes the algorithms used to implement various kinds of temporal operators.

6.2.1 Fundamentals and Overall Architecture

The lower part of Figure 6.1 shows how SAP HANA manages temporal data [24]. The same architecture has been adopted by DB2 [71]. For every table, SAP HANA keeps the *current* version of the table and the whole history of previous versions of the table in separate structures. For simplification we assume in this chapter that the *current* version is always replicated to the temporal table. The current table provides efficient access to the current state of the database as such accesses are the most common use cases for SAP HANA. Temporal features (e.g., timeslice) are implemented using the temporal table, and this is where the Timeline Index takes effect: It is an index that accelerates operations carried out on a temporal table. For each temporal table, there is exactly one Timeline Index. Temporal tables and the Timeline Index are the focus of this work.

Our work is based on the standard formalism for bitemporal data: Each tuple of the Temporal table carries two time intervals $[start_t, end_t)$ and $[start_v, end_v)$, representing *transaction time* and *valid time* (a.k.a. *system-time* and *application-time*, respectively). For the purpose of this chapter, we will focus on the *system-time* interval and call this interval $[start, end)$. The timestamps used in these intervals are discrete, monotonically increasing and scoped at the level of a database. In abstract terms, we call these values *Version_IDs*, in the concrete implementation of our system we use *Commit_IDs* of transactions as versions. Since SAP HANA uses Snapshot Isolation for concurrency control, these *Commit_IDs* provide discrete and monotonic temporal semantics.

Figure 6.2 gives an example of a Current Table (Figure 6.2a) and a temporal table (Figure 6.2b) in the SAP HANA temporal data model. This example models a small banking application with customer names and their account balance. The

Name	Balance
Carl	$100
Ellen	$700

(a) Current Table

Row-ID	Name	Balance	Begin	End
1	Alice	$200	101	103
2	Ann	$300	102	107
3	Carl	$100	103	∞
4	Alice	$500	103	106
5	Ellen	$700	105	∞
6	John	$400	105	106

(b) Temporal Table

Figure 6.2: Example Current and Temporal Tables

name of a customer is assumed to be a key. For brevity, Figure 6.2 represents the tables as they would be implemented in a row store; however, both current and temporal tables could just as well be implemented in the SAP HANA column store. Figure 6.2 shows that the new customer Alice was inserted by Transaction 101. Transaction 102 created customer Ann and Transaction 103 created Carl. In addition, Transaction 103 updated the balance of Alice, thereby invalidating the first version of the Alice record (identified by *row-ID* 1) and creating a new version of the Alice record (identified by *row-ID* 4). Transaction 104 did not update this table. Transaction 105 created two new customers (Ellen and John) and Transactions 106 and 107 deleted the accounts of Ann, Alice, and John. Figure 6.2a) shows the current state of the table after all these transactions have been applied; Figure 6.2b) captures the whole history as needed for temporal queries such as asking when Alice did have more money than Carl.

The temporal table of Figure 6.2b) is clustered by the *Begin* column. Sorting the table by *Begin* sounds like a good idea for temporal query processing and indeed many temporal index structures assume such a design. Unfortunately, this design is not good for compression: Systems like SAP HANA automatically detect the best way to sort a table, optimizing for compression ratio. It turns out that *Begin* rarely is the best clustering criterion for the temporal tables of SAP HANA. Therefore, the Timeline Index does not rely on any physical order of the temporal data in memory.

6.2.2 Timeline Index Data Structure

Figure 6.3 shows the Timeline Index for the temporal table of Figure 6.2b. The idea of the Timeline Index is to keep track of all the *visible* rows of the temporal table at every point in time. To this end, the Timeline Index returns all rows that are *activated* or *invalidated* at each point in time. For instance, Row 1 of the temporal table of Figure 6.2b) is activated at Version 101 and invalidated at Version 103 of

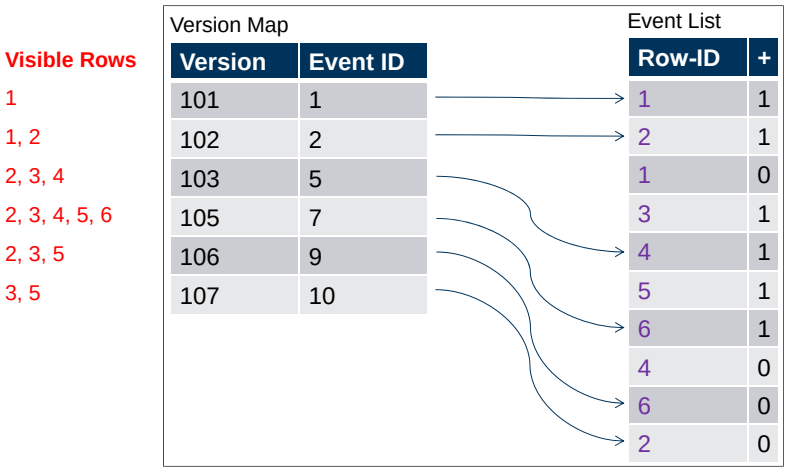


Figure 6.3: Timeline Index for temporal table of Figure 6.2b

the database. The basic idea of the Timeline Index has similarities to the LHAM approach [61]; the algorithms are based on Counting Sort [52].

More concretely, a Timeline Index consists of two data structures which are scanned concurrently to implement any kind of temporal operation (Section 6.3). The first data structure is the *Event List*. The Event List keeps track of each *invalidation* and *activation* event. Activation events are marked with a "1" and invalidation events are marked with a "0". For instance, the first entry of the Event List indicates the activation of Row 1. The second event indicates the activation of Row 2, and so on. The events in the Event List must be sorted by the (system) time when the event occurred; i.e., Row 1 was activated before Row 2. The order of events created by the same transaction is undefined; for instance, the order of the invalidation of Row 1 and activation of Row 3 is irrelevant because these events were created by Transaction 103.

The second data structure of the Timeline Index is the *Version Map*. The Version Map keeps track of the sequence of events that are *seen* by each version of the database; i.e., by each commit of a transaction. This is achieved by storing the *end* offset for each version in the Event ID column. For instance, the Version Map of Figure 6.3 indicates that Version 101 of the database sees only the first event of the Event List; Version 103 of the database sees the first five events of the Event List; its changes are contained in the range after the second event (last event of the previous version) to the fifth. By concurrently scanning and merging the Version Map and Event List, it is possible to reconstruct all the visible rows of the temporal table. All algorithms for temporal operators presented in Section 6.3 exploit this feature. Figure 6.3 shows the visible rows for each version of the database in red. Again, this information is implicit and generated while using the Timeline Index: It is not

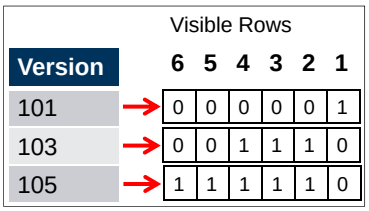


Figure 6.4: Checkpoint Index

materialized because the space overhead would be prohibitive.

Both the Version Map and Event List can be implemented efficiently using the existing structures of a column store like SAP HANA; i.e., these two structures are implemented as regular tables in SAP HANA and can be scanned and processed just like any other SAP HANA table. The only difference is that these two data structures are *append-only*; that is, once an entry has been inserted into either the Event List or Version Map, none of its fields will ever be updated. This restriction is acceptable for indexing *system-time*, but not for *application-time*. We will describe an extension of this index for *application-time* and bitemporal data in Chapter 7.

Again, it should be noted that only one Timeline Index is needed per temporal table and that the Timeline Index is significantly smaller than the Temporal Table, in particular, if the table has many columns. Our experiments (reported in Section 6.4) indicate that a Timeline Index is typically only a small fraction of the size of a temporal table, even if we include the additional space required for checkpoints, which are discussed in the next section.

6.2.3 Checkpoints

Encoding the deltas between different versions in the Timeline Index leads to a compact representation of how data evolves in time, supporting temporal aggregations well. However, reconstructing all tuples which are visible at a given version still requires the traversal of the index up to that version, leading to linearly increasing cost to access (later) versions. In addition, removing old versions for archiving or garbage collection is not possible. To overcome this problem, we augment the difference-based Timeline Index with a number of complete version representations at particular points in time. We call such a full view a *checkpoint*. As shown in Figure 6.4, a checkpoint is a bit vector which represents the visible rows of the temporal table at a certain version. In this straightforward implementation, the length of this bit vector is equal to the number of tuples stored in the temporal table at the time the checkpoint was created. We index these checkpoints by mapping the Version_ID at which the checkpoint was taken to the checkpoint contents. In addition, together with the checkpoint, we store the position of the entry in the Version Map, so that

we can start our scan there.

The cost for accessing a checkpoint is determined by the checkpoint creation policy: If checkpoints are created at fixed version intervals, the location of the latest checkpoint before a given version can be computed in $\mathcal{O}(1)$ by a simple modulo operation. In the example of Figure 6.4, checkpoints are taken regularly after 2 versions. If instead the distances are more varied (e.g., after a fixed number of operations for the specific table), we have to search, e.g., by using a binary search algorithm. As in practice a relatively small number of checkpoints is needed, even the overhead incurred by a tree search becomes almost a small constant. With this basic implementation we already reach a good tradeoff between storage space, update cost and query performance.

Further improvements are possible by using techniques such as delta checkpoints (storing the difference to a previous checkpoint, trading some computation time for space gains), bit vector compression such as run-length encoding or the Chord [19] bit vector format. A Master's thesis by Vagenas [82] covers this topic in full detail. Beside the direct benefit for query processing, checkpoints are also aiding archiving old temporal data on disk, garbage collection, parallelization and distribution to different nodes of a cluster by providing clear "cuts". Therefore, we can freely discard versions before the checkpoint, move them to a different location (disk or remote storage) or query the archived data in isolation. This enables storing all temporal data in main memory even if it exceeds the capacity of a single machine.

6.2.4 Timeline Index Construction

Based on the design of our index, we can now describe how to efficiently create and incrementally update it, even when the underlying data is not in *begin* time order. We will first show the bulk algorithm for ordered data and then generalize it. The maintenance algorithms are based on Counting Sort [52], as the index was designed to work with this approach in mind: In a first pass, we count the changed tuples per version and, based on this information, we can create compact Version Map and Event List tables and fill in the actual *row-IDs* in a second pass. The algorithm requires an intermediate table with size equal to maximum Version_ID, counting the number of events per version. First, all counters are initialized with 0. Next, at the first linear scan of the temporal table, we read the *begin* time of each tuple, take this value as position in the intermediate table and increase the counter value at this position by 1. In the same pass, we do the same for the *end* time if its value is not infinity. We can now scan the intermediate table, sum up the number of events occurring before the current version and write the value to the *Event ID* column of the Version Map, easily determining the offset of the events seen so far. Knowing the total number of versioned tuples from the last *Event ID*, we allocate space for the Event List. Now, in a second linear scan of the temporal table, we write the *row-ID*

for each *begin* and *end* at the *Event ID* given by the Version Map and increase this position by one. This results in the events for each version being sorted by *row-ID*, which minimizes random I/O. As outlined above, we add a "true" to the bit vector if the tuple is activated at this version and a "false" if it is invalidated.

The overall cost of this algorithm is linear with respect to the size of the temporal table since it needs to touch each tuple only twice – once for counting the number of events per Version_ID and once for writing the values to the Event List. The physical order of the data is irrelevant, since the Counting Sort of all Version_IDs is performed by the intermediate table.

Furthermore, the index can be updated incrementally by just appending the new versions and the corresponding events to the Timeline Index if the temporal table is sorted by *begin* time. Storing the temporal table in a different sort order, e.g., for achieving a better compression, incurs additional effort which is dominated by resorting the temporal table. In the latter case, the *row-IDs* are not stable any more and need to be updated in the Timeline Index, which can be done with cost linear with respect to the size of the temporal table. Alternatively, the index could be dropped and recomputed.

Finally, in contrast to algorithms on other temporal data structures ([3, 7, 23]), this scan-based algorithm expressing version differences lends itself well to parallelization and distribution.

6.3 Temporal Operators

The Timeline Index has been designed to provide efficient support for a wide range of temporal queries. This section covers three common types of temporal queries and shows how these queries can be implemented based on the Timeline Index: (a) temporal aggregation, (b) timeslice, and (c) temporal join.

6.3.1 Temporal Aggregation

The first temporal operator we discuss is temporal aggregation. A typical use case is a query that asks for the most expensive product at each point in time. As described in [9, 51], temporal aggregation involves the execution of an aggregate function for each *Version_ID*; i.e., each state in which the database has ever been in. Implementing temporal aggregation is demanding because it requires aggregation along the time *and* the spatial dimensions (e.g., *product*) and both of these dimensions have potentially many values. Being so challenging, temporal aggregation has already attracted considerable attention in the research literature [51, 14, 87]. The complexity of the temporal aggregation operator depends on the kind of aggregate: selective aggregates such as `MIN`, `MAX`, and `MEDIAN` are more complex than cumulative aggregates such as `SUM`. Therefore, we describe these two kinds of aggregates separately in the following subsections.

As already mentioned, temporal aggregation has not yet been standardized as part of SQL. For the purpose of this chapter, we express it with a special `GROUP BY VERSION_ID()` clause [47].

SUM, AVG, and COUNT

Example. What is the total sum of the account balances of all customers whose name start with "A" at each point in time? This query can be expressed as follows:

```
SELECT SUM(c.balance) AS sum, c.VERSION_ID()
FROM customer c
WHERE c.name LIKE 'A%'
GROUP BY c.VERSION_ID()
```

`SUM`, `AVG`, and `COUNT` are cumulative aggregates which means that a new aggregate value can be computed directly from the previous aggregate value and the changes between the next and previous version of the database. As a result, this kind of aggregation functions is the simplest case for temporal aggregation. Figure 6.5 shows how such temporal aggregates can be computed using a Timeline Index. For the sake of illustration simplicity, we use a shortened representation of the Timeline Index, which only lists the *row-IDs* for each version and indicates an

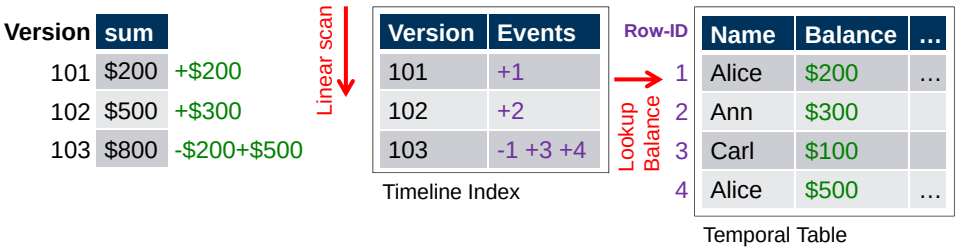


Figure 6.5: Temporal Aggregation: SUM

activation by a "+" and an invalidation by a "−". To compute a temporal `SUM`, we scan the Timeline Index to determine the new and invalidated *Customer* rows for each version. Furthermore, we keep a single variable, *sum*, that keeps track of the aggregate value during the scan for each point in time. For each entry of the Timeline Index, we check the `WHERE` clause (if the query has one) for all the new and invalidated *Customers*. If a *Customer* qualifies, we look up the *Customer*'s *balance* from the temporal table and adjust the *sum* variable accordingly (add the *balance* for a new *Customer*; subtract the *balance* for an invalidated *Customer*). This way, the *sum* variable reflects the correct aggregate value for each point time during the scan through the Timeline Index.

`COUNT` aggregates are computed analogously. For `COUNT`, we do not need to look up the *balance* values from the temporal table; only the `WHERE` clause needs to be evaluated and the running variable that keeps track of the count needs to be maintained. `AVG` is computed from `SUM` and `COUNT`. Likewise, `VARIANCE` and `STDEV` (standard deviation) can be computed from other aggregates.

Complexity. Let N be the number of rows in a temporal table. Let M be the number of events in the Event List in the Timeline Index. $N \leq M \leq 2 * N$ because each line in the table contributes at most twice to the Event List (once for activation and zero or once for invalidation). Since each event is processed exactly once and updates of the aggregation variable have a constant cost, the complexity of `SUM` and `COUNT` is $\mathcal{O}(M)$, which is in $\mathcal{O}(N)$.

MIN, MAX, MEDIAN

Example. What is the price of the most expensive unshipped item at each point in time?

```
SELECT MAX(li.l_extendedprice) AS max_price
FROM lineitem li
WHERE li.l_linestatus = 'O'
GROUP BY li.VERSION_ID()
```

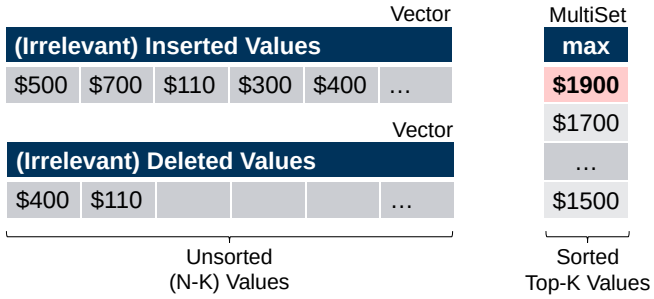


Figure 6.6: Temporal Aggregation: MAX

`MIN`, `MAX`, and `MEDIAN` are selective aggregate functions. That is, we cannot compute the new aggregate value based on the old aggregate value and information from the records that are activated and invalidated. For instance, if the current maximum is USD 1900 and the value 1900 is invalidated, we need to know about the second highest active value to retrieve the new maximum. In other words, we need to keep state of previous tuples as we go along.

To that end, we use an algorithm inspired by online Skyline computation [16] and introduce a data structure which is updated incrementally: That is, at each point in time, we keep a list of Top-K values. We keep those Top-K values sorted so that we have immediate access to these values if the maximum, top two, top three, or so values are invalidated. We keep all the other activated values (i.e., the Top K+1, K+2, ... values) in a separate, unsorted vector which we call *Inserted Values*. In addition, we store all invalidated values which are not in Top-K in the *Deleted Values* vector. These unsorted values are only needed if the Top-K values are all invalidated which happens rarely if K is chosen conservatively.

Figure 6.6 illustrates this approach. The Top-K values are represented as an ordered multiset (in order to simplify the invalidation of identical values), backed by a red-black tree. In an experiment with real-life SAP HANA data and queries, we set K to 0.01% of the number of distinct values in the data set and this way, almost all activations and invalidations were handled from the Top-K multiset.

Computing the `MEDIAN` requires special attention, but makes use of the same principles as `MIN` and `MAX`: Rather than keeping one Top-K list, two Top-K lists must be maintained to compute the `MEDIAN`. One list for the Top-K values below the median and another list for the Bottom-K values above the median.

Complexity. Determining the complexity for selective aggregates is more complicated since it involves an estimation of how the different parts of the Top-K data structure are used. Assuming N rows in the temporal table, each event is processed exactly once and inserted/removed into/from the Top-K data structure. For tuple activation, the new value is either added to the multiset or appended to the (un-

p1	**p2**	**p3**
99.440040 %	0.559953 %	0.000007 %

Table 6.2: Top-K Probabilities

sorted) vector of values that do not make it into the Top-K. For invalidation, two possible cases may occur: 1) The value is not in the multiset. In this case, the value will be simply appended to the *Deleted Values* vector. 2) The value is in Top-K. In this case, the value is removed from the Top-K multiset. If the Top-K multiset is empty as a result of this deletion, it is rebuilt with the Top-K values from the (unsorted) vector of values that initially did not make it into the Top-K multiset. The complexity of all these operations is as follows:

1. appending to one of the vectors: constant cost, i.e. $\mathcal{O}(1)$
2. inserting or removing from the multiset: cost is $\mathcal{O}(\log K)$
3. refilling the multiset: cost is $\mathcal{O}(2 \cdot (L + H \cdot \log H) + H)$ where $L \leq N$ is the number of values in the bigger vector and H is the number of elements that are retrieved from the irrelevant vectors when the multiset gets empty. H is currently chosen empirically as $2 \cdot K$. So the cost is $\mathcal{O}(N)$ assuming the cost for a partial sort is linear for the table size.

For each tuple of the temporal table, each of the three cases described above occurs with probability $p1$, $p2$ and $p3$, respectively. Table 6.2 shows the values for these probabilities that we have measured for data based on a real world scenario, generated by the TPC-H History Generator [46]. The worst-case for this algorithm in the example of a `MAX` function is a descending sequence of numbers. In this case, the cost is $\mathcal{O}(N \log H)$. It can be deduced from the resulting probabilities that the cheapest action 1) occurs in almost all the cases (99.4%). The more expensive actions 2) and 3) are rare events. For this reason, the total execution time is approximately linear with respect to the table size N in the expected case.

Custom Aggregation Functions

User-defined aggregate functions can also be supported using the Timeline Index. While none of the specific techniques we used for cumulative and selective aggregate functions are applicable without knowing the semantics of the aggregate function, the Timeline Index is nevertheless useful: In any case, it creates a window of *visible* tuples at each point in time and worst case, this window can be scanned with linear effort to construct the aggregate value. As part of future work, we intend to develop a framework that allows users to plug in efficient implementations for user-defined aggregates using a Timeline Index.

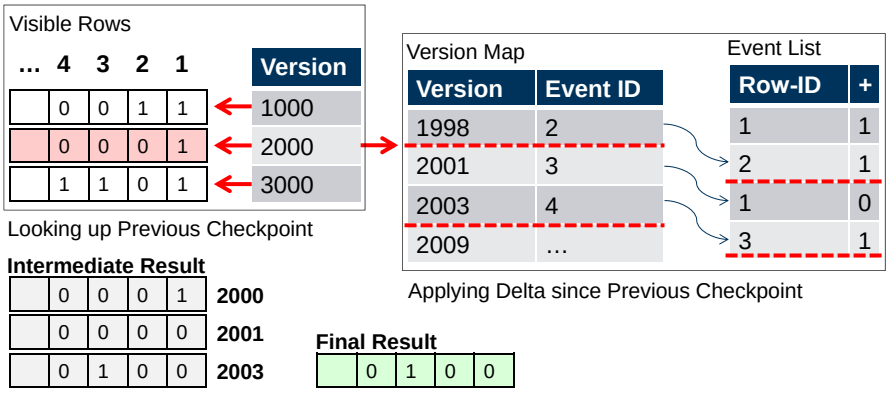


Figure 6.7: Timeslice to Version 2005

6.3.2 Timeslice

Establishing a consistent view of a previous version of the database is the most commonly used temporal operator in commercial systems. It allows the user to perform regular value queries on a single, older version of the database and corresponds closely to the *pure-timeslice* query class outlined in [70].

Example. At a given time in time, in how many cases did a product at a supplier have a stock level of less than 100 items?

```
SELECT COUNT(*)
FROM partsupp
WHERE ps_availqty < 100
AS OF TIMESTAMP '2012-01-01'
```

For timeslice, we need to establish a consistent version *VS*, i.e., provide access to exactly all those tuples that are valid for this version. As shown in Figure 6.7, we can achieve this by going back to the nearest previous checkpoint (if it exists) or otherwise the beginning of the Timeline Index. In the example of Figure 6.7, we use the checkpoint at Version 2000 in order to process the query that asks for Version 2005. The active set of that checkpoint is copied to an intermediate data structure. We then perform a linear traversal of the Timeline Index and stop when the version considered becomes greater than the version of the checkpoint, thereby covering versions 2001 and 2003 in this example. For these versions, we access the activated and invalidated *row-IDs* in the Event List and apply the changes to the intermediate data structure: 2001 invalidates *row-ID* 1, 2003 activates *row-ID* 3. Once we are finished, we can execute the query using the bit vector of the intermediate structure as a filter.

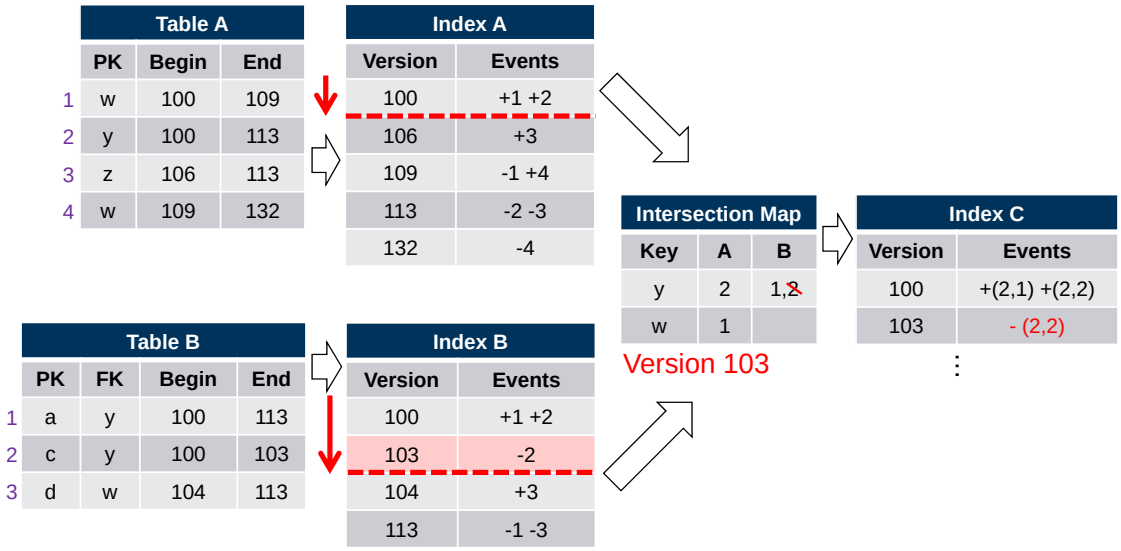


Figure 6.8: Timeline Join

Complexity. The cost depends on the rate at which we take checkpoints: The closer the better. Accessing a checkpoint can be done in constant or (small) logarithmic cost, as outlined in Section 6.2.3, whereas traversing the timeline and applying the differences is linear in the size of the Timeline Index. We will study the space/time tradeoffs of the checkpoint rate in Section 6.4.4.

6.3.3 Temporal Join (Timeline Join)

The third and most complex query class is the so-called *temporal join*. Just like temporal aggregation, this operator involves both the spatial dimension (i.e., the join predicate) and the temporal dimension (i.e., matching only tuples that were valid at the given point in time). In the interval-based temporal model, this means determining the interval intersection of versions.

Example. How many times did a customer with a balance smaller than 5000 have an open order with total price more than 10?

```
SELECT COUNT(*)
FROM customer TEMPORAL JOIN orders
WHERE o_orderstatus = 'O' AND c_acctbal < 5000
   AND o_totalprice > 10
   AND c_custkey = o_custkey
```

Our join algorithm, which we call *Timeline Join*, focuses on the temporal dimension, thereby providing most of its benefits serving temporally selective queries. It

performs an equijoin on the non-temporal (spatial) attributes, making it an instance of a temporal equijoin [27]. Its output is a slightly extended Timeline Index for the join result, where the entries in the Event List are not individual *row-IDs* for one table, but pairs of *row-IDs*, one for each partner in the respective table. This design has two benefits: 1) Additional temporal operations can easily be performed on the join results, enabling temporal n-way joins (in which the *row-ID* pairs become n-tuples). 2) Lookup of tuples in the temporal tables (e.g., for serializing the result or applying the `WHERE` condition) can be performed in a lazy manner, i.e. late materialization. *Timeline Join* is conceptually a merge-join on the already sorted Timeline Indexes, augmented by a hash-join style helper structure for the value comparisons.

Figure 6.8 shows how the Timeline Join works. The example shows the join of two tables A and B with a composite join predicate: A (spatial) value equality $A.PK = B.FK$ and time interval intersection with the condition $[A.begin, A.end)$ overlap $[B.begin, B.end)$. For both tables a Timeline Index is required. In addition, we utilize a hash-based *Intersection Map*, which relates each join key to the matching *row-IDs* in each table, formally IMap: (v) $\rightarrow (\{row - ID_A\}, \{row - ID_B\})$. To execute the join, we do a merge-join style linear scan of both Timeline Indexes (both ordered by *Version_ID*), using head pointers to the current row of each of the indexes. Starting from small *Version_IDs*, we advance the head pointer of the index with the lowest *Version_IDs*. When moving the head pointers we perform the following steps:

- If tuple a is activated in index A, we add its *row-ID* to the set for A in the intersection map, using the value of $a.PK$ as its key: IMap$(a.PK)[0] \cup (a.rowID)$.
- If tuple a is invalidated in index A, we remove its *row-ID* from the intersection map, using the value of $a.PK$ as key: IMap$(A.PK)[0] \setminus (A.rowID)$.

These steps are used for B in a similar fashion, using $b.FK$ as key and the second set. In our example, when we advance the head pointer for index B to version 103, we see the invalidation of the tuple with *row-ID* 2. Its FK value is y, so we modify the y entry in the intersection map, removing its *row-ID* 2 from the B set.

Changes to the Intersection Map will result in entries to the result table. Individual join partners are added or removed, yielding activation or deactivation pairs for this *row-ID* and its join partner. We show this case in Figure 6.8, where the removal of *row-ID* 2 for B at version 103 adds the invalidation pair (2, 2), since the B tuple *row-ID* 2, values (c, y, 100, 103), was joined with the A tuple *row-ID* 2, values (y, 100, 113) and now goes out of validity.

Complexity. In summary, the *Timeline Join* can be seen as a combination of a merge-join and a hash-join that are both adapted to consider temporal conditions

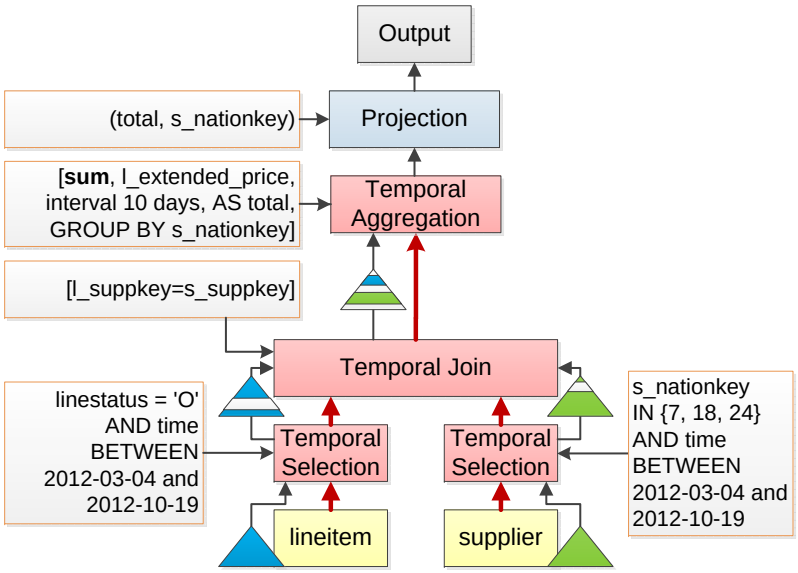


Figure 6.9: Plan of a Temporal Join and Aggregation Query

as an extra predicate in addition to the equality of the (spatial) join predicate. The cost and complexity analysis of this join algorithm shows that it requires linear time with regard to the number of versions.

6.3.4 Temporal Selection

In order to allow for generalized temporal query processing (see next section), we introduce a temporal selection operator in addition to the operators defined in [49]. Like a temporal join, it works on top of Timeline-indexed temporal tables and applies temporal as well as value-based predicates. The result is also a Timeline Index (containing the *row-IDs* of the tuples fulfilling the selection criteria), providing composability and the means for late materialization.

6.3.5 Generalizing Temporal Query Processing

Given the range of operators described in the previous section, we have all the means not only to support queries with a one-to-one correspondence to the operators, but to freely combine the operators towards generic, yet effective temporal queries. We study several classes of temporal operators with different properties: temporal selections and temporal joins maintain the temporal nature (including a Timeline Index) of the data, allowing them to serve as an input for any temporal operator. Temporal aggregations and timeslice translate temporal data into regular, non-versioned data, which can be processed by regular operators of the relational algebra. Fig-

ure 6.9 shows an example how these operators can been arranged in a complex temporal query plan, combining temporal selections, join and aggregation followed by non-temporal operators. As demonstrated in [50], all temporal operations can be implemented based on Timeline Indexes, ensuring efficient execution. Temporal operations are typically placed towards the leaves of the query plan, for accessing and processing Timeline-indexed temporal tables. Given their generic nature, these operators can participate in query optimization.

6.4 Experiments and Results

This section presents the results of experiments that assess the performance of the Timeline Index for temporal aggregation, timeslice and temporal joins. In each case, the Timeline Index is compared to the best-of-breed solutions from the literature. Furthermore, we compare our implementation of the Timeline Index with the performance of commercial database systems that support timeslice queries. Additionally, this section presents measurement for index maintenance and the storage requirements.

6.4.1 Software and Hardware Used

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 (Kernel 3.5.0-17). Our implementation of the Timeline Index was integrated into an experimental database system whose design closely resembles that of the SAP HANA [24] database product: a column store that carries out query processing entirely in memory. This prototype is used inside SAP to experiment with new query processing algorithms and data structures. It is written entirely in C++.

As mentioned in Section 6.2.3, the only tuning knob of the Timeline Index is the frequency of checkpoints. This knob trades memory consumption and update performance for query speed. We studied three versions of the Timeline Index:

1. *No Checkpoints*
2. *Few Checkpoints:* For each table, a new checkpoint was created every 22 million versions for the *Huge* dataset and every 2.2 million for *Medium*.
3. *Many Checkpoints:* For each table, a new checkpoint was created every 4.4 million versions for the *Huge* dataset and every 0.44 million for *Medium*.

Unless otherwise stated, all measurements are taken with data in random physical order. For reference, we studied the performance of two commercial database systems whenever possible. The first commercial database system was the current

Dataset	**SF_0**	**SF_H**	**|lineitem|**	**|partsupp|**	**#versions**
Tiny	0.01	0.01	0.3 Mio	0.1 Mio	0.2 Mio
Small	0.1	0.1	3.4 Mio	1.3 Mio	2.2 Mio
Medium	1.0	1.0	34 Mio	13 Mio	22 Mio
Huge	10.0	10.0	340 Mio	132 Mio	220 Mio

Table 6.3: Dataset properties

release of SAP HANA (without Timeline Indexes) and the second commercial system is referred to as *System X* because the license agreement does not allow us to reveal its true identity. Furthermore, we studied the performance of the following temporal index structures:

- *Elmasri 1990:* The Time Index as described in [23]. As no implementation was available from the authors, we implemented it ourselves. In its basic version, it only supports the time dimension. To gain better query performance, we implemented a two-level version, which uses a Time Index for every value.
- *MVBT:* The Multi-version B-tree in the Java-based XXL library[1], maintained by the authors of [7]. The MVBT provides a combined key/time index that allows for implementing a wide range of temporal queries. We tuned this implementation by using an in-memory storage container (instead of a disk-based container) and by adapting the page size for best performance. While we could measure basic index operations and timeslice, no support for temporal aggregates and temporal joins is available in XXL. Unfortunately, we could not get implementations for these operations from the authors of [88] and [87]. Therefore, we used our own implementation of MVBT-based temporal joins and did not include experiments for temporal aggregation.

As additional baselines for the experiments with temporal aggregation, we used implementations of the following algorithms:

- *Snodgrass 1995:* We used our own implementation of [51], as no other implementation was available.
- *Böhlen 2006:* We used the authors' implementation of [14].

[1] http://xxl.googlecode.com/

Table	TPC-H dbgen	Inserts	Updates	Deletes
Customer	0.2 Mio	0.2 Mio	0.6 Mio	0.0
Lineitem	6.0 Mio	1.6 Mio	1.2 Mio	0.2 Mio
Nation	25.0	0.0	0.0	0.0
Orders	1.5 Mio	0.4 Mio	0.3 Mio	0.1 Mio
Region	5.0	0.0	0.0	0.0
Partsupp	0.8 Mio	0.0	1.2 Mio	0.0
Part	0.2 Mio	0.0	0.0	0.0
Supplier	10000.0	0.0	0.0	0.0

Table 6.4: Operations per Table ($SF_0 = 1.0$ and $SF_H = 1.0$)

Timeslice is supported natively by the following commercial database systems:

- *SAP HANA:* As a baseline, we used the release version of our database system without the implementation of Timeline Index.
- *System X:* We compared our results to a (traditional) general-purpose database system which is row-based.

For temporal joins, we compared the Timeline Index to our implementation of a traditional *hash join*.

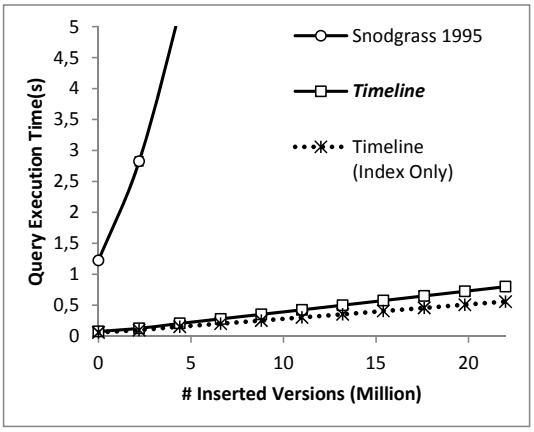
6.4.2 Benchmark

As there is no standard benchmark for temporal databases, we used our own TPC-BiH benchmark, which has been introduced in Chapter 4. For the measurements of this chapter, we used a modified version of the data generator which includes the *system-time* dimension only. According to the definition of the benchmark data set described in Section 4.3.2, our benchmark databases are characterized by two scaling factors:

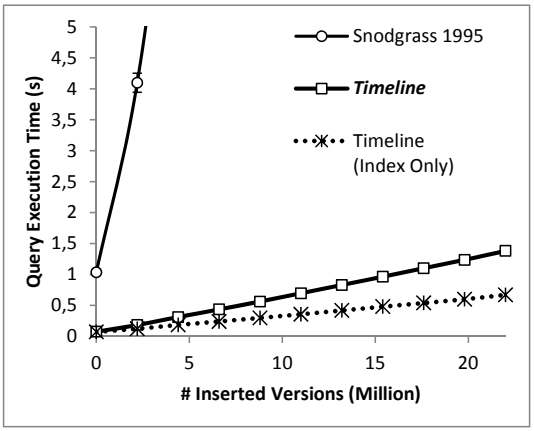
- SF_0: Scaling factor of the *dbgen* tool creating version 0.
- SF_H: Number of update transactions applied in Step 2. An example transaction is a new customer who registers his address and places an order. As a result, SF_H determines the number of versions in the benchmark database.

Given the widely varying cost of temporal operators and specific implementations, we studied four different database scaling factors ($SF_0 = SF_H$): *Tiny: 0.01*, *Small: 0.1*, *Medium: 1.0* and *Huge: 10.0*. These databases are characterized in Table 6.3. All four databases fit into the main memory of our server. This fact was exploited in the implementation of all approaches (Timeline Index, commercial products, etc.) so that no I/O was carried out as part of any of the experiments reported in this chapter. For a better understanding of the effects of creating versions with TPC-C transactions, Table 6.4 shows how many inserts, updates and deletes are carried out for each table of a *Medium* TPC-H database with $SF_0 = SF_H = 1.0$ (i.e., 2.2 million TPC-C transactions). Note that this distribution of updates keeps the properties (such as correlations and dependencies) of dataset equal to that of a normal TPC-H dataset.

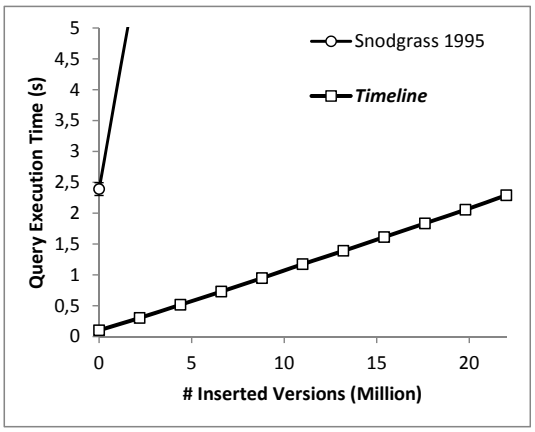
As benchmark queries, we adapted SAP customer use cases and applied them to the TPC-BiH schema (Section 4.3.1). We will describe the specific queries used in our experiments together with the experimental results in the following subsections.



(a) SUM, Sorted by Start Time



(b) SUM, Optimal Order



(c) MAX, Optimal Order

Figure 6.10: Temporal Aggregation [Medium Dataset]

# Inserted Versions (Mio)	Base Data	0.04	0.09	0.13	0.18	0.22
Elmasri 1990	48.44	108.44	266.17	543.79	779.51	1064.76
Böhlen 2006	1.57	2.62	4.04	5.44	7.43	10.90
Snodgrass 1995	0.02	0.03	0.06	0.12	0.16	0.20
Timeline Index	*0.0006*	*0.0013*	*0.0020*	*0.0027*	*0.0039*	*0.0051*

Table 6.5: Temporal Aggregation: SUM [Tiny Dataset] (sec)

# Inserted Versions (Mio)	Base Data	0.04	0.09	0.13	0.18	0.22
Elmasri 1990	55.39	119.05	295.26	600.42	881.98	1512.21
Böhlen 2006	8.53	19.61	54.46	102.60	149.26	198.73
Snodgrass 1995	0.01	0.03	0.06	0.11	0.16	0.21
Timeline Index	*0.0011*	*0.0025*	*0.0040*	*0.0056*	*0.0074*	*0.0095*

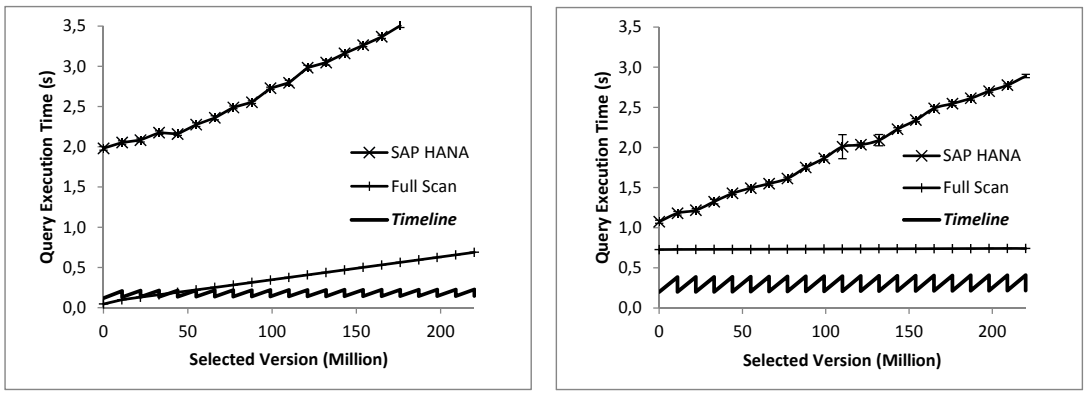
Table 6.6: Temporal Aggregation: MAX [Tiny Dataset] (sec)

6.4.3 Experiment 1: Temporal Aggregation

The first set of experiments studied the performance of the Timeline Index for temporal aggregation. As baselines, we used the classic algorithm Snodgrass 1995 [51] and the recently devised algorithm Böhlen 2006 [14]. Furthermore, we studied the performance of Elmasri 1990 [23] as a representative for a generic temporal index structure. Since the performance of the methods varied drastically, we split our results in two: 1) Figure 6.10 shows the results for the two most competitive methods on a *Medium* dataset; i.e., *Timeline* and *Snodgrass 1995*. 2) Tables 6.5 and 6.6 show the results of all methods on a *Tiny* dataset; all other approaches (except *Timeline* and *Snodgrass 1995*) timed out for any database bigger than *Tiny*.

Figure 6.10(a) and 6.10(b) shows the running time to compute a temporal aggregations according to the example of Section 6.3.1 with a `SUM`, using the Timeline Index and Snodgrass 1995. Timeline Index clearly outperforms Snodgrass 1995. The gap becomes larger when the duration of the temporal aggregation gets longer (i.e., the more tuples need to be aggregated). Comparing Figures 6.10(a) and 6.10(b), the difference becomes even more pronounced when the table is not ordered by the *begin* field; ordering by some other criterion is important to achieve optimal compression. While the Timeline Index is robust and does not require temporal order, Snodgrass 1995 is particularly sensitive to the order and, thus, limits the effects of compression in a column store. In a separate experiment (not shown for brevity) we found out that ordering by *begin* never achieves the best compression factor for SAP HANA.

We also include the cost of "Index Only" operations, which gives us some additional insights: 1) The (lower) cost of Index-Only operations such as `COUNT` 2) The



(a) Sorted by Begin Time (b) Optimal Order

Figure 6.11: Timeslice Query for Variable Version [Huge Dataset]

order-independence of index operations, as the index cost is roughly the same for a) and b) 3) The moderate, but order-dependent cost of fetching from the temporal table, in contrast to the prohibitive cost of random I/O on disk.

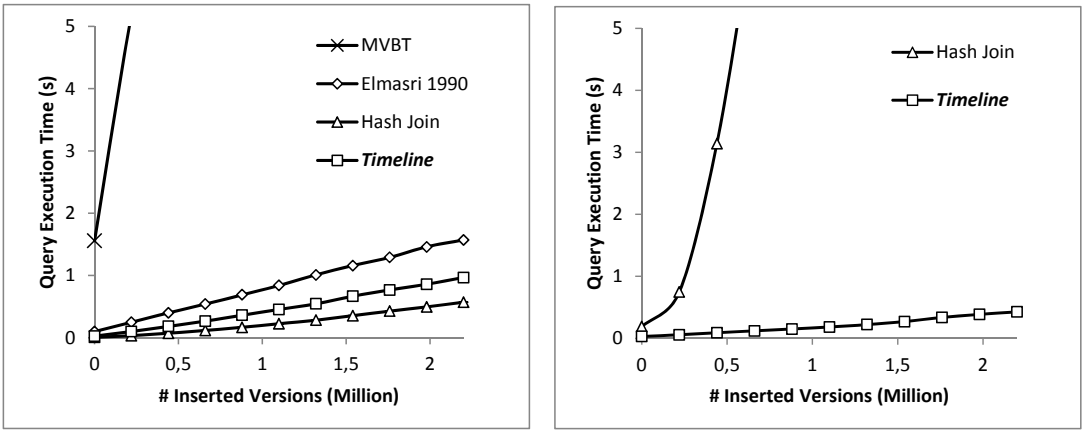
Figure 6.10(c) shows the results for the `MAX` temporal aggregation query in Section 6.3.1. Again, this query is, in theory, more complex to process with a Timeline Index whereas Snodgrass 1995 is agnostic to the aggregation function. Indeed, comparing Figures 6.10(b) and (c) the Timeline Index performs slightly worse for the *MAX* query, but the effects are small. Overall, the Timeline Index still clearly outperforms Snodgrass 1995. Even when varying the benchmark parameters and testing other queries, we could not find a single case in which Snodgrass' algorithm was better.

Since Böhlen 2006 and Elmasri 1990 did not scale well for any database bigger than *Tiny*, we present their results on the *Tiny* dataset only. Tables 6.5 and 6.6 summarize all the results. *Timeline* outperforms Böhlen 2006 by roughly two orders of magnitude and Elmasri 1990 by four.

Even though we did not experiment with this feature, *Timeline* provides an additional benefit: It is the only method that can effectively process a temporal aggregation over a limited time period; e.g., executing a temporal aggregation only for the years 2008-2010. Since the access to a specific version is fast (see next experiment), the cost for this aggregation is effectively linear to the number of versions in the query range, not in the table.

6.4.4 Experiment 2: Timeslice

Figure 6.11 and Table 6.7 show the performance of the Timeline Index for timeslice queries for a *Huge* dataset. Again, we split the presentation of the results for the



(a) Space Selective Join Query

(b) Time Selective Join Query

Figure 6.12: Join Execution Time for Increasing Table Size [Small Dataset]

Selected Version (Mio)	**0**	**44**	**88**	**132**	**176**	**220**
Elmasri 1990	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
MVBT	10.72	11.00	11.08	10.91	10.71	9.33
System X	5.32	5.61	8.21	9.86	8.86	1.14
SAP HANA	1.07	1.42	1.75	2.09	2.54	2.89
Full Scan	1.06	1.07	1.05	1.06	1.04	1.05
Timeline Index	*0.20*	*0.20*	*0.21*	*0.21*	*0.21*	*0.22*

Table 6.7: Timeslice for Variable Version [Huge Dataset] (sec)

various methods due to the huge variance in performance.

We varied the point in time that is queried: At the very left, the query is executed *AS OF* Version 0 of the database, the oldest possible version. At the very right, the query is executed against the current version of the database. We measured the Timeline Index with checkpoints created every 11 million versions and studied the two commercial database systems as well as the two general temporal indexes. As an additional baseline, we examined a table scan to process this query. Figure 6.11(a) shows the results for the case that all tables are ordered by *begin* time. The clear winner in this experiment is the Timeline Index: It performs well throughout the spectrum.

The response time of a scan-based approach grows linearly with the version number of the timeslice target if the table is ordered by *begin* time. With a growing version number, more and more of the table needs to be read and in an extreme case, the whole table (with all versions of all tuples) needs to be read in order to find the current version of all tuples.

The numbers for the release version of SAP HANA are significantly worse than the results that could be achieved with a scan because the cost of restoring an old transaction context exceeds the cost of a table scan. Figure 6.11(b) shows the results for cases in which the table was not clustered by *begin* time; instead, it was ordered to get the best possible compression. As can be seen, the Timeline Index is robust and shows (almost) the same performance, independent of the ordering of the table. The additional cost due to non-linear tuple fetching is minimal. The performance of SAP HANA improves significantly because it benefits from compression, thereby reading less data from main memory into the CPU caches. The scan-based approach performs worse: Without clustering by *begin* time, this approach needs to read the whole table in all cases.

Table 6.7 provides an overview on the remaining competitors: System X is relatively fast to access the current version, but otherwise the access time grows with the version number. It is roughly an order of magnitude slower than Timeline. MVBT as a temporal index gives approximately constant access time, but is also relatively slow. We omit Elmasri 1990, because its index could not be created within 24 hours for the *Huge* dataset.

6.4.5 Experiment 3: Temporal Join

In this set of experiments, we studied the performance of using the Timeline Index to process temporal joins. As a baseline, we used a regular hash join. The performance of a temporal join depends on two factors: (a) *spatial selectivity*, which determines how many tuples of each relation match regardless of the temporal dimension and (b) *temporal selectivity*, which determines the relation sizes for each version. Putting it differently, a temporal join is a two-dimensional join, where selectivity in both

# Inserted Versions (Mio)	Base Data	4.4	8.8	13.2	17.6	22.0
Elmasri 1990	T/O	T/O	T/O	T/O	T/O	T/O
MVBT	39.0	51.9	92.1	121.7	165.3	223.8
Böhlen 2006	13.0	25.4	42.0	64.0	87.7	114.0
Timeline - many checkpoints	0.1	1.5	3.4	5.2	7.3	9.5
Timeline - few checkpoints	0.1	1.5	3.4	5.1	7.2	9.3
Timeline - no checkpoints	0.1	1.4	3.2	4.9	6.9	9.0

Table 6.8: Index Construction Time (sec) [Medium Dataset]

dimensions matters.

To test temporal joins with varying selectivity, we studied two different join queries. First, we studied the temporal join query of Section 6.3.3. This query is highly selective in the spatial dimension. Figure 6.12(a) shows the results for this query. Then, we studied the following query which is less selective in the spatial dimension and, thus, relatively more selective in the temporal dimension:

```
SELECT COUNT(*)
FROM orders TEMPORAL JOIN Lineitem
WHERE l_returnflag = 'A'
  AND o_orderstatus = l_linestatus
  AND o_totalprice < 2500
```

Figure 6.12(b) shows the results for this query. In Figure 6.12(a), it can be seen that a traditional hash join is unbeatable if the query has a high selectivity in the spatial dimension. As Elmasri 1990 creates a tree of keys and for each key a tree of all versions, spatial selectivity can be exploited well by this data structure resulting in performance similar to hash join. Nonetheless, Timeline is also very competitive, within a small constant factor of the hash join. The performance of MVBT is somewhat unsatisfactory, as we could only rely on an index-nested loop join, and not on the fully optimized joins proposed in [88].

In turn, as shown in Figure 6.12(b), Timeline Join is the best choice if the selection along the temporal dimension matters. This result agrees with the outcomes of all other experiments: The Timeline Index is a great way to carry out any kind of selection in time. In contrast, both MVBT and Elmasri 1990 time out for this query because they rely on a space selective predicate.

6.4.6 Experiment 4: Index Construction and Maintenance

The time for constructing a new Timeline Index data structure is shown in Table 6.8. We measured the time for a complete index construction for the LINEITEM table with variable size and we compared the results to other index structures. The time for constructing an index for Elmasri 1990 was more than one hour already

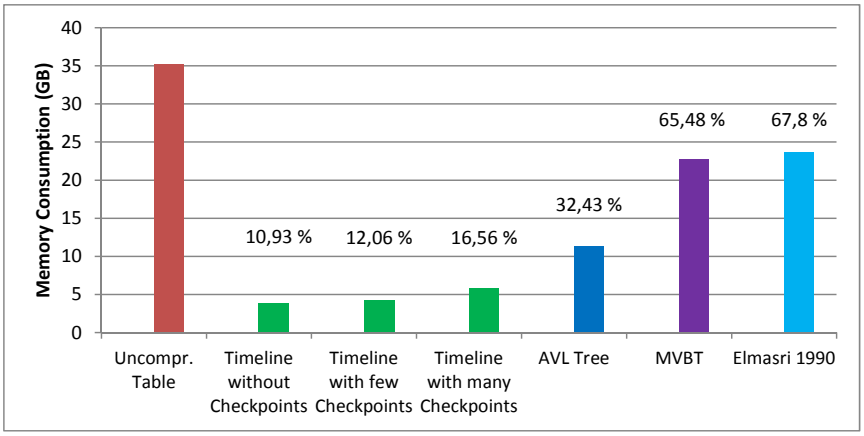


Figure 6.13: Memory for the LINEITEM Table [Huge Dataset]

with the *Small* dataset. Also for MVBT, index construction is very expensive. As shown by the measurements, the time for constructing the Timeline Index is linear with respect to the table size and much faster than an AVL tree used by Böhlen 2006. The construction of a Timeline Index is very efficient, so it is even feasible to construct the index lazily, e.g., with the first execution of a temporal operator. Checkpoints put only a minimal overhead on index construction.

Updates are supported well by Timeline Index by incrementally appending events to the index. Yet, checkpoints result in additional moderate costs. For space reasons, we omit the graph.

6.4.7 Experiment 5: Memory Consumption

The last experiment shows the memory consumption of the Timeline Index and its competitors. We measured the memory consumption for the LINEITEM table on the *Huge* dataset. As a comparison, we show the memory consumption of the uncompressed temporal table for LINEITEM, which is 35.2 GB. For this table, the size of the Timeline Index is 3.8 GB, which is approximately 10% of the table memory consumption. The size of one checkpoint is 40.5 MB, which is rather small because it is a single bit vector. Therefore the memory consumption only slightly increases for few checkpoints. For many checkpoints the memory consumption of the index data structure is still only 17% of the table which is much smaller than the memory required for MVBT and Böhlen 2006. MVBT has to deal with replicated entries, if they span active and outdated pages. The memory consumption of Elmasri 1990 is very high because of the replication of data for different versions.

6.5 Concluding Remarks

This chapter presented a novel, versatile index structure for temporal tables called *Timeline Index*, which is universal and thereby allows for an efficient implementation of a large variety of temporal operators. It is space-efficient; typically the size is only a small percentage of the size of a temporal table and a single Timeline Index per temporal table is sufficient. It is flexible and does not limit other decisions of the physical design such as compression. Furthermore, it integrates nicely into an existing database system, thereby taking advantage of highly optimized code paths to scan data, parallelize queries, and works well on modern (NUMA) hardware. Temporal operators can be nested and an efficient result construction can be achieved by late materialization. The performance is predictable, with only a single tuning knob, the number of checkpoints. Most importantly, the Timeline Index is fast: It beats all best-of-breed approaches in all our performance experiments with an in-memory column store; in some cases by orders of magnitude.

The Event Map of the Timeline Index directly returns all *change events* for each point in time, which can be exploited for numerous additional use cases, such as U2c) in Section 2.1.

The Timeline Index which has been introduced in this chapter supports the system-time dimension only. In the following Chapter 7 we will describe the *Bitemporal Timeline Index*, which is fully applicable to the bitemporal data model which has been described in Chapter 2.2.

7 Bitemporal Timeline Index

Many use cases rely on the *bitemporal* data model, i.e., a bitemporal database system stores both the state of the database at a particular point in time (called *system-time*) and the time a fact has been valid in the real world (called *application-time*). For example, it is often desirable to keep portfolio records as they actually were managed in combination with information how the portfolio was recorded at some point in time.

Since implementing these temporal features at the application level is error-prone and inefficient [71], native temporal data management support inside a DBMS has significant potential. Likewise, we are facing a lot of demand at SAP to provide such a solid temporal underpinning for the plethora of applications with temporal requirements.

Despite the tremendous amount of work on such temporal methods in the academic world, there are currently no appropriate options for real world systems on modern hardware dealing with storing and analyzing temporal data: As we will discuss in the related work of this chapter, most approaches published in research support only a single time dimension. Solutions for multiple time dimensions need to combine methods from ordered, temporal approaches and multidimensional data access.

Furthermore, given the time frame in which that research was performed (mostly 1990s), the focus was on disk-based structures optimizing for I/O behavior. The system landscape is currently changing towards main memory databases due to their significant performance benefits [67] and the affordability of large amounts of RAM.

In the previous Chapter 6 we provided dedicated, native implementations of important temporal operators such as temporal aggregation, timeslice and temporal join using a novel index data structure called *Timeline Index*. Discarding the focus on I/O optimizations permits a drastic simplification in the index design as well as better exploiting partitioning and the properties of modern hardware. While this approach shows excellent performance and only requires a small amount of memory and little maintenance cost, it is limited to system-time only.

In this chapter we propose a main memory index structure for bitemporal data which retains the useful properties of the Timeline Index, but also provides support for application-time as well as fully bitemporal queries. There are two main challenges in doing so: 1) In contrast to system-time, application-time may see updates to "past" data items, making data organization and updates more complicated. In particular, the very efficient append-only update patterns of the Timeline Index are no longer directly applicable. 2) Supporting two time dimensions introduces the challenges common to multidimensional indexing such as the need to balance the dimensions and discovering good clustering and partitioning schemes. Temporal data often contains unbounded intervals since a data item that is currently valid can stay active until a yet unknown point in the future. Therefore, classical space-minimizing partitioning strategies are less effective.

To overcome these challenges, we design the *Bitemporal Timeline Index* which utilizes single-dimension Timeline Indexes for both dimensions. The system-time index is maintained as outlined in Chapter 6, while complete application-time indexes are only kept at selected snapshots. Queries requesting values between snapshots use the nearest snapshot and the delta between those snapshots, which is readily available in the system-time Timeline Index. From an index maintenance point of view, this strategy closely resembles the delta-main merge approach common in most column stores or other read-optimized environments. Note that the Bitemporal Timeline Index is not restricted to in-memory column stores. It can equally be applied to disk-based systems and row-based database systems. However, our implementation in SAP HANA is optimized for effective usage in main memory.

In summary, this work makes the following contributions:

- a novel main memory index capable of supporting a wide range of temporal operations on bitemporal data,
- index maintenance algorithms that can trade off space consumption, update cost and query performance,
- uniform implementations for temporal operations, regardless of the time dimension, and
- a thorough performance analysis of the index and operators, showing their performance characteristics and comparing them to existing methods and systems to manage bitemporal data.

This chapter is structured as follows: Section 7.1 provides a general overview of the state-of-the-art of bitemporal data management, with a special focus on indexing approaches and systems. Section 7.2 introduces the novel aspects of the Bitemporal Timeline Index, their impact on index maintenance and the minor adaptations needed for query processing. Section 7.3 gives details on the implementation of the temporal operators. Our approach is evaluated experimentally in Section 7.4 showing the high performance and low maintenance cost of the index. Section 7.5 concludes the chapter and provides some insights into future work.

7.1 Related Work

Storage methods for temporal data have been studied for several decades now, and are described in a number of surveys in the late 1990s [55, 70]. Out of this large set, we cover methods that specifically provide indexes for a single temporal dimension as well as bitemporal indexes. In a complementary manner, we also survey techniques deployed in current commercial database systems.

7.1.1 Indexes for a single time dimension

The majority of research has been focused on indexing a single time dimension, either application or system-time. Generally speaking, there are two main approaches to organize the indexed temporal data: 1) tree structures and 2) log sequences.

Given their general availability and maturity, B-trees are a promising basis for temporal indexes, yet their limitation on totally ordered domains for keys poses an significant challenge. Therefore, a large number of approaches to organize the keys for temporal data has been proposed, some of them stemming from interval storage: Time points for the boundaries of intervals [22], composites of values and time (e.g., MAP21 [63]). The Time Index [23] is a more specialized temporal index which indexes valid time points and stores active versions at the beginning of each leaf node, followed by all changes, making it a hybrid of tree- and log-based approaches.

R-trees [32] were originally designed to index spatial data, but can naturally be used to store (time) intervals or combinations of keys and time. Some R-tree variants are optimized to meet the requirements specific for temporal indexing: the Historical R-tree [79] maintains an R-tree for each timestamp to efficiently answer time point queries. The 2R-tree [55] uses two R-trees – one R-tree stores the currently visible tuples to answer queries on currently visible data and the other one handles past data.

Furthermore, multi-version techniques can be applied to trees where trees are constructed for different versions in time, for example, the multi-version B-tree (MVBT) [7] and the multi-version 3D R-tree (MV3R) [80].

Generally speaking, all tree-based approaches have been designed for disk storage, trading off I/O operations against additional storage space and computational complexity.

In addition to trees, non-tree structures may support temporal data. The Differential Files method [36] stores changes incrementally in a *log*, and thus it is an intuitive approach to index temporal data by system-time.

7.1.2 Bitemporal Indexes

Significantly less research has been done so far for indexing bitemporal data. One straightforward way to index bitemporal data is applying a spatial index structure over rectangles which are bounded by application and system-time intervals. Such spatial indexes include the TP-Index[72], the GR-tree [13] and the 4R-tree [12]. While this approach is very intuitive, it does not allow for exploiting individual temporal orders, making them most useful for selections, but not more complex temporal operations.

An approach to compensate for this issue is to decouple the application-time and system-time dimension. In theory, any two unitemporal index structures introduced in Section 7.1.1 can be combined to support bitemporal indexing. A highly refined variant is the Multiple Incremental Valid Time Tree (M-IVTT) [64]. The M-IVTT follows a pattern of two-level bitemporal indexing trees (2LBIT) [62], which use a B+-tree to index system-time at the top level, whereas each leaf contains a pointer to an application/valid time tree (VTT) for each point in system-time. In the simplest form, each VTT is turn again a B+-tree containing keys as MAP21 values, which lead to massive replication and thus space consumption. As a refinement, Incremental VTTs (IVTT) store only the changes since the last system version in non-current versions, which incurs a significant runtime overhead when running queries. Therefore, an improved approach is to combine IVTTs and multiple full VTTs at some older system-times together to leverage between space and query performance, which constitutes the M-IVTT.

This concept can further be improved by utilizing partial persistence [55], which only allows for the tuples at the latest system-time being updated, while older versions are read-only. The Bitemporal interval tree (BIT) and bitemporal R-tree (BRT) introduced in [55] are two examples of the partial-persistent methodology.

In summary, a large number of tree-based indexes have been described in the literature which were originally designed to reduce disk I/O, often with a significant computational effort. With the presence of memory resident databases, it is not clear how these general structures can efficiently support bitemporal queries.

	Name	City	Balance	BeginApp	EndApp	BeginSys	EndSys
1	John	Smallville	50	10	∞	100	102
2	John	Smallville	50	10	11	102	∞
3	John	Largevill	40	11	∞	102	105
4	John	Largevill	30	11	13	105	110
5	John	Costtown	100	13	14	105	110
6	John	Largevill	30	14	∞	105	106
7	John	Largevill	30	14	16	106	110
8	Max	Newtown	80	14	∞	109	∞
9	John	Largevill	30	11	12	110	∞
10	John	Newtown	120	12	15	110	∞
11	John	Largevill	30	15	16	110	∞
12	John	Largevill	50	16	20	111	∞

Figure 7.1: Bitemporal Table

7.2 Bitemporal Timeline Index

The one-dimensional Timeline Index presented in [49] and summarized in Chapter 6 has many advantages. First, it supports the most important temporal operators. Second, it is fast and works extremely well on modern NUMA hardware. Third, it is space efficient as only a single Timeline Index is needed for each temporal table. Finally, it integrates well into modern DBMS such as SAP HANA.

The big limitation of the Timeline Index is that it only supports *system-time*. Unfortunately, most SAP applications and SAP HANA customers require a much richer temporal data model with, in addition to system-time, one or multiple application-defined time dimensions. The main contribution of this chapter is to propose and evaluate the *Bitemporal Timeline Index*, which builds upon the key ideas of the Timeline Index but generalizes them to implement the full bitemporal data model of the SQL:2011 standard.

Figure 7.1 shows example data which includes both a system-time (referred to as *BeginSys* and *EndSys*) and an application-time dimension (*BeginApp* and *EndApp*). In this example, the application-time refers to the time when people actually lived in a city and had a certain account balance, whereas the system-time (denoted as *BeginSys, EndSys*) refers to the time when changes were recorded in the database. We will use this example to illustrate the additional complexity introduced by the bitemporal data model.

First, updates of the application-time require a new version of the database. That is, a new version in application-time implies a new version in system-time. The opposite is not necessarily true. Second, application-time updates may change

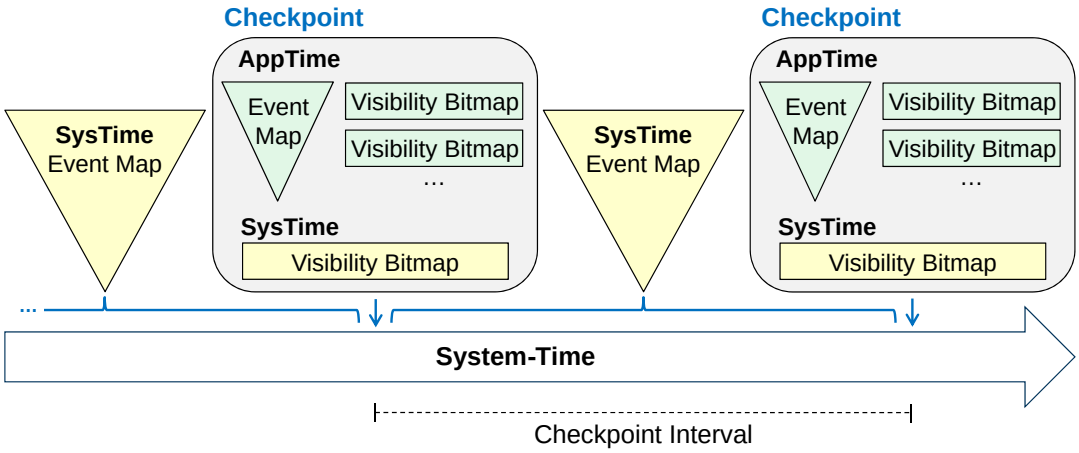


Figure 7.2: Bitemporal Timeline Index Architecture

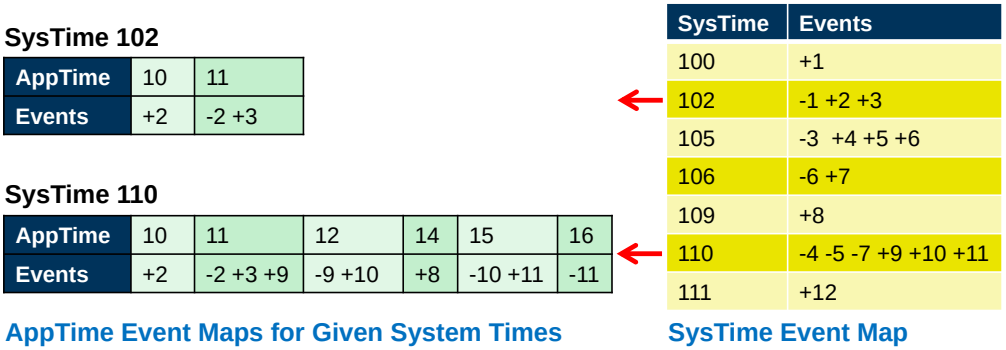


Figure 7.3: Bitemporal Timeline Index for the Table in Figure 7.1

values that were considered "past". This effect is illustrated for user John: *Row-ID* 5 stores the information that John lives in Costtown from application-times 13 to 14. However, *row-ID* 10 indicates that he lived in Newtown from 12 to 15 while later he moved to Largeville.

The Timeline Index structure as presented in Chapter 6 is insufficient to deal with these complexities because it is based on an append-only update scheme. Furthermore, the Timeline Index cannot handle queries that span multiple time dimensions.

7.2.1 Index Data Structure

The key idea of the Bitemporal Timeline Index is depicted in Figure 7.2. The Bitemporal Timeline Index extends the (regular) Timeline Index by maintaining an application-time *Event Map* for every application-time dimension in every checkpoint. This application-time Event Map can directly be used for temporal operators

(such as timeslice, join or aggregation) in application-time, if the query matches the checkpoint in system-time. Otherwise, we need to consider the system-time Event Map in order to pick up all events that may have changed the application-time after the checkpoint. In addition, the Visibility Bitmaps for application-time allow for accessing a given point in application without scanning the complete Event Map.

The Bitemporal Timeline Index for our running example is given in Figure 7.3 (for simplicity we omit application-time Visibility Bitmaps). For instance, to find out where John lived at application-time 13 according to the state of the database at system-time 105, we consult the application-time Event Map denoted "SysTime 105" in the top left corner of Figure 7.3. The application-time Event Map tells us that row 5 is visible for application-time 13. The concrete change is only stored in the table, i.e., that John moved to Costtown.

A single Bitemporal Timeline Index is sufficient for each temporal table and within each checkpoint one Event Map and a set of Visibility Bitmaps can be created for each application-time dimension. The frequency of checkpoints and the choice for which application-time dimensions to create an application-time Event Map at each checkpoint is tunable based on the database workload.

7.2.2 Index Construction and Maintenance

The construction of a Bitemporal Timeline Index is similar as for system-time only. Yet, for each application-time dimension, at each checkpoint we create an (application-time) Event Map and a set of Visibility Bitmaps, considering all tuples that are visible at the system-time of this checkpoint. This Event Map be can either be created dynamically for query processing or serialized as a new checkpoint. In this section we describe how to create such an application-time Event Map incrementally.

Again, we can take advantage of previous checkpoints in order to limit the scope of this scan through the temporal table. The process of how to construct a new (updated) application-time Event Map from a previous checkpoint incrementally is depicted in Figure 7.4, which shows the changes to the underlying data in red, either as additions (tuples 7-11) or as system-time invalidations (EndSys of tuples 4-6). The starting point is the application-time Event Map from the previous checkpoint, taken at system-time 105 in this example and denoted as (A) in Figure 7.4. Furthermore, we compute a *Delta* (denoted as (D)) against this application Event Map using the system-time Event Map (B) and the temporal table (C). This Delta contains insertions and deletions (denoted as "()") of events that occurred after the checkpoint (i.e., from system-times 106 to 110 in this example). For instance, at system-time 109, Tuple 8 is added which affects an event at application-time 14 so that the Delta records a "+8" event at application-time 14. As another example, the insertion of Tuple 9 at system-time 110 invokes two events in application-time:

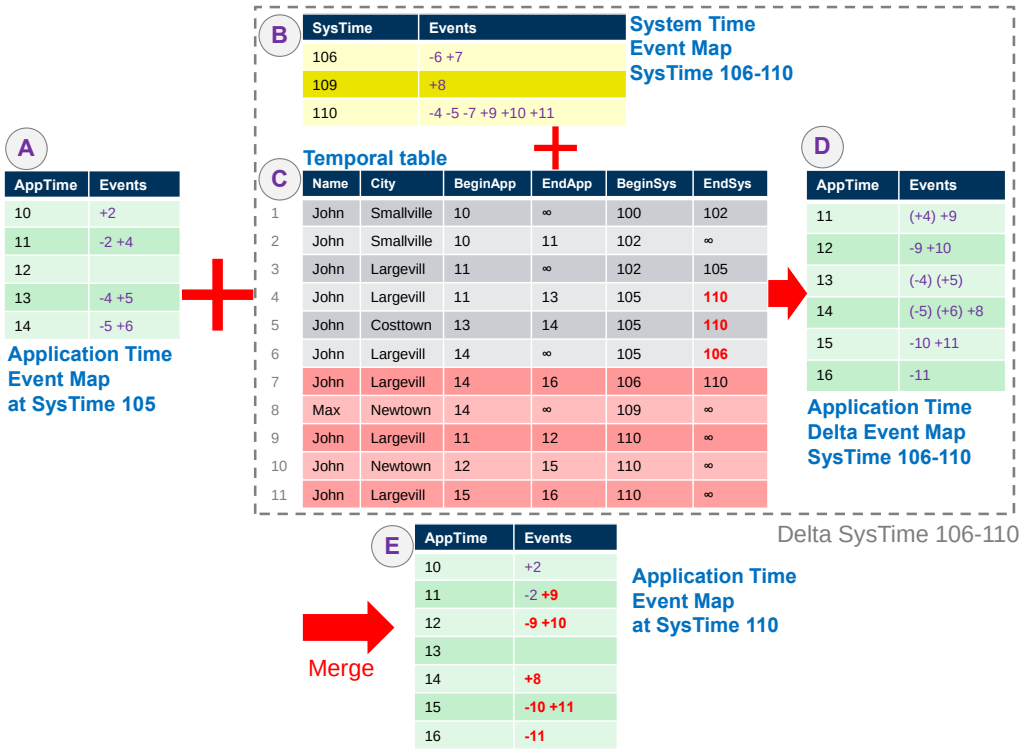


Figure 7.4: Incremental Construction of an Application-Time Event Map

First, the insertion of Tuple 9 at application-time 11 and second, the invalidation of Tuple 9 at application-time 12. As yet another example, the deletion of Tuple 4 at system-time 110, invokes two *deletion* events in the Delta: In order to express that Tuple 4 should be removed from the Event Map, we encode these *deletion* events as "(+4)" and "(-4)" in the Delta.

As a final step to construct the new application-time Event Map at system-time 110, the Event Map from system-time 105 (A) is merged with the Delta (D): insertions in D (e.g., +9 at Time 11) are added to A; invalidations in D (e.g., -9 at Time 12) are also added to A; deletions in D (e.g., (+4) and (-4) at Times 11 and 13) result in deleting these entries from A. This merge is performed in linear time because both A and D are sorted by application-time.

Once an application-time Event Map has been created it is immutable because it is valid for a fixed version in system-time. As a result, the index can be stored in a read-optimized form. Creating the delta explicitly instead of applying the changes directly allows us to decouple computation and cache these deltas for later use.

7.2.3 Bitemporal Operators

Application-time Event Maps are used in exactly the same way as regular system-time Event Map to process temporal aggregation, timeslice, and temporal joins (Section 6.3). We gave a simple example in Section 7.2.1 by showing how the Bitemporal Timeline Index of Figure 7.3 can be used to find out where John lived at application-time 13 for a database at system-time 105.

The application gets more interesting if the system-time does not match a checkpoint; e.g., finding John's address at application-time 13 for system-time 108. In this case, we apply the same technique described in Figure 7.4, carrying out the following steps:

1. Find the latest checkpoint (i.e., at system-time 105 in this example) and use the application-time Event Map from that checkpoint as a basis. If only a limited range within the application-time dimension is selected, checkpoints for the application-time Event Map can be exploited for an efficient access of the points in application-time.

2. Construct a *Delta* by scanning all the events in the system-time Event Map after that checkpoint. This step is carried out in exactly the same way as shown in Figure 7.4.

3. With the application-time Event Map and the Delta, we have all we need to carry out the temporal operation for the given system-time. Instead of materializing a new application-time Event Map, we pipeline the result of the merge into a temporal operator, thereby applying this operator.

This approach works for any temporal operator that can be implemented based on Timeline Indexes (i.e., timeslice, range queries, temporal aggregation, and temporal join). It is also applicable if the query involves several application-time dimensions. In this case, we retrieve the application-time Event Map from the latest checkpoint for each application-time dimension, construct a Delta for each application-time dimension and merge all the Deltas and application-time Event Maps in order to execute the temporal operator.

7.3 Implementation

After having discussed the basic idea of the Bitemporal Timeline Index in Section 7.2, we now elaborate on its effective usage. First, we explain generic access patterns used by temporal operators. Second, we describe these temporal operators in detail.

7.3.1 Index Access Patterns

With the Bitemporal Timeline Index we support temporal queries on both time dimensions: system-time and application-time. We express all these access patterns as range queries on a temporal range $[b, e)$, where $\perp$ denotes an unspecified point in time. A query may access each time dimension in 3 different ways:

Point in Time t: All tuples are selected which are visible at a particular point in time t.

Range $[b, e)$: A range $[b, e)$, $b < e$ means we look at a (half-open) time interval. All tuples are added to the result whose visibility interval overlaps.

Agnostic $(\perp, \perp)$: There is no restriction for this time domain, all tuples are selected.

Point in time and agnostic access are both special cases of retrieving a temporal range. The queries often have specific access patterns: Table 7.1 gives an overview how we can use the Bitemporal Timeline Index for different combinations of these access patterns. Let us consider the case where both dimensions are constrained to a point (S/A), which may be used for timeslice in both dimensions: We start from latest previous (system-time) checkpoint, which gives us access to 1) the set of all tuples that are active in that system-time and 2) an application-time Event Map and Visibility Bitmaps at this point. We search for the nearest previous (application-time) Visibility Bitmap, which provides us with the information on the tuples that are active in application-time. We then traverse the application-time Event Map to retrieve the event in the application-time domain until we reach the desired point in application-time. If the requested system-time corresponds to the system-time of the checkpoint, we are done. If not, we need to apply the deltas from the system-time Event Map until the requested point in system-time, while filtering them against their application-time. This way, we save the cost of building and merging an index.

Most operations can directly be executed in this way, as indicated by the other cases shown in Table 7.1. Some operations, however, such as temporal aggregation or temporal join over application-time only, require the presence of an application-time Event Map at a specific system-time. As we are not able to store a complete

application-time Event Map for each point in system-time, we need to reconstruct the information from the checkpoints and the system-time delta at runtime. By changing the checkpoint interval we can trade faster execution time for increased space consumption. We consider three different alternative approaches to retrieve the application-time state for a given system-time:

Recompute (R). Rebuild the application-time Event Map completely from scratch (without making use of checkpoints).

Index Delta Merge (M). Retrieve the application-time Event Map from the latest checkpoint, compute the application delta index, merge both into a new Event Map.

Dual Index (D). Retrieve the application-time Event Map from the latest checkpoint, compute the application delta index and give both as an input to the temporal operators.

Recompute **(R)** is the slowest approach with a constant overhead, whereas the other two alternatives Delta Merge **(M)** and Dual Index **(D)** have similar performance. **(M)** has the advantage that we can re-use same implementation of our temporal operators for both time dimensions, whereas **(D)** requires an adapted implementation for each operator. We therefore use **(M)** for the experiments. For future work, we want to investigate if there are benefits from caching deltas as well as considering the next "future" checkpoint.

7.3.2 Bitemporal Operators

Temporal Aggregation. For a temporal aggregation over system-time with the pattern $([S_b, S_e)/(\bot, \bot))$ and indicated by `GROUP BY SYSTEM_TIME()` in our extended SQL syntax, we can immediately use the implementation of [49], relying on the system-time Timeline Index. If we perform a temporal aggregation over application-time for a fixed point S in system-time $(S/[A_b, A_e))$, using `GROUP BY APPLICATION_TIME()` in our syntax, we can reconstruct the corresponding application-time Event Map, as outlined above.

Timeslice. In this chapter we extend the scope of [49] and consider temporal conditions on both system- and application-time. Pure system-time $(S/(\bot, \bot))$ can be computed efficiently as discussed in [49] using only the system-time Event Map and Visibility Bitmaps. A pure application-timeslice $((\bot, \bot)/A)$ becomes a table scan, since there is no selection on the system-time, while all application-time Event Maps are only valid for a specific point in system-time. Constraining both dimensions (S/A) leads to the algorithm outlined in the previous section, avoiding the creation of an application-time Event Map.

Temporal Join. For the bitemporal join we have to distinguish three cases for the join predicates: 1) If the temporal join predicate is on system-time, we rely on the system-time Event Map and Visibility Bitmaps in available checkpoints to evaluate the join as described in [49]. Any non-temporal predicates can be checked after matching tuples were found. 2) If the temporal predicates use both system-time and application-time, we use the same algorithm as in 1) and apply the join predicate for application-time after evaluating the predicate on system-time. 3) Finally, if the temporal predicate is on application-time (assuming temporal restrictions on system-time for the inputs), we can construct the application-time Event Map for the requested system-times using the approach described in Section 7.3.1. This allows us to use the same algorithm as for 1) also for joining on application-time.

Range Queries. A range query generalizes the definition of timeslice to visibility intervals for one or many time dimensions. All tuples are included in the result for which the visibility interval overlaps. As outlined in Table 7.1, there is a wide range of options depending on ranges on each dimension. Whenever system-time is involved, we get all visible tuples which are valid at the lower bound of the system-time interval as described for the timeslice operator above. We then resume scanning the system-time Event Map and apply the delta. Any other predicates including conditions on the application-time can be applied by accessing the temporal table for all matching tuples.

Thus, the Bitemporal Timeline Index supports both time dimensions effectively whenever the system-time dimension is restricted. In case the index selectivity is too high, we always have the option to fall back to a scan of the temporal table, as this is very competitive in main-memory DBMSs.

SysTime	**AppTime**	**Index Usage**
S	A	• Search latest previous checkpoint based on SysTime S • Search latest previous AppTime Visibility Bitmap for A • Follow AppTime Event Map until A is reached (toggle bits) • Follow SysTime Event Map until S is reached (toggle bits, apply events only for tuples visible for A)
S	$[A_b, A_e)$	• Like S/A, but continue following AppTime Event Map until A_e (set bits to true for all activated tuples in $[A_b, A_e)$ to implement a union operation) • Follow SysTime Event Map until S is reached (toggle bits, apply events only for tuples visible for $[A_b, A_e)$)
S	$(\bot, \bot)$	• Like S/A, but only use system-timeline Index
$[S_b, S_e)$	A	• Like S/A, but continue following Event Map until S_e • Set bits for all activated tuples in $[S_b, S_e)$
$[S_b, S_e)$	$[A_b, A_e)$	• Like $S/[A_b, A_e)$, but continue following Event Map until S_e is reached (set bits, apply events only for tuples visible for $[A_b, A_e)$)
$[S_b, S_e)$	$(\bot, \bot)$	• Ignore the application-time
$(\bot, \bot)$	*	• Do a table scan instead because the Timeline Index would be inefficient

Table 7.1: Index Usage for Different Access Patterns

7.4 Experiments and Results

In this section we evaluate the performance of a comprehensive temporal workload and compare the Bitemporal Timeline Index to several state-of-the-art index types as well as a commercial DBMS.

7.4.1 Software and Hardware Used

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. Our implementation of the Bitemporal Timeline Index was integrated into a database prototype whose design closely resembles that of the SAP HANA database product: a column store that carries out query processing entirely in memory. This C++-prototype is used inside SAP to develop and evaluate new query processing algorithms and data structures. For all measurements we set a timeout of 60 minutes and repeated them 10 times after a warmup.

7.4.2 Benchmark

Benchmark Definition. In order to provide a good coverage of temporal workloads, we chose the data sets and selected queries from our TPC-BiH benchmark proposal (Section 4.3), which we already used to evaluate the performance of several commercial bitemporal DBMS (Section 4.5). The benchmark provides a bitemporal schema, a data evolution workload produced by a generator and a set of queries stressing a comprehensive set of operators and access patterns.

Data Sets. The data generator from the TPC-BiH benchmark takes the output of the standard TPC-H generator as version 1 and adds a history to it by executing update scenarios (i.e., new order, deliver order, cancel order). These scenarios were designed to match real use-cases from SAP and its customers, providing a realistic workload which corresponds the properties of a real-life temporal database. Each update scenario results in one transaction which generates a new version in our temporal database. As shown in Table 7.2, the size of the data set is determined by two scaling factors:

- SF_0: The scaling factor of the TPC-H generator.
- SF_H: The scaling factor determining the size of the history as number of update transactions (in Millions).

The schema of the TPC-BiH data set is based on the standard TPC-H schema, but includes additional attributes reflecting the time dimensions for system-time

and application-time for each table. The granularity of these time dimensions can be configured per table individually and has been chosen as minutes for application-time and nanoseconds for system-time in the measurements of this section.

We used three data sets with different sizes for our experiments as described in Table 7.2:

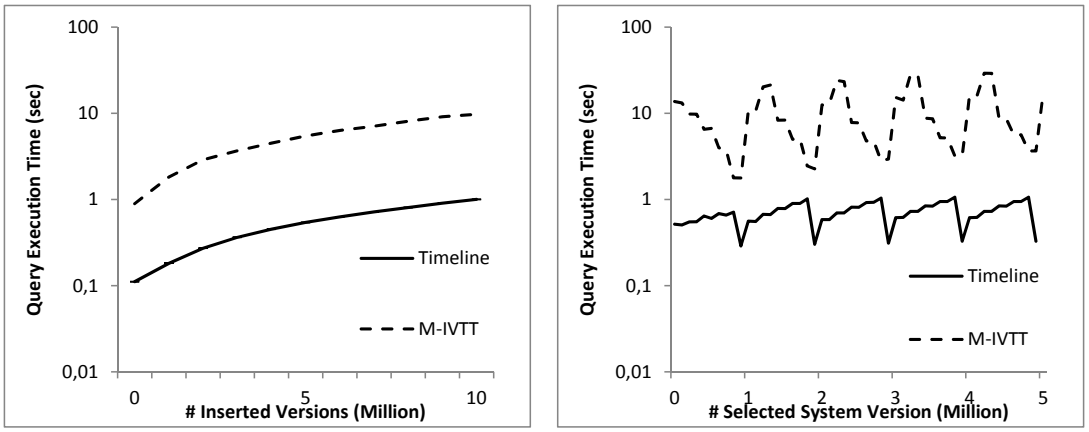
- **Tiny Data Set**, for expensive/unoptimized operators.
- **Medium Data Set**, default workload with a short history.
- **Large Data Set**, extending the history of the Medium Data.

Contenders. For our measurements we compared 4 competitors:

- Our Bitemporal Timeline prototype (referred to as "**Timeline**") uses the data structures and algorithms introduced in this chapter. This stand-alone prototype is written in C++ and reflects the architecture of SAP HANA. Unless stated otherwise, we use 10 system checkpoints and thus 10 application-time Event Maps with 10 Visibility Bitmaps each.
- The **M-IVTT** [64] uses a two-level bitemporal indexing tree to index bitemporal data. As no source code was available from the authors, we developed our own implementation of M-IVTT. This is the best B-tree-based implementation for implementing bitemporal operators we are aware of. Similar to Timeline, we use 10 full VTTs.
- The **RR*-tree** [8] is an optimized R*-tree which reduces the imbalance caused by updates. For our experiments we used an RR*-tree implementation from the authors. It is the fastest R-tree-based version we know about.
- **System A** is a commercial disk-based relational database with native support for bitemporal features. Due to license regulations we are not allowed to reveal the actual name. We created indexes for this system which have been recommended by the index advisor for each workload. Given the workload parameters (RAM, data set size), we could ensure that the entire workload is served from RAM after warmup.

Data Set	**SF_0**	**SF_H**	**|lineitem|**	**|partsupp|**	**#versions**
Tiny	0.01	0.1	0.3 Mio	0.08 Mio	0.1 Mio
Medium	1	10	28 Mio	8 Mio	10 Mio
Large	1	25	68 Mio	19 Mio	25 Mio

Table 7.2: Data Set Properties



(a) System-Time (b) Application-Time

Figure 7.5: Temporal Aggregation [Medium Data Set]

7.4.3 Experiment 1: Temporal Aggregation

In the first set of queries we evaluate the performance of temporal aggregation. This operator stresses the temporal order aspect significantly, as it traces the evolution of the data in the temporal dimension. As such, it is also a good representative for many temporal analyses such as window queries or time series.

We utilized the TPC-BiH Query R.3b and varied the time dimension. We do not show any results for System A, as the measurements already timed out for the *Tiny* workload. Likewise, no implementation of temporal aggregation is available for RR* at the moment, as it does not deliver the results in any temporal order.

A1: Temporal Aggregation over System-Time. We start our analysis with a temporal aggregation over system-time for a fixed application-time, using a selective aggregation function.

```
SELECT MAX(l_extendedprice)
FROM lineitem l
  FOR APPLICATION_TIME AS OF TIMESTAMP '[APP_TIME]'
WHERE l_linestatus = 'O'
GROUP BY l.SYSTEM_TIME()
```

This query is evaluated on the *Medium* data set for an increasing history size, increasing the history in steps of 10% from 0 to the full Medium data set. As it is shown by Figure 7.5(a), the temporal aggregation algorithm based on the *Timeline Index* scales linearly with the size of the data set. The query execution time is about 1 second for a data set of 10 million versions. For a temporal aggregation over

system-time, the system-time Timeline Index is directly applicable. As described in Section 6.3, it can be scanned linearly for activations and invalidations of tuples, which can be exploited well for the computation of the aggregation. The results are in line with performance seen in [49].

On the other hand, *M-IVTT* also seems to scale linearly with the data set, but with a about an order of magnitude times slower execution time. The reason for the worse performance of M-IVTT are: 1) A large amount of time is spent on constructing a full snapshot of the valid time tree (VTT). 2) Scanning the VTT is less efficient than a scan of the Timeline Index because it results random access patterns by following pointers in the tree structure. 3) M-IVTT encodes the time interval as a single value, and thus, the encoding and decoding of an interval takes extra effort.

A2: Temporal Aggregation over Application-Time. In this query, the aggregation is performed on the application-time domain:

```
SELECT MAX(l_extendedprice)
FROM lineitem l
  FOR SYSTEM_TIME AS OF TIMESTAMP '[SYS_TIME]'
WHERE l_linestatus = 'O'
GROUP BY l.APPLICATION_TIME()
```

We keep the size of the data set constant (full Medium) and vary the point in system-time instead. For better visibility we show the version range from 0 to 5 million (out of a 10 million).

As Figure 7.5(b) shows, *Timeline* exhibits a sawtooth pattern with a generally flat trend. These variations in runtime can be explained by the limited number of application-time Event Maps which are kept at the 10 checkpoints in system-time. If such a checkpoint/index is available, the aggregation is performed on the Timeline Index in the same way as in the previous experiment, leading to a dip in the graph. In turn, when no index is available we need to reconstruct the fitting Event Map from the existing index, as outlined in Section 7.3.2. We show details of the tradeoffs among different reconstruction methods in Section 7.4.7; as we use the Delta Merge **(M)** approach, the cost increases the further we get from the closest *previous* checkpoint.

M-IVTT is also able to perform a backwards scan from the immediately *following* checkpoint, therefore producing a more symmetric pattern. Again, the performance of M-IVTT is about an order of magnitude worse. This is due to the fact that in the case of application-time the whole valid time tree has to be traversed each time as no patches are available for this time dimension.

7.4.4 Experiment 2: Timeslice

The next (and most popular) class of queries covers the timeslice operator, which restores a certain state in time stressing the selection capabilities of the index. In bitemporal settings, a timeslice can be applied to either dimension individually or on both. To this end, we examine three variants of the following query where we use different conditions for the timeslice operator. For the queries in this section we adopt TPC-BiH Query T.1 and vary the point in each time dimension, using `TEMPORAL_CONDITION` as a placeholder:

```
SELECT AVG(ps_supplycost)
FROM partsupp
TEMPORAL_CONDITION
```

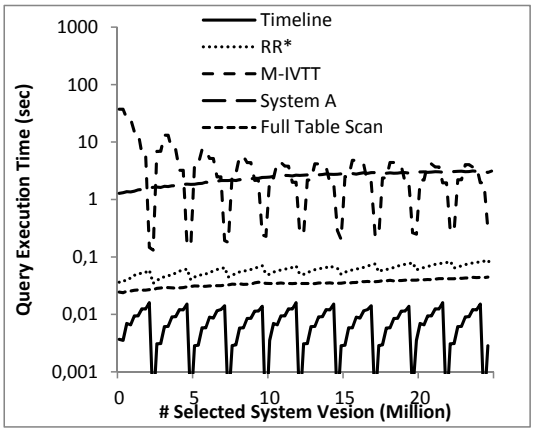
The queries are measured on the *Large* data set, varying the selected version. Figure 7.6 summarizes our results for the different time dimensions.

T1: System-Timeslice Only. The first query (Figure 7.6(a)) performs a timeslice to a given point in the system-time dimension while considering the entire application-time. Hence we replace the placeholder `TEMPORAL_CONDITION` with `FOR SYSTEM_TIME AS OF TIMESTAMP '[SYS_TIME]'`.

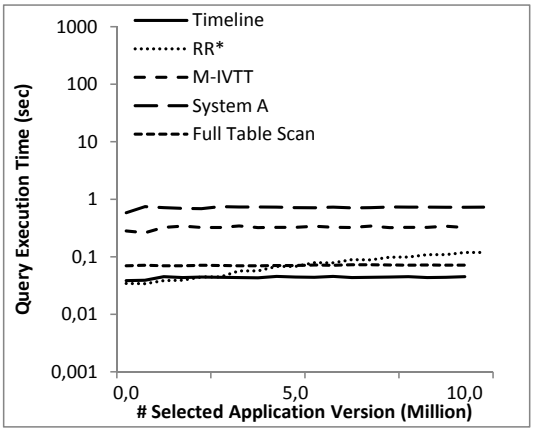
For *Timeline* we see once more a sawtooth pattern as the evaluation starts on the closest previous checkpoint on the system-time Timeline, retrieves the bitmap containing the tuples valid at this point and traverses this index sequentially until it reaches the desired version. The performance is always clearly better than an in-memory table scan, typically by more than an order of magnitude faster. *M-IVTT* also shows the symmetric sawtooth pattern driven by the reconstruction of the VTTs, but the performance is significantly worse, at least an order magnitude worse than a table scan in the best case. *RR** performs better, since it can answer selection queries directly. Yet, the overhead of the tree index, including probes to overlapping regions, prevents it from outperforming the table scan. The query execution time of the commercial row-store *System A* is almost three orders of magnitude slower than Timeline. It always performs a full table scan, since the temporal filter is not selective enough to benefit from a conventional index.

T2: Application-Timeslice Only. The next query performs a timeslice to a given point in application-time for the CURRENT system-time version, where we use `FOR APPLICATION_TIME AS OF TIMESTAMP '[APP_TIME]'` as `TEMPORAL_CONDITION`. The results are shown in Figure 7.6(b).

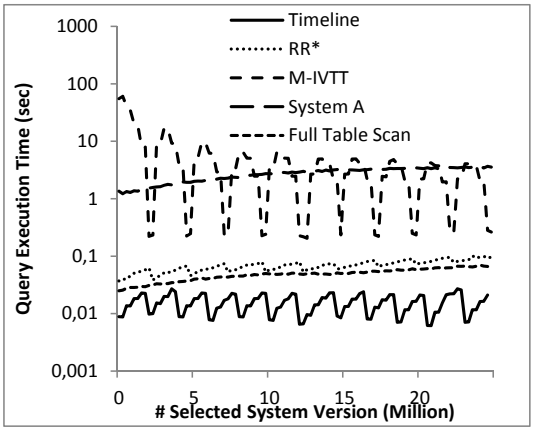
Timeline shows a constant performance, but is slower compared to T1. A timeslice to the CURRENT system-time needs to be performed, and the result is filtered according to the application-time condition. Timeline therefore performs similar to



(a) System-Time Only (T1)



(b) Application-Time Only (T2)



(c) Bitemporal (T3)

Figure 7.6: Timeslice [Large Data Set]

a full table scan. *M-IVTT* needs perform the same kind of VTT reconstruction for all app times and therefore shows a constant performance, but it again does not outperform a table scan. *RR** and *System A* perform similarly as for T1. Given the results sizes, these systems perform slightly faster and see an increased runtime for higher application-times as the size of the results grows.

T3: Timeslice for Both Time Dimensions. The final timeslice query constrains both the application-time and system-time dimension, combining the expressions of T1 and T2. The application-time is fixed to a point in the middle of the time range, and the system-time is varied on the x-axis.

Timeline shows the familiar sawtooth pattern caused by the checkpoints. Within a checkpoint the application-time Visibility Bitmaps and Event Maps are exploited to compute the application-timeslice for the system-time of the checkpoint. Next, the system-time Event Map is applied for tuples matching in application-time only, adding some additional cost compared to T1.

M-IVTT, *RR** and *System A* show very similar behavior as in T1 since the workload is influenced by the system-time constraints.

The results for timeslice show that Timeline is very competitive for the majority of workloads, whereas none of the competitors can clearly outperform table scans.

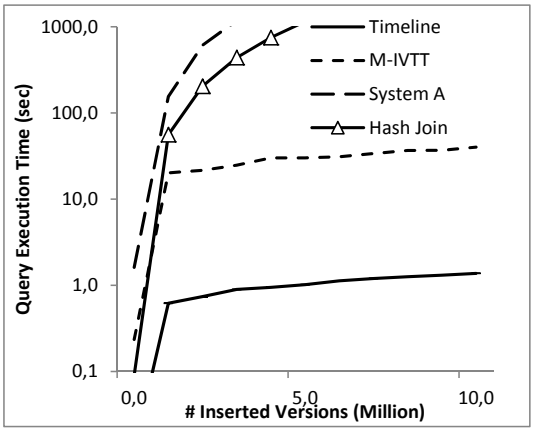
7.4.5 Experiment 3: Temporal Join

The third class of experiments examines temporal joins. This operation retrieves all tuples from different tables whose validity interval overlaps, i.e., which are visible at the same time with respect to a certain time domain. As such it stresses the support of the index structures for correlations, complementing the two previous experiments. We examine temporal joins with join conditions on (1) system-time, (2) application-time and (3) both time dimensions.

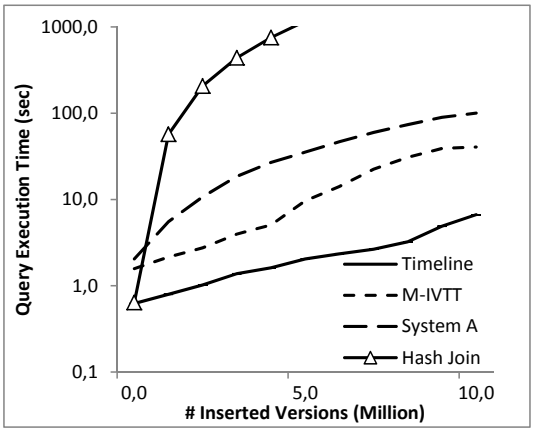
We utilize the following query, which is a non-temporal equijoin with a temporal join condition `TEMPORAL_CORRELATION` and a timeslice specification `TEMPORAL_CONDITION`: "Which expensive orders were open while the related customers had a low balance".

```
SELECT COUNT(*)
FROM customer c TEMPORAL JOIN orders o
  ON TEMPORAL_CORRELATION
TEMPORAL_CONDITION
WHERE c_custkey = o_custkey
  AND o_orderstatus = 'O' AND o_totalprice > 5000
  AND c_acctbal < 100
```

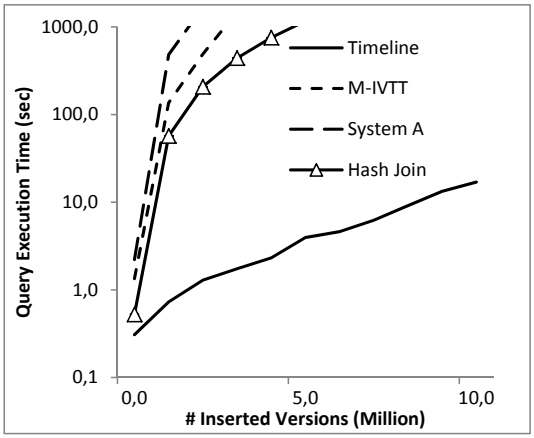
The results for this experiment are shown in Figure 7.7. The selectivities of the join predicates are depicted in Table 7.3.



(a) Join on System-Time (J1)



(b) Join on Application-Time (J2)



(c) Join on Two Dimensions (J3)

Figure 7.7: Temporal Join [Medium Data Set]

The following queries are measured on the *Large* data set:

J1: Temporal Join on System-Time. The first experiment (depicted in Figure 7.7(a)) shows the results for performing a temporal join over the system-time domain `ON c.SYSTEM_TIME OVERLAPS o.SYSTEM_TIME` and fix the application-time by `FOR APPLICATION_TIME AS OF TIMESTAMP '[APP_TIME]'`. When changing the size of the history, *Timeline* scales linearly with the number of versions since it performs a concurrent scan over both indexes, merges the time-ordered lifetime intervals with little overhead and directly evaluates the value join predicate, as outlined in Section 7.3.2. This way, the set of join candidates can be bounded effectively. *M-IVTT* can use the same algorithm, but needs to pay much higher index access cost. *System A* cannot exploit any of the temporal semantics and is slowed down by the combinatorial explosion of versions. *RR** is even worse, since the spatial join algorithms provided with it only consider temporal overlap but not value correlations, leading to timeout even in the Tiny workload. We also measured a standard *Hash Join* to complement the investigation with a join algorithm that focuses on the value domain. Similar to System A, its effectiveness is limited, as it only exploits the value domain.

J2: Temporal Join on Application-Time. In turn, the second experiment (depicted in Figure 7.7(b)) shows the results when we perform a temporal join over the application-time domain `ON c.APPLICATION_TIME OVERLAPS o.APPLICATION_TIME` and fix the system-time by using `FOR SYSTEM_TIME AS OF TIMESTAMP '[SYSTEM_TIME]'`, mirroring the workload of J1. *Timeline* fares slightly worse since it needs to pay the cost of reconstruction an application-time Event Map for the particular system-time. Given the different index organization, *M-IVTT* has now lower delta reconstruction cost, but it is still significantly more expensive than Timeline. *RR** and *Hash Join* fare roughly the same way as in J1, while *Systen Y* benefits from a different temporal selectivity.

J3: Temporal Join on System- and Application-Time. Our last experiment for temporal joins (shown in Figure 7.7(c)) correlates on both time dimensions and thus drops the timeslice present in the previous experiment, making it the most demanding workload due to further combinatorial effort. As a result, all approaches

Join Query	Foreign Key	Time Domains	Combined
J1	0.013%	78.8%	0.012%
J2	0.013%	99.6%	0.013%
J3	0.013%	78.5%	0.012%

Table 7.3: Join Selectivities on the Filtered Tables

see further cost increases. Yet, *Timeline* still copes best, as it is able to exploit both time and value constraints.

We also performed experiments where we varied the ratio between temporal and spatial selectivity, which we omit for space reasons. These experiments confirmed that the hybrid time/value approach for temporal joins supported by a time-ordered index provides the best tradeoffs. Although it is sometimes outperformed by other methods at extreme selectivity distributions, it always comes close to the best approach and usually beats its competitors.

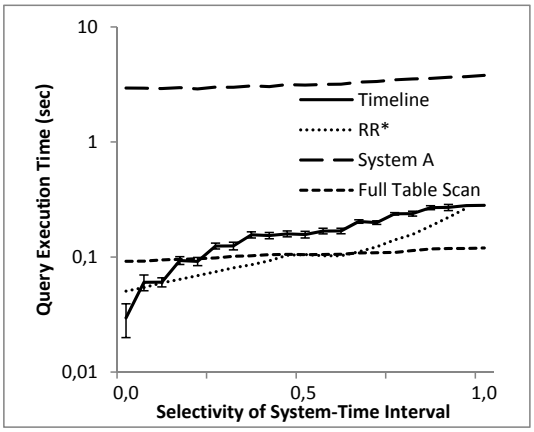
7.4.6 Experiment 4: Range Queries

Our last query performance experiment explores how well *Timeline* deals with arbitrary range selections. Given that it decouples the time dimensions, we may expect it to perform worse for arbitrary selections than a dedicated spatial index such as RR*.

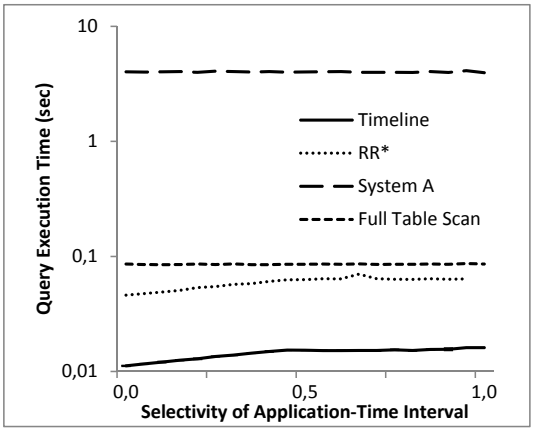
For this experiment we investigate the following query pattern which includes two time dimensions.

```
SELECT COUNT(*), AVG(ps_supplycost)
FROM partsupp
FOR APPLICATION_TIME BETWEEN
  '[APP_TIME_LOWER]' AND '[APP_TIME_UPPER]'
FOR SYSTEM_TIME BETWEEN
  '[SYS_TIME_LOWER]' AND '[SYS_TIME_UPPER]'
```

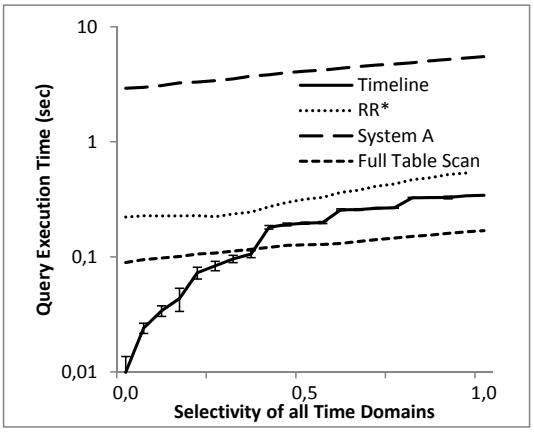
We examine 5 parameter settings: full range in one dimension, varying the other (leading to two experiments), fixing one dimension and varying the other (again leading to two experiments) and finally varying both dimension in concert. Selected results are shown in Figure 7.8, and we will discuss all of them here: Figure 7.8(a)/R1 yields the full application-time and varies the size of the system-time interval by decreasing the lower interval bound `[SYS_TIME_LOWER]`. Given the high selectivity of this workload, table scans are only outperformed for small system-time intervals. *Timeline* holds up rather well against *RR**, even beating it at low selectivities. R2 (not shown) inverts this workload by taking the full system-time range and varying application-time. Given its design *Timeline* cannot directly support this query, and we need to rely on scans. R3 (not shown) fixes the application-time to a point and varies the system-time, leading to results very similar to R1. Figure 7.8(b)/R4 fixes the system-time and varies the application-time range, allowing *Timeline* to work on a system-time Event Map and thus outperform all competitors. Finally, in Figure 7.8(c)/R5, we change both time ranges simultaneously. *Timeline* scales well, as it can benefit from its system-time index.



(a) Vary System Time, full Application Time (R1)



(b) Fixed System Time, vary Application Time (R4)



(c) Vary System Time, vary Application Time (R5)

Figure 7.8: Range Queries [Large Data Set]

	Timeline	M-IVTT	RR*	System A
Tiny Data Set	0.9	1.6	0.25	1.8
Medium Data Set	3.8	268.9	33.9	32.2
Large Data Set	7.8	504.7	85.0	128.8

Table 7.4: Index Construction Time (sec)

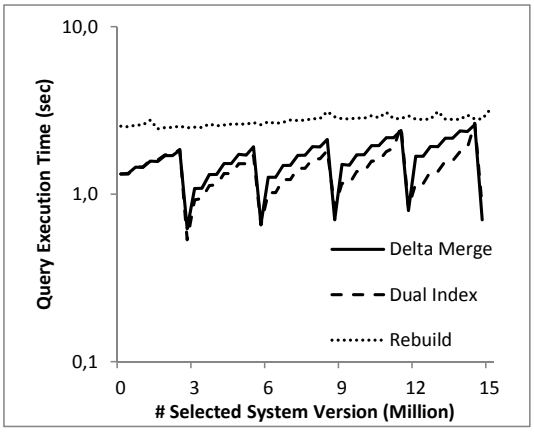


Figure 7.9: Temporal Aggregation Query for Alternative Ways of Index Construction [Large Data Set]

In summary, Timeline provides rather strong support for temporal range queries, keeping up with dedicated indexes and outperforming table scans under many settings.

7.4.7 Experiment 5: Index Creation Time

One of the key goals of Timeline is the ability to quickly create indexes when needed, in particular for two scenarios: 1) Building an index from scratch when loading data 2) Creating the appropriate application-time index. Table 7.4 shows the time of the index creation for the PARTSUPP table and different sizes of the data set. As it can clearly be seen, Timeline is the only index structure that scaled almost linearly and is fast enough to allow ad-hoc index creation for almost all workloads. In contrast to *M-IVTT* and *RR**, it only requires two scans instead of sorting or tree operations. The tradeoffs on generating intermediary application-time Event Maps (as outlined in Section 7.3.1) are more complex: Figure 7.9 compares different reconstruction approaches for temporal aggregation. Building a Timeline Index from scratch is always slower than incorporating existing snapshots. Merging the Event Map with the changes (*Delta Merge*) it is often slightly outperformed by running adapted

	Memory Total (MB)	**Relative**
Uncompressed Table	3416	100%
Timeline without Chkpt.	113	3.3%
Timeline with 10 Chkpt.	774	22.8%
M-IVTT	2262	66.5%
RR*	1100	32.4%
System A with Indexes	282	*9.0%*

Table 7.5: Memory of PARTSUPP Table [Large Data Set]

operators on the snapshot and the changes separately (*Dual Index*), but allows us to keep the complexity of operators low. As such it is the best alternative. The cost between dips and peaks of around 1.5 seconds indicates the maximum cost of delta construction and merge. Comparing this value with the results of direct evaluations for timeslice in Figure 7.6(c) confirms our decision of only reconstructing an Event Map when needed, as the cost of scanning is around 0.2 seconds.

7.4.8 Experiment 6: Memory Consumption

Figure 7.5 shows the memory consumption for each index data structure when loading the PARTSUPP table of the *Large* data set. We have chosen 10 checkpoints and 10 Visibility Bitmaps for application-time per checkpoint as well as 10 VTTs for M-IVTT. The cost for *Timeline* is dominated by the number of checkpoints: without checkpoints it only requires around 3% of the space of the temporal table. Checkpoints drive up this cost – in our case with 10 checkpoints we end up at 23% of the temporal table. Despite never outperforming Timeline, *M-IVTT* requires significantly more storage. Likewise, *RR** is more expensive than a Timeline Index with checkpoints. The story for *System A* is quite complex due to the results of the index advisor: While the index for the *Large* PARTSUPP requires only around 9%, it also just supports timeslice. Furthermore, slight workload variations can lead to drastic changes in indexing, e.g., PARTSUPP for Medium triggers additional indexes due to different selectivities, requiring 51%.

7.5 Conclusion

With the Bitemporal Timeline Index a wide variety of temporal queries can be evaluated efficiently. Our initial proposal – the Timeline Index for system-time, which we introduced in Chapter 6 – is easy to update because new transactions generate an immutable, strictly monotone sequence of timestamps. As we add support for application-time, we need to consider updates of the time dimension, even for past events. Consequently, we cannot simply incrementally update an index on the application-time dimension.

In this chapter we proposed to store an application-time Timeline Index for the points in system-time only where we create a checkpoint. Using the idea of incremental updates, we can travel to the desired point in time on the application-time dimension. With these two data structures in place, we are able to implement the basic temporal operators timeslice, temporal aggregation and temporal join for the full bitemporal data model. Compared to the state-of-the-art algorithms we achieve orders of magnitude performance improvement for several queries. Overall, we are now able to address the full spectrum of real world requirements for temporal query processing.

We currently recompute the index for the application-time dimension for every query, but in the future we may keep this index for further queries and thereby reduce the construction overhead for the application-time dimension. Also, as temporal tables can become quite large, it may not always be feasible to keep all temporal data in main-memory. Consequently, we want to investigate how we can partition and distribute temporal tables and the corresponding index structures.

8
Conclusions

8.1 Summary

In this dissertation we evaluated alternative ways of providing a native implementation of various temporal operators in a commercial, in-memory column store database system such as SAP HANA.

Many business applications have been using temporal data for several years. Yet, as the corresponding use cases were not sufficiently covered by the database systems, the time domain had to be modeled by the developers on the application layer. Since this overhead results in bad performances and additional complexity, SAP and its customers have high demand for a native implementation of temporal operators in the database system.

In the first part of our work we analyzed the use cases of various SAP customers to understand their requirements for temporal features. This use case analysis resulted in the definition of the TPC-BiH benchmark. A comprehensive performance analysis of state-of-the-art commercial database systems revealed 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. In particular, complex operations such as temporal aggregation and temporal join cannot be executed efficiently by state-of-the-art commercial systems.

As column stores are very well-suited for analytical workloads, and main-memory is getting cheaper, we opted to store both current and previous versions of the data in a main memory column store. We compared three alternative memory layouts to store temporal data physically in a column store and evaluated the tradeoffs for various access patterns. We achieved the most balanced results with a hybrid approach, which combines both segments of data clustered by time and by space.

The requirement to physically reorganize data in favor of compression and to further improve performance was the motivation for developing a novel, universal index structure which supports a large variety of temporal operators. The *Timeline Index* is space-efficient, typically only a small percentage of the size of a temporal table, and a single instance of this index per temporal table is sufficient. Furthermore, it integrates naturally into an existing database system such as SAP HANA, thereby taking advantage of highly optimized code paths to scan data, parallelize queries, and utilize modern hardware. Query execution time is predictable and very fast: It beats all best-of-breed approaches in all our performance experiments with an in-memory column store; in some cases by orders of magnitude.

It turns out that most SAP applications and customers require a much richer, bitemporal data model that involves system-time in addition to one or possibly several application-time dimensions. In order to support the full bitemporal data

model of the SQL:2011 standard, we proposed the Bitemporal Timeline Index as an extension of the basic ideas of the Timeline Index. Comprehensive performance experiments with the TPC-BiH benchmark show that the Bitemporal Timeline Index significantly outperforms all existing commercial database systems, as well as all approaches that have been proposed in the research literature to process queries on bitemporal data.

8.2 Future Work

Avenues for future work include a parallel and distributed execution of temporal operators. On the one hand, distribution of temporal data to several nodes allows the system to handle more data. On the other hand, the distributed algorithms of temporal operations may result in a faster execution. In addition, we intend to investigate alternative strategies for processing temporal operators, for instance shared scans, as implemented in the Master's thesis by Köhl [53]. We plan to investigate further use cases for temporal data such as sliding window queries and time series.

So far we have implemented the Timeline Index and the temporal operators in a prototype database system based on the architecture of SAP HANA. The integration of the Timeline Index into a productive release of SAP HANA is currently in progress. The current release version of SAP HANA supports the timeslice operator for system-time only. Given the efficient implementation based on Timeline Index, new temporal operators such as temporal aggregation and temporal join will be included in SAP HANA soon. In future, all temporal operators shall be implemented based on the Timeline Index.

List of Tables

List of Figures

List of Algorithms

Bibliography

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. "Integrating Compression and Execution in Column-oriented Database Systems". In: *SIGMOD Conference*. 2006, pp. 671–682.

[2] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. "Column-Stores vs. Row-Stores: How Different are they Really?" In: *SIGMOD Conference*. 2008, pp. 967–980.

[3] G. Adelson-Velskii and E. M. Landis. "An Algorithm for the Organization of Information". In: *Proceedings of the USSR Academy of Sciences*. 1962.

[4] Christian S. Jensen et al. *A Consensus Test Suite of Temporal Database Queries*. Tech. rep. Dept. of Mathematics and Computer Science, Aalborg University, Denmark, 1993.

[5] Mohammed Al-Kateb, Alain Crolotte, Ahmad Ghazal, and Linda Rose. "Adding a Temporal Dimension to the TPC-H Benchmark". In: *TPCTC*. 2012, pp. 51–59.

[6] Mohammed Al-Kateb, Ahmad Ghazal, Alain Crolotte, Ramesh Bhashyam, Jaiprakash Chimanchode, and Sai Pavan Pakala. "Temporal Query Processing in Teradata". In: *EDBT*. 2013, pp. 573–578.

[7] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. "An Asymptotically Optimal Multiversion B-Tree". In: *VLDB J.* 5.4 (1996), pp. 264–275.

[8] Norbert Beckmann and Bernhard Seeger. "A Revised R*-Tree in Comparison with Related Index Structures". In: *SIGMOD Conference*. 2009, pp. 799–812.

[9] Jacov Ben-Zvi. "The Time Relational Model". AAI8219638. PhD thesis. 1982.

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

[11] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. "Dictionary-based Order-preserving String Compression for Main Memory Column Stores". In: *SIGMOD Conference*. 2009, pp. 283–296.

[12] Rasa Bliujute, Christian S. Jensen, Simonas Saltenis, and Giedrius Slivinskas. "Light-Weight Indexing of General Bitemporal Data". In: *SSDBM*. 2000, pp. 125–138.

[13] Rasa Bliujute, Christian S. Jensen, Simonas Saltenis, and Giedrius Slivinskas. "R-Tree Based Indexing of Now-Relative Bitemporal Data". In: *VLDB*. 1998, pp. 345–356.

[14] Michael H. Böhlen, Johann Gamper, and Christian S. Jensen. "Multi-dimensional Aggregation for Temporal Data". In: *EDBT*. 2006, pp. 257–275.

[15] Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. "Temporal Statement Modifiers". In: *ACM Trans. Database Syst.* 25.4 (2000), pp. 407–456.

[16] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. "The Skyline Operator". In: *ICDE*. 2001, pp. 421–430.

[17] Richard Cole et al. "The Mixed Workload CH-benCHmark". In: *DBTest*. 2011, p. 8.

[18] George P. Copeland and Setrag Khoshafian. "A Decomposition Storage Model". In: *SIGMOD Conference*. 1985, pp. 268–279.

[19] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David R. Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. "Building Peer-to-Peer Systems with Chord, a Distributed Lookup Service". In: *HotOS*. 2001, pp. 81–86.

[20] Anton Dignös, Michael H. Böhlen, and Johann Gamper. "Temporal Alignment". In: *SIGMOD Conference*. 2012, pp. 433–444.

[21] Margaret H. Dunham, Ramez Elmasri, Mario A. Nascimento, and Marion Sobol. *Benchmark Queries for Temporal Databases*. Tech. rep. Southern Methodist University, 1993.

[22] Herbert Edelsbrunner. "A New Approach to Rectangle Intersections Part I". In: *International Journal of Computer Mathematics* 13.3-4 (1983), pp. 209–219.

[23] Ramez Elmasri, Gene T. J. Wuu, and Yeong-Joon Kim. "The Time Index: An Access Structure for Temporal Data". In: *VLDB*. 1990, pp. 1–12.

[24] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. "The SAP HANA Database – An Architecture Overview". In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 28–33.

[25] Raphael A. Finkel and Jon Louis Bentley. "Quad Trees: A Data Structure for Retrieval on Composite Keys". In: *Acta Inf.* 4 (1974), pp. 1–9.

[26] Johann Gamper, Michael H. Böhlen, and Christian S. Jensen. "Temporal Aggregation". In: *Encyclopedia of Database Systems*. 2009, pp. 2924–2929.

[27] Dengfeng Gao, Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. "Join Operations in Temporal Databases". In: *VLDB J.* 14.1 (2005), pp. 2–29.

[28] Lukasz Golab, Shaveen Garg, and M. Tamer Özsu. "On Indexing Sliding Windows over Online Data Streams". In: *EDBT*. 2004, pp. 712–729.

[29] Jim Gray. *Benchmark Handbook: For Database and Transaction Processing Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[30] Jim Gray. "Transaction Research History and Challenges, Invited talk". In: *SIGMOD Conference*. 2006.

[31] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. "Quickly Generating Billion-Record Synthetic Databases". In: *SIGMOD Conference*. 1994, pp. 243–252.

[32] Antonin Guttman. "R-Trees: A Dynamic Index Structure for Spatial Searching". In: *SIGMOD Conference*. 1984, pp. 47–57.

[33] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. "Generalized Search Trees for Database Systems". In: *VLDB*. 1995, pp. 562–573.

[34] Sándor Héman, Marcin Zukowski, Niels J. Nes, Lefteris Sidirourgos, and Peter A. Boncz. "Positional Update Handling in Column Stores". In: *SIGMOD Conference*. 2010, pp. 543–554.

[35] Vertica Systems Inc. *The Vertica Analytic Database Technical Overview White Paper*. Tech. rep. Vertica, 2012.

[36] Christian S. Jensen, Leo Mark, Nick Roussopoulos, and Timos K. Sellis. "Using Differential Techniques to Efficiently Support Transaction Time". In: *VLDB J.* 2.1 (1993), pp. 75–111.

[37] Christian S. Jensen and Richard T. Snodgrass. "Bitemporal Relation". In: *Encyclopedia of Database Systems*. 2009, pp. 243–244.

[38] Christian S. Jensen and Richard T. Snodgrass. "Temporal Data Management". In: *IEEE Trans. Knowl. Data Eng.* 11.1 (1999), pp. 36–44.

[39] Kevin Jernigan. *Oracle Total Recall with Oracle Database 11g Release 2*. Tech. rep. Oracle, 2009.

[40] Khaled Jouini and Geneviève Jomier. "Avoiding Version Redundancy for High Performance Reads in Temporal Databases". In: *DaMoN*. 2008, pp. 41–46.

[41] Patrick P. Kalua and Edward L. Robertson. *Benchmarking Temporal Databases - A Research Agenda*. Tech. rep. Indiana University, Computer Science Department, 1995.

[42] Martin Kaufmann. "Storing and Processing Temporal Data in a Main Memory Column Store". In: *PVLDB* 6.12 (2013), pp. 1444–1449.

[43] Martin Kaufmann, Peter M. Fischer, Donald Kossmann, and Norman May. "A Generic Database Benchmarking Service". In: *ICDE*. 2013, pp. 1276–1279.

[44] Martin Kaufmann, Peter M. Fischer, Norman May, and Donald Kossmann. "Benchmarking Bitemporal Database Systems: Ready for the Future or Stuck in the Past?" In: *EDBT*. 2014, pp. 738–749.

[45] Martin Kaufmann, Peter M. Fischer, Norman May, Andreas Tonder, and Donald Kossmann. "TPC-BiH: A Benchmark for Bitemporal Databases". In: *TPCTC*. 2013, pp. 16–31.

[46] Martin Kaufmann, Donald Kossmann, Norman May, and Andreas Tonder. *Benchmarking Databases with History Support*. Tech. rep. SAP AG, 2013.

[47] Martin Kaufmann, Donald Kossmann, Norman May, and Andreas Tonder. *SQL Extension for History Tables*. Tech. rep. SAP AG, 2013.

[48] Martin Kaufmann, Amin Amiri Manjili, Stefan Hildenbrand, Donald Kossmann, and Andreas Tonder. "Time Travel in Column Stores". In: *ICDE*. 2013, pp. 110–121.

[49] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, Franz Färber, and Norman May. "Timeline Index: A Unified Data Structure for Processing Queries on Temporal Data in SAP HANA". In: *SIGMOD Conference*. 2013, pp. 1173–1184.

[50] Martin Kaufmann, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, and Franz Färber. "Comprehensive and Interactive Temporal Query Processing with SAP HANA". In: *PVLDB* 6.12 (2013), pp. 1210–1213.

[51] Nick Kline and Richard T. Snodgrass. "Computing Temporal Aggregates". In: *ICDE*. 1995, pp. 222–231.

[52] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 1998.

[53] Florian Köhl. "Towards Processing Temporal Data in Crescando". MA thesis. ETH Zurich, Sept. 2013.

[54] Krishna G. Kulkarni and Jan-Eike Michels. "Temporal Features in SQL:2011". In: *SIGMOD Record* 41.3 (2012), pp. 34–43.

[55] Anil Kumar, Vassilis J. Tsotras, and Christos Faloutsos. "Designing Access Methods for Bitemporal Databases". In: *IEEE Trans. Knowl. Data Eng.* 10.1 (1998), pp. 1–20.

[56] David B. Lomet, Roger S. Barga, Mohamed F. Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. "Transaction Time Support Inside a Database Engine". In: *ICDE*. 2006, p. 35.

[57] David B. Lomet, Mingsheng Hong, Rimma V. Nehme, and Rui Zhang. "Transaction Time Indexing with Version Compression". In: *PVLDB* 1.1 (2008), pp. 870–881.

[58] David B. Lomet and Feifei Li. "Improving Transaction-Time DBMS Performance and Functionality". In: *ICDE*. 2009, pp. 581–591.

[59] David B. Lomet and Betty Salzberg. "Access Methods for Multiversion Data". In: *SIGMOD Conference*. 1989, pp. 315–324.

[60] Amin Amiri Manjili. "Timeline Index: A Unified Data Structure for Processing Queries on Temporal Data". MA thesis. ETH Zurich, Sept. 2012.

[61] Peter Muth, Patrick E. O'Neil, Achim Pick, and Gerhard Weikum. "The LHAM Log-Structured History Data Access Method". In: *VLDB J.* 8.3-4 (2000), pp. 199–221.

[62] Mario A. Nascimento. "Efficient Indexing of Temporal Databases via B+-Trees". PhD thesis. 1996.

[63] Mario A. Nascimento and Margaret H. Dunham. "Indexing Valid Time Databases via B+-Trees". In: *IEEE Trans. Knowl. Data Eng.* 11.6 (1999), pp. 929–947.

[64] Mario A. Nascimento, Margaret H. Dunham, and Ramez Elmasri. "M-IVTT: An Index for Bitemporal Databases". In: *DEXA*. 1996, pp. 779–790.

[65] Oracle. *Oracle Database Development Guide, 12c Release 1 (12.1)*. 2013.

[66] Gultekin Özsoyoglu and Richard T. Snodgrass. "Temporal and Real-Time Databases: A Survey". In: *IEEE Trans. Knowl. Data Eng.* 7.4 (1995), pp. 513–532.

[67] Hasso Plattner. "A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database". In: *SIGMOD Conference*. 2009, pp. 1–2.

[68] Ravi Rajamani. *Oracle Total Recall / Flashback Data Archive*. Tech. rep. Oracle, 2007.

[69] Sridhar Ramaswamy. "Efficient Indexing for Constraint and Temporal Databases". In: *ICDT*. 1997, pp. 419–431.

[70] Betty Salzberg and Vassilis J. Tsotras. "Comparison of Access Methods for Time-Evolving Data". In: *ACM Comput. Surv.* 31.2 (1999), pp. 158–221.

[71] Cynthia M. Saracco, Matthias Nicola, and Lenisha Gandhi. *A Matter of Time: Temporal Data Management in DB2 10*. Tech. rep. IBM, 2012.

[72] Han Shen, Beng Chin Ooi, and Hongjun Lu. "The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases". In: *ICDE*. 1994, pp. 274–281.

[73] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. "Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth". In: *SIGMOD Conference*. 2012, pp. 731–742.

[74] Richard T. Snodgrass. *Developing Time-oriented Database Applications in SQL*. Morgan Kaufmann, 1999.

[75] Richard T. Snodgrass et al. "TSQL2 Language Specification". In: *SIGMOD Record* 23.1 (1994), pp. 65–86.

[76] Michael D. Soo, Richard T. Snodgrass, and Christian S. Jensen. "Efficient Evaluation of the Valid-Time Natural Join". In: *ICDE*. 1994, pp. 282–292.

[77] Michael Stonebraker. "The Design of the POSTGRES Storage System". In: *VLDB*. 1987, pp. 289–300.

[78] Michael Stonebraker et al. "C-Store: A Column-oriented DBMS". In: *VLDB*. 2005, pp. 553–564.

[79] Yufei Tao and Dimitris Papadias. "Efficient Historical R-trees". In: *SSDBM*. 2001, pp. 223–232.

[80] Yufei Tao and Dimitris Papadias. "MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries". In: *VLDB*. 2001, pp. 431–440.

[81] Reginald P. Tewarson. *Sparse Matrices (Part of the Mathematics in Science & Engineering series)*. Academic Press Inc., 1973.

[82] Panagiotis Vagenas. "Efficient Temporal Data Processing with the Timeline Index". MA thesis. ETH Zurich, Apr. 2013.

[83] Paul Werstein. "A Performance Benchmark for Spatiotemporal Databases". In: *In: Proc. of the 10th Annual Colloquium of the Spatial Information Research Centre*. 1998, pp. 365–373.

[84] David Wheeler. *Temporal Tables Extension for PostgreSQL*. Website. http://pgxn.org/dist/temporal_tables/1.0.0/. 2014.

[85] Jun Yang and Jennifer Widom. "Incremental Computation and Maintenance of Temporal Aggregates". In: vol. 12. 3. 2003, pp. 262–283.

[86] Alessandro Zala. "Algorithms for Efficient Evaluation of Queries on Historic Data". MA thesis. ETH Zurich, May 2012.

[87] Donghui Zhang, Alexander Markowetz, Vassilis J. Tsotras, Dimitrios Gunopulos, and Bernhard Seeger. "On Computing Temporal Aggregates with Range Predicates". In: *ACM Trans. Database Syst.* 33.2 (2008).

[88] Donghui Zhang, Vassilis J. Tsotras, and Bernhard Seeger. "Efficient Temporal Join Processing Using Indices". In: *ICDE*. 2002, pp. 103–113.