ParTime: Parallel Temporal Aggregation

Markus Pilman, Martin Kaufmann, Florian Köhl
Systems Group, ETH Zürich, Switzerland
{mpilman,martinka,koehlf}@ethz.ch

Donald Kossmann
Microsoft Research, Redmond, WA, USA
donaldk@microsoft.com

Damien Profeta
Amadeus Group, Sophia Antipolis, France
damien.profeta@amadeus.net

ABSTRACT

This paper presents ParTime, a parallel algorithm for temporal aggregation. Temporal aggregation is one of the most important, yet most complex temporal query operators. It has been extensively studied in the past, but so far there has only been one attempt to parallelize this operator. ParTime supports data parallelism and has a number of additional advantages: It supports the full bi-temporal data model, it requires no a-priori indexing, it supports shared computation, and it runs well on modern hardware (i.e., NUMA machines with large main memories). We implemented ParTime in a parallel database system and carried out comprehensive performance experiments with a real workload from the airline industry and a synthetic benchmark, the TPC-BiH benchmark. The results show that ParTime significantly outperforms any other available temporal database system. Furthermore, the results show that ParTime is competitive as compared to the Timeline Index, the best known technique to process temporal queries from the research literature and which is based on pre-computation and indexing.

1. INTRODUCTION

Support for temporal data is a core feature of many data management systems. There are a number of application areas that rely heavily on temporal data. Examples are the financial industry, health care, and the travel industry. Correspondingly, the SQL standard has been mandating support for temporal data processing since 2011. As of today, most major relational database systems comply and support temporal data and queries (e.g., IBM DB2, Oracle, SAP HANA, and Teradata).

Temporal databases and the efficient implementation of temporal operators have been studied extensively since the early Nineties [22]. There are a number of different temporal operators such as time travel, temporal join, windowing, and temporal aggregation. In general, temporal databases are larger than non-temporal databases because every tuple can come in several versions. Furthermore, temporal operators are typically more complex than non-temporal operators. As a result, parallelization is important in order to achieve acceptable response times in temporal database systems.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2903732

The focus of this work is on the temporal aggregation operator. Temporal aggregation is a critical operator in many application domains. More specifically, temporal aggregation is critical in the particular workload from the travel industry that motivated this work. In that industry, for instance, analysts are interested to plot the number of available seats of all flights for a certain connection over time. It turns out that temporal aggregation is one of the most expensive temporal operators, and it has been one of the most difficult operators to optimize. As a result, there has been extensive prior work on this operator in the past; e.g., [3, 26, 16, 13] to name just a few. Because of its high cost, it is particularly important to parallelize this operator. To the best of our knowledge, however, there has been only one attempt by Gendoranoy et al. [9] to parallelize this operator. The reason is that most approaches to implement termporal aggregation are based on tree traversals and these approaches do not parallelize well as shown in Gendoranoy's work. In addition, tree-based approaches do not perform well in modern, main-memory and NUMA-aware parallel database systems as shown in [5].

This paper proposes the first parallel temporal aggregation algorithm with (almost) linear speed up, called ParTime. ParTime is a two pass algorithm that is based on the simple insight that the impact that a record has on the aggregate value can be computed independent of any other record. Based on this insight, ParTime is able to compute this impact of all records in parallel and in any given order. In the first pass, ParTime scans the whole table and computes the impact of each record at every point in time. ParTime executes this scan in parallel and there are no constraints on how the table is partitioned or the degree of parallelism. The second pass merges the results of the first pass, thereby computing the aggregation result for each point in time. This second pass works like the merge phase of an ordinary (non-temporal) sort-based `GROUP BY` operator [11]. In principle, this second pass can be parallelized, too. It turns out, however, that this second pass is cheap for most practical queries so that we propose to carry out this second pass sequentially. For completeness, we studied queries for which this second pass was not cheap with synthetic benchmarks, but we have not seen any such cases in real workloads yet.

In addition to its simplicity and embarrassing parallelism (in the first pass), ParTime has a number of other important advantages. One important advantage is that it works on any kind of bi-temporal data (i.e., databases that have multiple dimensions of business time and transaction time). In contrast, many algorithms devised in the literature only work for one time dimension. In our experience, support for the bi-temporal data model is crucial. In the airline industry, for instance, data scientists are often interested to aggregate over the time when a booking was made and the departure time of a flight. Another advantage is that ParTime integrates well

into databases that were designed for modern NUMA hardware. ParTime is based on a parallel table scan that allows each core to separately and independently compute data from a different partition of the database with memory affinity. Last but not least, ParTime integrates well into any parallel database system. It is implemented by extending the scan operatior. In particular, ParTime can make use of *shared* parallel scans which are implemented in many modern parallel database systems [27, 6, 25]. As we will see, this feature is particularly important for high-throughput workloads with many concurrent non-temporal queries and updates. Due to data freshness and cost considerations, it is often not affordable to build a separate data warehouse just to execute temporal queries; instead, all queries and updates must be served by a single database system.

We implemented ParTime and integrated it into Crescando, a main-memory parallel database system deployed at Amadeus, one of the leading airline reservation systems [25]. We carried out comprehensive performance experiments with both the Amadeus workload and with a synthetic temporal database benchmark, the TPC-BiH benchmark [14]. The results confirm that Crescando with ParTime significantly outperforms any other available temporal database system. ParTime is also competitive to the best known way to implement temporal aggregation from the research literature [13]. This approach is based on the so-called Timeline Index, which effectively pre-computes the results of temporal aggregation; i.e., the Timeline Index can be seen as a materialized temporal aggregation view. The Timeline Index, thus, serves as a lower bound for the best possible performance that can be achieved and it shows that with sufficient parallelism, ParTime can reach this lower bound even without any materialization. Unfortunately, materialization and the Timeline Index are not viable for update-intensive workloads such as the Amadeus workload because of the prohibitively high cost to maintain the Timeline Index with every update; nevertheless, the Timeline Index is a great baseline to study the performance for read-only workloads.

In summary, this paper presents the following contributions:

- The ParTime method for parallel temporal aggregation. This paper shows how ParTime works for one-dimensional and multi-dimensional temporal aggregation. Additionally, we present a special optimization that is applicable to a popular class of temporal aggregation queries, so called *windowed temporal aggregation queries*.
- The implementation and integration of ParTime into Crescando, a parallel, main-memory database system. The most important contribution here is to show how the processing of temporal aggregation queries with ParTime seamlessly co-exists with the processing of other temporal and non-temporal queries and updates as part of a shared scan. It is only this embedding of ParTime into a shared scan that makes it possible to sustain the high query and update throughput of the Amadeus workload. No other database system that we are aware of is able to sustain this workload.
- The results of comprehensive performance experiments. These results study the speed-ups that can be achieved with ParTime and demonstrate the robustness of ParTime for complex and update-intensive workloads.

The remainder of this paper is organized as follows: Section 2 discusses related work. Section 3 presents the ParTime method. Section 4 shows how to integrate ParTime into Crescando. Section 5 summarizes the results of our performance experiments. Section 6 contains conclusions and possible avenues for future work.

2. RELATED WORK

Temporal databases have been the subject of extensive research since the 1990s. In particular, numerous algorithms for temporal aggregation have been devised. These algorithms often involve tree data structures which are either built on demand as an intermediate data structure or persisted as an index. Kline and Snodgrass [16] were among the first to develop sophisticated algorithms for temporal aggregation. Their best algorithm operates in two passes: In the first pass, it scans the data and builds a so-called *Aggregation Tree*. This tree encodes all the time intervals in which the aggregate function remains constant. In the second phase, this tree is traversed, thereby computing the aggregate value for each time interval. The initial work on Aggregation Trees was designed for sequential execution. Later, Gendranoy et al. studied how this approach could be parallelized and the effect of different partitioning schemes of the data, number of nodes, and placement of the intermediate results. Goa et al. [8] revisited these results for main-memory databases. The results of [8], however, show that overall the Aggregation Tree approach does not parallelize well; there is some improvement, but the speed-up is far from linear and the scalability is limited. A recent comprehensive performance study showed that even with a high degree of parallelism, the performance of the Aggregation Tree approach is not competitive [13]. From a bird's eye perspective, ParTime is similar to the original Aggregation Tree algorithm proposed in [16]: It also works in two passes and scans the table in the first pass in order to compute the affects of each tuple. However, the data structures used are different and what makes ParTime unique is the way it computes the contribution of each tuple.

One of the particular problems of the original Aggregation Tree algorithm is that the Aggregation Tree is not necessarily balanced and can degenerate into a linked list. In this case, the Aggregation Tree algorithm has quadratic complexity. To overcome this problem, Böhlen et al. [3] proposed an algorithm which is based on AVL trees for the upper and lower bounds of the time intervals stored in the Aggregation Tree. While that approach does lead to significantly improved performance and guarantees O(n * log n) complexity, [13] shows that even this approach is not competitive and is outperformed by the Timeline Index by at least one order of magnitude on modern hardware. That is why we used the Timeline Index as a baseline in the performance experiments (Section 5) and refer the interested reader to [13] for a comprehensive performance study of the most important algorithms for temporal aggregation in the research literature.

There has been a large variety of index structures for temporal databases [18]. One prominent example is the multi-version B-tree [2]. The multi-version B-tree can only index a single time dimension and it indexes both, the time dimension and a (non temporal) search key. Another example is the work by Nascimento et al. [19]. This work proposes a two-level bi-temporal indexing scheme that is based on a B+-tree on transaction time, in which every entry of a leaf node points to another tree that indexes business time. The state-of-the-art in this area, however, is the Timeline Index, which was shown to clearly outperform these indexes [13]. At the core of the Timeline Index is the *event map*, which is a pre-computed sorted list of points in time when versions of records became valid and invalid. Given this event map, computing the result of a temporal aggregation query involves only one scan of this highly compressed sorted list. To further speed the computation up, the Timeline Index features checkpoints, which materialize a bitmap with all active records for a specific point in time: This way, the scans can start at the appropriate checkpoint, rather than scanning through the whole event map from the very beginning.

The Timeline Index works particularly well for the *transaction time* dimension (see next section) which is naturally ordered. However, the Timeline Index has recently been amended to support the full bi-temporal data model [15].

In terms of query performance, the Timeline Index can be seen as a lower bound for what is possible today because it precomputes the results of temporal aggregation. Depending on checkpoints in the query interval, the Timeline Index has linear or even constant complexity. That is why we use it in Section 5 to assess the performance of ParTime for queries. However, the Timeline Index is not a viable solution for update-intensive workloads. For such workloads, maintaining the Timeline Index is prohibitively expensive. Another short-coming of the Timeline Index is that it cannot be partitioned. If the size of the Timeline Index grows beyond the size of a NUMA region or even beyond the main memory of a single machine, the performance of the Timeline Index degrades significantly.

Most vendors of commercial database systems started adding temporal features into their systems in the last few years. To the best of our knowledge, Teradata, IBM DB2, Oracle, and SAP HANA are the systems with the most advanced support for temporal database features. Of these systems, only Teradata has native support for temporal aggregation [1]. Teradata implements temporal operators with non-temporal operators. Specifically, Teradata uses the Temporal Statement Modifier approach devised by Böhlen et al. [4]. Teradata supports bi-temporal tables with at most one business time dimension. This restriction also applies to IBM DB2 [23]. HANA only supports transaction time. Oracle has been supporting temporal data for more than 10 years as part of Flashback [21]. With Oracle Version 12c, there is full support for bi-temporal tables. As shown in Section 5, ParTime significantly outperforms any commercial database system.

An important feature of ParTime is that ParTime integrates well into any system that makes use of scans and in particular shared scans. Crescando uses the ClockScan algorithm [25], but ParTime can be integrated into any other scan implementation. RedBrick was the first database system to rely heavily on shared scans in the late Nineties. In the meantime, there has been a great deal of work on shared scans; e.g., as part of MonetDB [27], Blink [20], Crescando [25], Aster Data [6], and SharedDB [10]. Based on all this work, it is widely believed that robust database performance for high-throughput workloads can only be achieved with such shared scans. To the best of our knowledge, however, ParTime is the first temporal database algorithm that tries to take advantage and integrates nicely into a shared scan.

3. PARTIME METHOD

This section presents the ParTime technique. There are two variants of the ParTime algorithm. First, a general variant that works for any kind of temporal aggregation. Second, an optimized variant that works for "windowed temporal aggregation queries." We show how both variants work for one-dimensional temporal data and for bi-temporal data with arbitrary dimensionality.

3.1 Running Examples

Figure 1 shows an example *Employee* table. For each employee, it records the *name*, *job description*, and *salary*. The *Employee* table of Figure 1 has two time dimensions, denoted as `BT` for business time (often also referred to as application time) and `TT` for transaction time (often also referred to as system time). The business time indicates when the information was true in the real world; the timestamps of the business time are set by the application. The transaction time indicates when the information was entered into the database; i.e., which transaction carried out the update. The timestamps of the transaction time are set automatically by the temporal database system as a side-effect of committing transactions and creating new versions of tuples.

	NAME	DESCR	SALARY	START_BT	END_BT	START_TT	END_TT
Row 0	Anna	CEO	10k	01-01-1993	∞	t_0	t_7
Row 1	Anna	CEO	10k	01-01-1993	01-06-1994	t_7	∞
Row 2	Anna	CEO	15k	01-06-1994	∞	t_7	∞
Row 3	Ben	Coder	5k	01-01-1993	∞	t_0	t_7
Row 4	Ben	Coder	5k	01-01-1993	01-06-1994	t_7	∞
Row 5	Ben	Manager	5k	01-06-1994	∞	t_7	t_{11}
Row 6	Ben	Manager	8k	01-06-1994	∞	t_{11}	∞
Row 7	Chris	Coder	5k	01-08-1993	∞	t_5	t_{16}
Row 8	Chris	Coder	5k	01-08-1993	01-01-1995	t_{16}	∞

Figure 1: Temporal *Employee* Table

In Figure 1, for instance, Row 0 indicates that at the beginning (after processing Transaction t_0), the database recorded the information that Anna's salary is 10k and it has been 10k since January 1, 1993. Then, Transaction t_7 updated this information indicating that Anna had a salary increase on June 1, 1994. As a result, t_7 updates Row 0: It sets the `END_TT` timestamp of Row 0 to t_7, thereby indicating that this version of Anna's record is only valid from Version t_0 to t_7. Furthermore, t_7 creates Rows 1 and 2: Row 1 indicates that Anna's salary was 10k from January 1, 1993 to June 1, 1994. Row 2 indicates that Anna's salary has been 15k since June 1, 1994.

The bi-temporal data model that we use for this work has been described in detail in [24]. (In the literature, other definitions can be found, too.) In practice, it is not unusual to have multiple business time dimensions. In the travel industry, for instance, an application could involve a business time dimension that keeps track of when the departure of a flight was scheduled and another business time dimension that records when the flight actually departed. However, there is always only one transaction time that describes the versioning of the temporal database. In the literature, tables with only transaction time are typically denoted as *temporal tables* whereas tables with transaction time and one or several business times are denoted as *bi-temporal tables*.

There are a number of specific temporal operators that can be applied to such (bi-) temporal tables [24]. The best known operator is the so-called *time-travel operator*. This operator fixes a certain point in time for each temporal dimension. For instance, one could ask about the total payroll on June 1, 1994 given Version t_3 of the database. The result of this query applied to the table of Figure 1 is 15k because only Rows 0 and 3 existed in Version t_3 of the database. We could also ask about the total annual payroll on June 1, 1994 given the current version of the database. In our example, the result is 28k as Rows 2, 6, and 8 qualify. The time travel operator has been standardized inSQL 2011 and it has been supported for a long time in the Oracle database system.

It turns out that the time-travel operator is straightforward to implement in parallel database systems like Crescando: It is a simple selection on the time dimensions. A more involved operator is *temporal aggregation*. Temporal aggregation computes an aggregate value for every point in time. Temporal aggregation can be one or multi-dimensional. For instance, Figure 2 shows the result of asking for the total payroll in 1995 for each version of the database. This query is an example of a one-dimensional temporal aggregation query because only the transaction time is varied (the business time is fixed to 1995).

START_TT	END_TT	SUM
t_0	t_5	15k
t_5	t_7	20k
t_7	t_{11}	25k
t_{11}	t_{16}	28k
t_{16}	∞	23k

Figure 2: Example 1, Result of One-dim Temp. Aggr.
Total payroll in 1995 for each version of the database.

START_BT	END_BT	START_TT	END_TT	SUM
01-01-1993	∞	t_0	t_5	15K
01-01-1993	01-08-1993	t_5	t_7	15K
01-08-1993	∞	t_5	t_7	20K
01-01-1993	01-08-1993	t_7	t_{11}	15K
01-08-1993	01-06-1994	t_7	t_{11}	20K
01-06-1994	∞	t_7	t_{11}	25K
01-01-1993	01-08-1993	t_{11}	t_{16}	15K
01-08-1993	01-06-1994	t_{11}	t_{16}	25K
01-06-1994	∞	t_{11}	t_{16}	28K
01-01-1993	01-08-1993	t_{16}	∞	15K
01-08-1993	01-06-1994	t_{16}	∞	20K
01-06-1994	01-01-1995	t_{16}	∞	28K
01-01-1995	∞	t_{16}	∞	23K

Figure 3: Example 2, Result of Two-dim Temp. Aggr.
Total payroll for each moment in time and each version.

Figure 3 shows the result of a query that asks for all the payroll values for every point in time and all versions of the database. This is an example of a two-dimensional temporal aggregation because both the business time and the transaction time are varied. As with SQL `GROUP BY` queries, the result size of temporal aggregation grows exponentially with the number of dimensions.

Finally, Figure 4 shows the result of a query that asks for the payroll at the beginning of each year given the current state of the database (i.e., all tuples with $\texttt{END\_TT} = \infty$). This query is one-dimensional (transaction time is fixed, business time is varied) and an example of a *windowed temporal aggregation query*.

Snodgrass defined the semantics of the temporal aggregation operator in [16]. Due to its complexity, many algorithms have been proposed in the literature (Section 2). As we will show, ParTime is the first parallel algorithm that scales linearly, takes advantage of shared scans, and does not require any pre-computation.

3.2 General Temporal Aggregation

ParTime is a two step algorithm. The first step scans the temporal table and computes the *deltas* of each record for the time when the record became valid and for the time when the record became invalid. The first step is the most expensive step and runs in parallel. That is, each core processes a different set of tuples independently. The second step *merges* the results of Step 1. Step 2 is more difficult to parallelize (Section 3.4). Fortunately, Step 2 is cheap for most practical queries so that ParTime typically achieves linear scale-up and scale-out, even without parallelizing Step 2 (Section 5).

3.2.1 Example 1: Generating Delta Maps (Step 1)

START_BT	END_BT	SUM
01-01-1993	31-12-1993	20k
01-01-1994	31-12-1994	28k
01-01-1995	31-12-1994	23k

Figure 4: Example 3, Result of Windowed Temp. Aggr.
Total payroll by year.

	NAME	DESCR	SALARY	START_BT	END_BT	START_TT	END_TT
Row 0	Anna	CEO	10k	01-01-1993	∞	t_0	t_7
Row 1	Anna	CEO	10k	01-01-1993	01-06-1994	t_7	∞
Row 2	Anna	CEO	15k	01-06-1994	∞	t_7	∞
Row 3	Ben	Coder	5k	01-01-1993	∞	t_0	t_7
Row 4	Ben	Coder	5k	01-01-1993	01-06-1994	t_7	∞
Row 5	Ben	Manager	5k	01-06-1994	∞	t_7	t_{11}
Row 6	Ben	Manager	8k	01-06-1994	∞	t_{11}	∞
Row 7	Chris	Coder	5k	01-08-1993	∞	t_5	t_{16}
Row 8	Chris	Coder	5k	01-08-1993	01-01-1995	t_{16}	∞

Figure 5: Generating Delta Maps (Example 1)

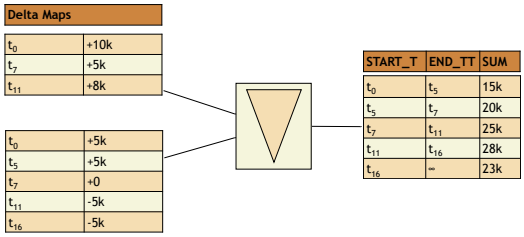


Figure 6: Merging Delta Maps (Example 1)

We illustrate the ParTime technique using the simple, one-dimensional temporal aggregation query of Example 1 (Figure 2). For the purpose of this example, we assume that we have two cores and that Core 1 processes all even rows (Rows 0, 2, 4, etc.) and Core 2 processes all odd rows (Rows 1, 3, etc.). ParTime works with any kind of partitioning; it works best if all cores process the same number of records so that *random* or *round-robin* are good partitioning schemes.

The key idea of ParTime is that every core creates a *delta map* as a result of scanning its partition of the temporal data. A *delta map* is a data structure that maps timestamps to deltas of aggregate values. That is, the delta map captures the affects of all tuples that became valid or invalid at each point in time. Furthermore, a *delta map* is ordered by timestamp, which speeds up the merge operation as part of Step 2. We used B-trees in our implementation of *delta maps*, but other data structures can be used, too, and may give even better performance.

Figure 5 shows the two delta maps that are computed for our running example and the query that asks for the total payroll for each version of the database (i.e., varying transaction time, fixing business time to 1995). To be more specific, Step 1 generates the following two deltas when processing Row 0 of Figure 1:

$$\langle t_0, +10k \rangle \langle t_7, -10k \rangle$$

These two entries indicate that at Version t_0 the total payroll needs to be increased by 10k as a result of Row 0 becoming visible in this version. Correspondingly, the payroll needs to be decreased by 10k at Version t_7 when the life time of Row 0 expires.

Because the life time of Row 2 is ∞, only one delta entry is generated for Row 2:

$$\langle t_7, +15k \rangle$$

This entry indicates that starting at Version t_7, the total payroll needs to be increased by 15k. When this entry is inserted into the delta map, it is immediately merged with the existing $\langle t_7, -10k \rangle$ entry (the delta map is indexed by timestamp). As a result, the delta map of Figure 5 shows only a single consolidated entry for t_7 that reflects the affects of both Rows 0 and 2; i.e.:

$$\langle t_7, +5k \rangle$$

```
function GENERATEDELTAMAP(chunk)
    dm ← new Btree()
    for all record in chunk do
        value ← record(column)
        validFrom ← record(validFromIdx)
        validTo ← record(validToIdx)
        DM-PUT(validFrom, value)
        if validTo ≠ ∞ then
            DM-PUT(validTo, −1 · value)
        end if
    end for
end function
```

Figure 7: Step 1 of General Aggregation (Sum)

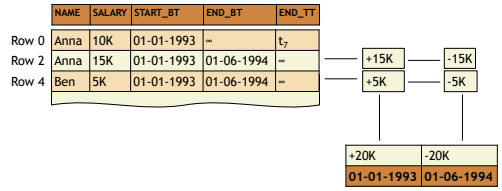


Figure 8: Generating Fixed-sized Delta Maps (Example 3)

In Example 1, Rows 1, 4, and 8 are ignored. These rows involve data that is not relevant for the query because these rows contain information for the years 1993 and 1994 only and the query asks specifically for salaries of the year 1995. Accordingly, these rows are filtered out before the ParTime algorithm takes effect and no delta entries are generated for these rows.

3.2.2 Example 1: Merging Delta Maps (Step 2)

Figure 6 shows the second step of the ParTime algorithm for general temporal aggregation; i.e., merging the delta maps generated in parallel in Step 1. This merge can be implemented in exactly the same way as a merge in a sort-based, regular (non-temporal) group-by operator [11]. That is, the timestamp is used as the group-by key and increments / decrements are simply added to generate the final payroll value for each version of the database system. Comparing Figures 6 and 2, it can be seen that the ParTime algorithm computes indeed the correct result.

3.2.3 Algorithm Description

Figure 7 gives the pseudo-code of Step 1 of the general ParTime algorithm for one-dimensional temporal aggregation if *sum* is used as an aggregate function. Assuming that B tree lookups are O(log n), the complexity of this algorithm is O(n * log n). We extended our B tree implementation to support a special *put* operation which adjusts an existing entry, if it exists, or creates a new entry, if the search key cannot be found. We omit details of Step 2 of the general ParTime algorithm because it is identical with any other merge operation (e.g., [11]).

The ParTime algorithm works very well for all aggregate functions that can be computed incrementally. Examples of such aggregate functions are *sum*, *product*, or *count*. Not all aggregate functions are incremental; e.g., *min*, *max*, or *median*. For these aggregate functions, it is not sufficient to keep a single aggregate value at all times in the delta map. Instead, the delta map keeps the set of values that became valid / invalid at each point in time. The merge step then involves keeping a priority queue to compute the aggregate value at each point in time.

```
function GENERATEDELTAMAP(chunk, size)
    dm ← new Array[size]
    for all record in chunk do
        value ← record(column)
        validFrom ← record(validFromIdx)
        validTo ← record(validToIdx)
        dm[validFrom] ← dm[validFrom] + value
        if validTo ≠ ∞ then
            dm[validTo] ← dm[validTo] − value
        end if
    end for
end function
```

Figure 9: Step 1 of Windowed Aggregation (Sum)

3.3 Windowed Queries

We will now turn to Example 3 of Section 3.1; Figure 4. In principle, this query can be executed using the very same general ParTime algorithm that was described in the previous section. However, we can do better and apply an important optimization for such windowed temporal aggregation queries.

The main observation is that the size of the result of the temporal aggregation is known in advance. This particular query asks for the total payroll at the beginning of each year. The database contains only information for three years, 1993, 1994, and 1995. This fact can be exploited to speed up the algorithm of Figure 7 significantly. Instead of maintaining the delta map as a dynamic B tree (or some other dynamic data structure), we can implement the delta map as an array and use the timestamps (i.e., years) to access the cells of this array directly. Fundamentally, the algorithm of Figure 7 is unchanged, but the *dm-put()* operations can be implemented in a much more efficient way by a simple array look-up. For completeness, Figure 9 shows the pseudo code: Compared to the algorithm of Figure 7, the only difference is that the delta map is implemented as an array instead of a B-tree.

In theory, this optimization is applicable to any temporal aggregation query, not only windowed queries. We can always calculate an upper limit for the number of versions in each time dimension. In the general case, however, this optimization is less effective because the array can become sparse and we would pay a high price in terms of memory footprint.

Figure 8 shows how windowed aggregation is applied to the third query of our running example. In this example, Row 0 is ignored because the query asks only for tuples of the *current version* of the database; i.e., records with `END_TT` = ∞.

3.4 Multi-dimensional Temporal Aggregation

The same two-step techniques, described in the previous two sections, can be applied to any multi-dimensional temporal aggregation query. Example 2 of Section 3.1 is a two-dimensional temporal aggregation query that varies business time *and* transaction time. The important difference to one dimensional temporal aggregation is that the deltas consider all temporal dimensions involved in the query. Another difference is that the deltas *pivot* on one temporal dimension and then capture the information along all other temporal dimensions. For instance, the ParTime algorithm generates the following two deltas for Row 0 and Example 2, if we pivot on the transaction time:

$$\langle +10k, \text{01-01-1993}, \infty, t_0\rangle, \langle -10k, \text{01-01-1993}, \infty, t_7\rangle$$

The first delta reads as follows: "At Transaction Time t_0, 10k need to be added to the total payroll for the business time duration 01-01-1993 to ∞." If we pivot for transaction time in this example,

```
function GENERATEDELTAMAP(chunk)
    dm ← new Btree()
    for all record in chunk do
        value ← record(column)
        pivotBegin ← record(idxOfBegin(0))
        pivotEnd ← record(idxOfEnd(0))
        validFrom ← array(numDimensions − 1)
        validTo ← array(numDimensions − 1)
        for i in (1..numDimensions − 1) do
            validFrom(i − 1) ← record(idxOfBegin(i))
            validTo(i − 1) ← record(idxOfEnd(i))
        end for
        fstKey ← concat(validFrom, validTo, [pivotBegin])
        DM-PUT(fstKey, value)
        if pivotEnd ≠ ∞ then
            sndKey ← concat(validFrom, validTo, [pivotEnd])
            DM-PUT(sndKey, −1 · value)
        end if
    end for
end function
```

Figure 10: Step 1 of Multi-dim. Aggregation (Sum)

ParTime generates two deltas because Row 0 expires along the transaction time dimension at t_7. If we pivot for business time, ParTime generates only one delta for Row 0 because Row 0 has no expiration date along the business time. In this case, ParTime generates the following delta. (As a convention, we keep the pivot time dimension last.)

$$\langle +10k, t_0, t_7, \text{01-01-1993} \rangle$$

This delta reads as follows which is a different yet equivalent interpretation of Row 0: "Starting at business time 01-01-1993, 10k need to be added to the total payroll for the transaction time duration t_0 to t_7." For correctness, any time dimension can be used as pivot dimension. For performance, it is best to choose the time dimension with the least distinct values (i.e., timestamps) because that will minimize the size of the delta map generated in Step 1. Typically, one of the business time dimensions has the least distinct values and our implementation of ParTime keeps statistics to pivot for the best possible time dimension.

Figure 10 shows the pseudo code for Step 1 of the multi-dimensional temporal aggregation. For simplicity, the pseudo code of Figure 10 always chooses time dimension 0 as a pivot; our implementation is more sophisticated in that regard. In Figure 10, $validFrom$ and $validTo$ are arrays that represent the start and end times for all temporal dimensions; for the pivot dimension, only the validFrom time is recorded. These times are filled in the inner loop which iterates over all temporal dimensions involved in the query. Other than that, this algorithm is the same as the one-dimensional algorithm shown in Figure 7.

Step 2 of the generalized, multi-dimensional ParTime algorithm implements the merge of delta maps in a similar way as the merge operation of a sort-merge join. Just as for the sort-merge join, Step 2 must carry out a Cartesian product for all overlapping delta entries. This Cartesian product results in the (correct) explosion of the query result of multi-dimensional temporal aggregation, as observed in Figure 3. Again, this exponential complexity in the number of time dimensions is fundamental to the temporal aggregation operator and *not* a weakness of the ParTime method, as explained in Section 3.1. Unlike the merge phase of a sort-merge join, Step 2 of the ParTime algorithm must aggregate the values of delta entries that are identical in all time dimensions.

In principle, Step 2 of the ParTime algorithm can be parallelized just as the merge phase of a sort-merge phase. For instance, this parallelization can be achieved with a multi-level merge operation as described in [11]. So far, we have not implemented such a parallel Step 2 of ParTime. The reason for this decision was that for all the temporal aggregation queries of the Amadeus workload that motivated this work, Step 2 is cheap and the cost of a temporal aggregation query is always dominated by Step 1. As shown in Section 5, however, the synthetic TPC-BiH benchmark does include cases in which Step 2 is indeed expensive and worth parallelizing. Studying how such a parallelization of Step 2 could improve performance is left for future work.

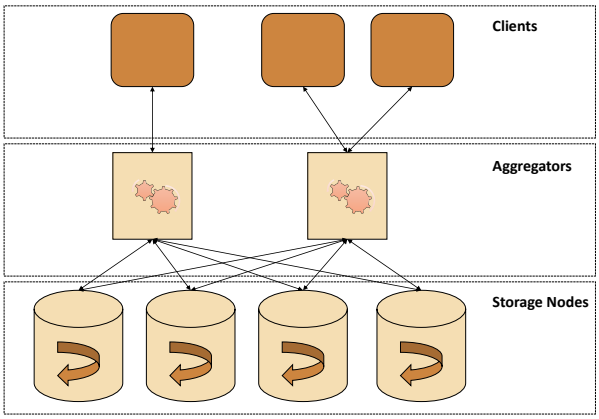


Figure 11: Architecture of Crescando

4. PARTIME AND SHARED SCANS

This section describes our implementation of ParTime in Crescando, Amadeus' parallel database system as one particular example system that can make use of ParTime. ParTime, however, can be integrated into any parallel database system: All that needs to be done is to extend the scan operator to generate delta maps and a separate operator to merge delta maps. Furthermore, the optimizer must be extended to determine the right degree of parallelism.

4.1 Overview

Figure 11 shows the architecture of Crescando; details of Crescando can be found in [25]. This architecture has been pioneered by Amazon as part of the Dynamo system [7]. Crescando is a shared-nothing database system with a two-tier architecture. At the lower layer are the storage nodes. The storage nodes persist the data. Each storage node keeps a different partition of a (temporal or non-temporal) table, thereby making use of horizontal partitioning of the data. Crescando supports any kind of partitioning scheme; in particular, it supports round-robin partitioning as used in the examples of the previous section.

Crescando is a main-memory database system. If the database grows, new storage nodes must be added. All read-requests are served completely out of main memory. Write-requests are logged to disk for crash recovery. In order to improve fault-tolerance, each storage node has a hot stand-by node (not shown in Figure 11) that fully replicates all the data and events of the storage node, thereby following state-machine replication [17].

The second tier of Crescando is composed of aggregator nodes. Crescando processes queries (and updates) in the following way: A client connects to one of the aggregators which coordinates the execution of the query. Any aggregator can be chosen for this purpose. In the default configuration, a client always connects to the same aggregator, but a round robin approach is also possible.

Query processing is then split between the storage nodes and the aggregator. Basic operations such as selections and projections are carried out by the storage nodes. Complex operations such as grouping are carried out by the aggregator. Furthermore, the aggregator collects the results from all storage nodes and produces the final query result and ships it to the client.

Depending on the partitioning scheme and the query, all or a subset of the storage nodes are involved in processing a query. For a round-robin partitioning scheme, *all* storage nodes are involved in *all* queries because it is not possible to infer that a particular storage node does not contain any relevant data that is needed to process a query. This worst-case scenario is not unusual in practice, and it is the scenario that was measured in the performance experiments reported in Section 5.

One problem of the architecture of Crescando are stragglers. If one storage node is significantly slower than all the others, then this storage node dominates the response time of all queries because the aggregator needs to wait for the results of *all* storage nodes. Crescando treats stragglers in the same way as failed nodes: It shoots them down and continues to operate with the hot standby node. These mechanisms are independent of ParTime and describing these mechanisms in detail is beyond the scope of this paper.

Another problem of running all queries on all storage nodes is *cost*. If the queries were executed one at a time, the storage nodes could easily become the bottleneck of the system. To address this challenge, Crescando makes heavy use of shared scans in the storage nodes. That is, a batch of queries and updates from multiple clients and aggregator nodes are processed by a storage node with a single scan, thereby processing the queries and updates on the fly. Specifically, Crescando makes use of the ClockScan algorithm described in [25] which was shown to give robust performance for demanding workloads with high update rates and complex queries. Crescando also supports indexes. For the experiments reported in Section 5, however, no indexes were used. That is, every query and update involved a full (parallel) table scan of the partition in all storage nodes. The larger the number of storage nodes, the smaller the partitions and correspondingly the faster the parallel scan.

Finally, aggregators are stateless in the architecture of Figure 11. Crescando does not make use of caching at the aggregator layer because the shared, parallel scans at the storage layer are so fast that caching does not help. When an aggregator fails, all its queries (and updates) are restarted. Furthermore, new aggregator nodes can be added at any point in time to sustain a higher load. Likewise, an aggregator can be shut down at any time if the load decreases.

4.2 Temporal Extensions

Initially, Crescando was not designed for temporal data and did not support the bi-temporal data model or temporal operators. Adding support for temporal data, however, was straightforward. As shown in Figure 1 temporal tables can be stored in the same way as regular, non-temporal tables in the storage nodes. The *start* and *end* timestamps of the temporal dimensions are stored and processed like any other column.

The time-travel operator can also be implemented in a straightforward way in Crescando. As mentioned in Section 3, time-travel involves the evaluation of range predicates on the time dimensions and range predicates were already supported in Crescando.

ParTime also fits nicely into the architecture of Figure 11. Step 1, generating the delta maps, is carried out by the storage nodes for all temporal aggregation queries. More specifically, the storage nodes generate the delta maps as part of the same shared scan that carries out all other queries and updates. Each storage node creates one delta map for each temporal aggregation query in addition to the result tuples for other queries and updates. The storage nodes generate these delta maps (and other results) in parallel independently of each other.

Step 2, merging the delta maps, is carried out by the aggregator that handles the temporal aggregation query. To this end, the aggregator waits until it has received the delta maps of all storage nodes. Since each query is assigned to exactly one aggregator, it is not easy to parallelize Step 2 in Crescando, even though Step 2 can be parallelized in principle as described in Section 3.4. Aggregation nodes also execute other complex operations such as joins, ranking and sorting, or other grouping and aggregation operations which are part of the query. It is also possible that a query involves several temporal aggregation operators. Within the aggregation nodes, the execution of the operators is pipelined using the iterator model. We are currently generalizing and improving the query processing capabilities of aggregator nodes, including parallel query operators, as part of the SharedDB project [10].

4.3 Query Optimization

Independent of ParTime, there is a need to optimize queries that involve several temporal and non-temporal operators. Query optimization is beyond the scope of this paper, but ParTime can be added to the compiler of an extensible temporal database system just like any other new algorithm [12, 11].

One special consideration for most parallel database systems is to optimize the degree of parallelism. As shown in Section 5, it is indeed advisable to carry out this optimization for ParTime, just like for any other parallel algorithm. Unfortunately, this optimization is not possible in Crescando. Crescando fixes the degree of parallelism as the number of storage nodes because every query is executed on all storage nodes. As a result, Crescando does not exploit the full flexibility and versatility of ParTime. Other parallel database systems might achieve even better performance with ParTime than Crescando by optimizing the degree of parallelism.

5. PERFORMANCE EXPERIMENTS AND RESULTS

This section studies the performance of ParTime. It compares the performance of Crescando with ParTime to two best-of-class commercial database systems and the best-known approach for temporal aggregation from the research literature. We conducted the experiments using a real workload from the travel industry and a synthetic database benchmark for bi-temporal data.

5.1 Software and Hardware Used

In order to assess the performance of ParTime, we used the following baselines in our performance experiments:

Timeline Index. [13]: The Timeline Index is the best known approach to compute temporal aggregation from the research literature. It precomputes all temporal information and keeps this information in a sorted way so that temporal aggregation can be carried out fast. We implemented the Timeline Index and the temporal aggregation algorithm based on the description of [13]. The big disadvantage of the Timeline Index is that it is a materialized view and it is costly to maintain the Timeline Index for update-intensive workloads. As a result, the Timeline Index is not practical for the Amadeus airline reservation application that motivated this work. For the purpose of this paper, the Timeline Index serves as a baseline to demonstrate the best possible query performance for read-only workloads and how close ParTime can come to that.

System D:. A commercial disk-based, general-purpose database system. We studied several general-purpose database systems and System D was the system that performed best throughout the experiments reported in this paper. We used the index advisor shipped with the product to generate indexes for the benchmark workload.

System M:. System M is a commercial main-memory database system which was specifically designed for analytics and has support for temporal data and transactions. Again, System M can be seen as the best of its class for temporal workloads and the queries and workloads studied in our experiments. By default, System M only creates indexes for the primary keys of tables. It turned out that this default was the best configuration for all our experiments.

A comprehensive performance study of other algorithms from the literature is given in [13]. Licensing agreements do not allow us to reveal the real product names of System D and System M.

We used Crescando in two modes:

- *Shared scans:* Exploit shared scans as described in Section 4. This mode is the default for Crescando.
- *No sharing:* Crescando without shared scans. In this mode, Crescando processes each query individually.

With No sharing, we could isolate the performance advantages that Crescando gets from making use of the ParTime algorithm vs. the performance advantages that Crescando has from exploiting shared scans. To the best of our knowledge, Systems D and M do not make use of shared scans. We measured throughput and response time. For all response time experiments, we ran Crescando in the No sharing mode. In all experiments reported in this paper, Crescando did not make use of any data-indexes, independent of the mode. Crescando did however use indexes on queries in sharing mode, as discribed in [25].

All experiments were carried out on a machine with 1.5 TB of main memory (DDR3, 1066MHz RAM) and four Intel Xeon E5-4650 processors with eight 2.7 GHz cores each (32 cores in total). The machine ran Linux (Kernel 3.14.17-100). In all experiments, the whole database fit in main memory and we warmed up the buffers in order to make sure that disk I/O was only required to log updates for recovery. Naturally, warming up the buffers was particularly important for System D.

Systems D and M made use of all 32 cores. To study the effectiveness of parallelization with ParTime, we varied the number of cores for Crescando. If not reported otherwise, Crescando used half of the cores to implement storage nodes and half of the cores as aggregator nodes (Figure 11). That is, in a configuration with 18 cores, Crescando used 9 cores as aggregators and partitioned the database across the other 9 cores. Temporal aggregation with the Timeline Index does not allow for parallelization so that all response time experiments with the Timeline Index were carried out with a single core. In all cases, however, we made sure that the allocated memory was close to the used cores to the extent possible. This NUMA-awareness was critical to achieve good performance for all four systems. Crescando and our implementation of the Timeline Index were in C++.

5.2 Benchmark Environment

We studied a real workload from the Amadeus airline reservation system. The results of the experiments with this workload gave insights into the end-to-end performance of a temporal database system for a complex application that includes various types of temporal queries, non-temporal queries, and a high update rate. In order to specifically study the effects of ParTime, we also carried out experiments with a synthetic benchmark.

Query	Freq.	Type	Description
ta1	1%	Temp.Aggr.	Number of Open Flights grouped by transaction time
ta2	1%	Temp.Aggr.	Number of Open Flights grouped by business time
other temporal	8%	Time Travel, Range	number of bookings at a certain time, bookings by day, etc.
other	90%	Non-temp.	e.g., number of bookings for a given flight and airline, passenger lists, etc.

Table 1: Queries of Airline Res. System

5.2.1 *Amadeus Workload*

Table 1 gives an overview of the queries of the Amadeus workload. In that workload, there are only two kinds of temporal aggregation queries, denoted as *ta1* and *ta2*. The first kind of these queries involves transaction time and asks, for instance, how the number of bookings for a specific flight evolved over time. The second kind of temporal aggregation queries of this system involves business time (e.g., the validity of a ticket) and queries such as the number of open and valid tickets over time. While both of these kinds of queries are quite complex, it is important to note that these queries are rare: Only about 2 percent of the queries of this workload involve temporal aggregation. 8 percent of the queries involve other temporal operators and the bulk of the queries are non-temporal (i.e., current time in all temporal dimensions).

The airline reservation workload is update-intensive. In addition to these queries, it involves 250 updates per second. These updates include inserts (e.g., new bookings) and updates (e.g., setting the frequent-flyer number, registering dietary requirements, etc.).

The database of the airline reservation system contains 2.4 Billion bookings. On average, a booking has five versions, but there is skew and some bookings are updated much more often than others. The information in the bookings table is encoded in a highly compressed way (using flags and fixed-length codes) so that the size of the database is only 300 GB (without indexes) and easily fits into main memory. Unfortunately, System D and System M were not able to execute the temporal aggregation queries on the full 2.4 Billion bookings; the queries timed out. As a result, we also carried out experiments on a 1% subset of 24 Million bookings.

5.2.2 *TPC BiH Benchmark*

In order to focus better on the performance of temporal operators and study a wider range of different kinds of temporal queries, we used the TPC-BiH benchmark [14]. This benchmark also allowed us to compare ParTime to the Timeline Index. We could not implement the Amadeus workload with the Timeline Index because the Amadeus workload requires a full-fledged database system with comprehensive support for the SQL standard and our prototype implementation of the Timeline Index does not have that. SAP is currently working on an implementation of the Timeline Index for HANA; however, that work has not yet been released.

Table 2 gives an overview of the queries of the TPC-BiH benchmark. All of these queries can be executed on all three systems and with our implementation of the Timeline Index. The most relevant queries for this work are queries r_1 to r_4. These are four different temporal aggregation queries that differ on whether they aggregate over transaction time and business time and that vary the time range. Queries r_1 and r_2 involve the whole database (without any selection or window), similar to Examples 1 and 2 in Section 3. Queries r_3 and r_4 are windowed temporal aggregation queries, similar to Example 3 in Section 3.

Query	Type	Description
t2	Time Travel	What was the total revenue of all orders at a given business time known as recorded at a certain previous time in history?
t3_sys	Time Travel	Compare the estimated revenue of the open orders at a given business time recorded at two different times in history?
t3_app	Time Travel	Compare the estimated revenue of the open orders at two given business times?
t6_sys	Time Travel	What is the average revenue per customer over business time, at a given time in history?
t6_app	Time Travel	What is the average order revenue per customer over history at a given point in application?
t8	Time Travel	For a given airline, how much time before the departure is the booking done in average?
t9	Time Travel	Flight Bookings per Point in System Time (time interval)
k1_sys	Key-in-Time	How did a given order as it is valid at a certain business time evolve in history?
k1_app	Key-in-Time	How did a given order as it is valid at a certain system time evolve in history?
r1	Temp.Aggr.	Which customers moved to US and still live there as registered within full system time?
r2	Temp.Aggr.	Which customers moved to US and still live there as registered within full business time?
r3	Temp.Aggr.	Which customers moved to US and still live there as registered within a system time interval?
r4	Temp.Aggr.	Which customers moved to US and still live there as registered within a business time interval?

Table 2: TPC-BiH Queries

The TPC-BiH benchmark database is created using the TPC-H benchmark database as a starting point (Version 0) and then running TPC-C style transactions in order to generate versions of the database. Each transaction creates a new version of the database. Like TPC-H, the TPC-BiH benchmark specifies a scaling factor that determines the size of the database. We used SF=1 to generate a small database of 5 GB of raw data (without indexes) and SF=100 for a large database of 312 GB of raw data. Systems D and M timed out for almost all queries on the large database so we only show the results for those systems on the small database.

5.3 Amadeus Workload

5.3.1 *Exp 1: No Sharing, Small Database*

Figure 12 shows the throughput of Systems D, M, and Crescando with ParTime for the Amadeus workload. For this experiment, we used a read-only variant of the Amadeus workload. That is, we replayed the queries from the traces we got from the airline reservation application and did not replay the updates. Furthermore, we used a small subset of the original booking database (24 Million bookings, 3 GB) because the temporal aggregation queries timed out for Systems D and M on the full 300 GB data set. Our baselines, Systems D and M, used all 32 cores throughout this experiment; both systems showed robust performance with 32 cores and we could not find a good way to vary the number of cores for these systems. For Crescando with ParTime, we varied the number of cores in order to see how it scales with the number of cores. In order to better compare ParTime with Systems D and M in this initial experiment, we ran Crescando in the No sharing mode. (The next subsection studies the importance of shared scans.)

Figure 12 shows that System M has the highest throughput and ParTime beats System D, even if ParTime runs only on two cores and System D runs on 32 cores. To better understand these results, Figure 13 shows the response times of two representative temporal

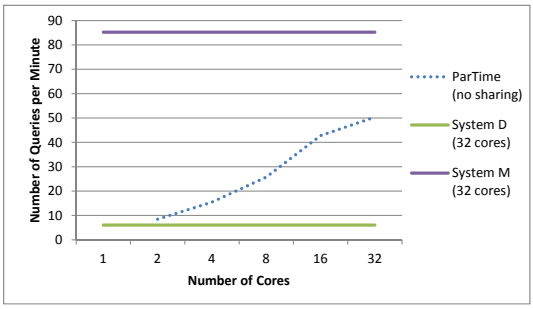


Figure 12: Tput: Amadeus, Small DB, Vary Cores, No Sharing

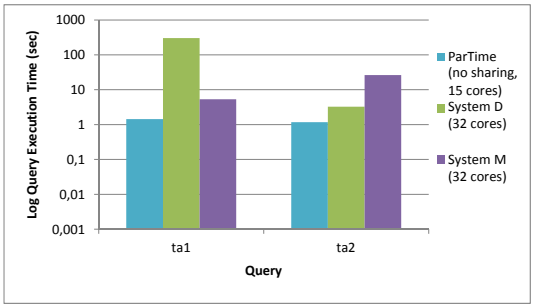
aggregation queries (Fig. 13a) and two non-temporal queries (Fig. 13b) of the Amadeus workload. It becomes clear that Crescando is one order of magnitude better than both System D and System M for the temporal aggregation queries; that is the effect of ParTime. Yet, Crescando is several orders of magnitude worse for the non-temporal queries for which ParTime obviously does not matter. Since the non-temporal queries are much more frequent than the temporal aggregation queries, System M outperforms Crescando in this experiment. Systems D and M outperform Crescando for the non-temporal queries because Systems D and M make use of indexes. In contrast, Crescando had to rely on a table scan for each query in this experiment which obviously resulted in poor single-query performance. Our decision to run Crescando without indexes becomes clearer in the next subsection when we show the effect of shared scans and one of the main results of this paper: ParTime helps Crescando to sustain real world workloads that no other database can handle today.

5.3.2 *Exp 2: The Importance of Sharing*

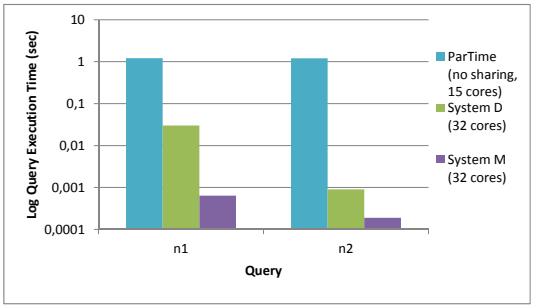
Figure 14 shows the throughput of Crescando (with and without shared scans) for the (read-only) Amadeus workload and the full database (i.e., all 2.4 Billion bookings and all their versions). No other database system that we are aware of can sustain this workload. Even though rare, the temporal aggregation queries consume so many resources in Systems D and M that the throughput virtually drops to zero. To make things worse, these temporal aggregation queries time out in Systems D and M.

Of course, ParTime's ability to process temporal aggregation queries efficiently is critical for the Amadeus workload. What this experiment, however, also demonstrates is another critical strength of ParTime: ParTime co-exists nicely with traditional (non-temporal) query processing techniques as part of a shared scan. Ultimately, that is the key to the success of ParTime for this real-world workload. With every scan through the booking table, Crescando processes a batch of up to 2000 queries. Some of these queries are cheap range queries, some of these queries are more expensive non-temporal queries, and some of them are expensive temporal aggregation queries. Unlike for Systems D and M, however, the expensive temporal aggregation queries do not slow down everybody else with ParTime. Instead, the shared scan seamlessly executes Step 1 of the ParTime algorithm. Step 2 is then carried out on dedicated aggregator nodes, again without hurting the performance of other concurrent queries.

Figure 14 also shows how nicely ParTime and Crescando scale with the number of cores. With 32 cores, the throughput is roughly 15 times as high as with 2 cores. This observation is true for Crescando in both modes. By increasing the number of cores, the size of the partition handled by each storage node gets smaller and the shared scan on each partition correspondingly requires less time. With two cores, for instance, there is only one storage node



(a) Temporal Aggregation Queries



(b) Non-temporal Queries

Figure 13: Resp. Time (logscale): Amadeus, Small DB, 32 Cores (Selected Queries)

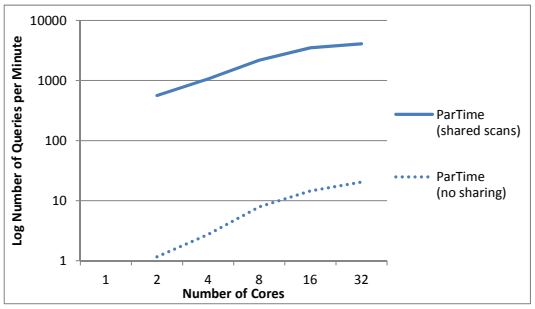


Figure 14: Tput: Amadeus, Large DB, Vary Cores

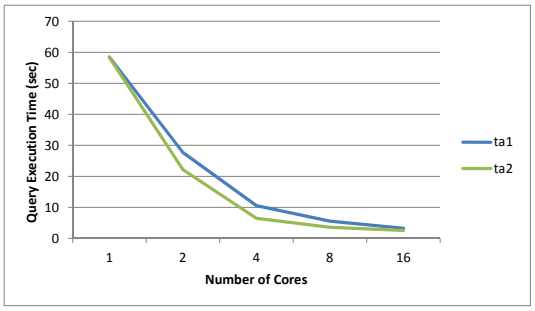


Figure 15: Resp. Time: Amadeus, Large DB, Vary Cores

(and one aggregator node) and that storage node handles the whole 300 GB database. With 30 cores, there are 15 storage nodes and each storage node needs to scan only 20 GB of data for every batch of queries.

To demonstrate the speed-up of ParTime for the temporal aggregation queries of the Amadeus workload, Figure 15 shows the response time of two selected queries with a varying number of cores. These are the same queries shown in Figure 13(a). The figure shows that ParTime has almost linear speed-up for both queries up to sixteen cores. Section 5.4 revisits the effects of parallelism and presents more response time results for a more diverse set of temporal queries of the TPC-BiH benchmark.

5.3.3 Exp 3: Updates

Experiments 1 and 2 studied the Amadeus workload without any updates; i.e., read-only workloads. Figure 16 shows the throughput of Crescando for the full Amadeus workload including 250 updates per second. For this experiment, we used the complete Amadeus database with 2.4 Billion bookings.

Figure 16 shows that Crescando requires at least 18 cores to sustain this workload. With less than 18 cores, Crescando requires

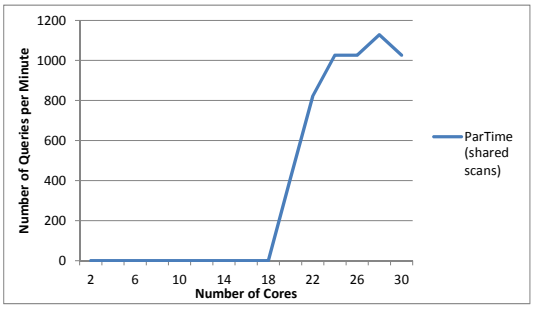


Figure 16: Tput: Amadeus, Large DB, 250 Upd/sec, Vary Cores

all the resources to process the updates and has no more capacity to process any queries. Neither System D nor System M are able to sustain this workload, even with 32 cores and the whole database residing in main memory.

Overall, this experiment confirms the most important findings of Experiments 1 and 2: ParTime helps to handle workloads that traditional database systems cannot sustain. It is the seamless integration of processing temporal aggregation queries with ParTime into regular query processing and shared scans that makes the difference. The presence of updates does not change this basic observation. Update-intensive workloads, however, are a show-stopper for the Timeline Index, the best known approach to process temporal aggregation queries. Specifically studying the performance of ParTime vs. Timeline Index for temporal queries is the subject of the next set of experiments.

5.4 TPC-BiH Benchmark

5.4.1 Exp 4: Overview

Figures 17 and 18 give the results of Systems D, M, and ParTime for the queries of the TPC-BiH benchmark. Furthermore, these figures contain results for the Timeline Index which is directly applicable to the queries of the TPC-BiH benchmark. Again, the Timeline Index is not a good fit for the Amadeus workload because updates are extremely expensive and most queries of the Amadeus workload are non-temporal and the Timeline Index is not applicable to those queries. Furthermore, our prototype implementation of the Timeline Index was not comprehensive enough to execute all the queries and updates of the Amadeus workload.

For ParTime, we studied the response time with 2 cores (1 storage node, 1 aggregator) and with 31 cores (30 storage nodes, 1 aggregator). Systems D and M used all 32 cores in these experiments. We only show results for Systems D and M for the small

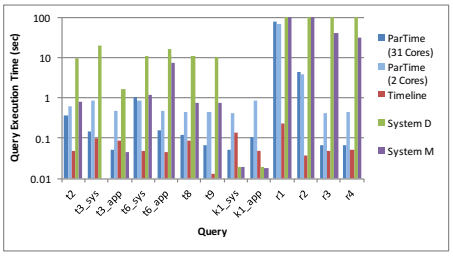


Figure 17: Resp. Time: TPC-BiH, Small DB (SF=1)

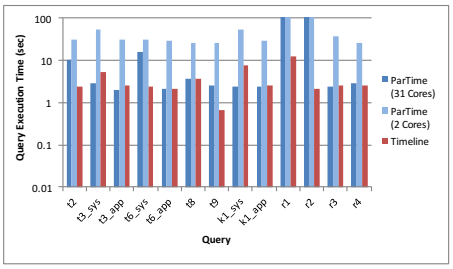


Figure 18: Resp. Time: TPC-BiH, Large DB (SF=100)

TPC-BiH database (Figure 17) because Systems D and M timed out for all queries and a large TPC-BiH database (Figure 18).

Overall, these experiments confirm all expectations. The Timeline Index is the clear winner because it precomputes all temporal information and, thus, makes the execution of temporal queries extremely fast. System D performs worst because it is a disk-based database system and cannot compete with main-memory database systems even if all the data is kept in the main-memory buffers of System D. ParTime with 31 cores outperforms System M with 32 cores because of the effectiveness of the ParTime algorithm. System M with 32 cores, however, outperforms ParTime with only 2 cores because in that configuration ParTime has no advantage from its parallelization as it runs on only one storage node.

The most interesting observation from this experiment can be made by comparing Figures 17 and 18 and focusing on the results for ParTime with 30 cores (the dark blue bars) and the Timeline Index (the red bar). We can see a good example for Amdahls law. For the small database (Figure 17), the difference between ParTime and the Timeline Index is huge: For such a small database, parallelization does not buy us much because there are a number of query processing steps that cannot be parallelized (including Step 2 of the ParTime algorithm in our implementation) and that dominate the response time. For the large TPC-BiH database (Figure 18), the dark blue and red bars are almost the same. Here, the parallelism of ParTime pays back because these queries involve scanning large volumes of data. The amazing result is that parallelization is (almost) as good as pre-computation for such large data sets. We demonstrate this effect in more detail in the next experiment.

5.4.2 Exp 5: Parallelization

Figure 19 shows the response time of the ParTime algorithm for two selected TPC-BiH queries, r_2 and r_4, with a varying number of cores on the large TPC-BiH database (SF=100, 320 GB of raw data). r_2 is a full temporal aggregation over the whole database on business time. r_4 is a windowed temporal aggregation over the whole database on business time.

As expected, r_4 scales almost linearly up to 16 cores (15 storage nodes, 1 aggregator). After that, it flattens out because of Amdahl's

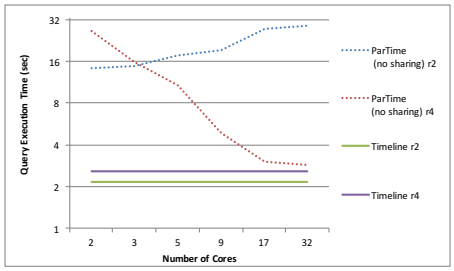


Figure 19: Resp. Time: TPC-BiH, Large DB, Vary Cores

law. For a larger database, this flattening would happen later, with more than 16 cores. This experiment confirms the results of Figure 14 and the scalability of the ParTime algorithm for the temporal queries of the Amadeus workload. The interesting observation is that due to parallelization, ParTime is competitive with the Timeline Index for r_4, even though ParTime does not precompute anything and needs to scan the whole temporal table in order to compute the temporal aggregation.

The second somewhat disappointing result of Figure 19 is that the ParTime algorithm gets worse for r_2 and a growing number of cores. The main reason for that is that the result set of this query is huge: The query result has roughly the same size as the whole temporal table. Partitioning the data does not help for this query because each partition generates a very big delta map independent of the granularity of the partition; again in the order of the size of the whole base table. With two storage nodes (3 cores), for instance, each storage node generates a big delta map and the aggregator in Step 2 merges *two* big delta maps each approximately the size of the base table. With ten storage nodes (11 cores), each storage node again generates a big delta map and the aggregator merges *ten* big delta maps that have the size of the whole base table each. Clearly, this bad behavior is an artifact of Query r_2 of the TPC-BiH benchmark which tests a specific corner case in which the temporal aggregation query virtually does not aggregate any data. Fortunately, such queries are rare in practice.

In summary, these experiments show that the degree of parallelism needs to be optimized and controlled with ParTime. As explained in Section 4, Crescando does not provide this flexibility due to its specific architecture. Other parallel database systems, however, might be able to take advantage of this additional optimization opportunity for ParTime.

5.5 Exp 6: Memory Consumption

One benefit of ParTime is that it does not need any indexes to do efficient query processing. As pointed out earlier, this allows for higher update performance. Another positive effect is that ParTime has a lower memory footprint than all the other approaches that we studied. In most data centers, main-memory footprint translates directly to cost for energy and investments into machines. Table 3 summarizes this result for the TPC-BiH database with scaling factor SF=1. Only System M beats ParTime in terms of memory footprint. The reason is that System M has much better compression than Crescando as it is the more mature main-memory database system. The Timeline Index incurs a storage overhead of almost 30 percent.

5.6 Exp 7: Bulkload Times

We also evaluated the time it takes to load the data into the system. This includes parsing of the data which was initially stored in a simple text file. Even for this relatively small data set of the

System	Size (GB)
Uncompressed Table	2.3
ParTime	2.3
Timeline	3.0
System D	2.5
System M	2.1

Table 3: Memory Consumption, Small DB (SF=1)

System	Time (minutes)
ParTime	2.5
Timeline	4
System D	220
System M	962

Table 4: Bulkload Time, Small DB (SF=1)

TPC-BiH benchmark with SF=1 (i.e., 2.3 GB), Systems D and M take a considerable amount of time. The problem is missing support for bulkloading *temporal* data. These systems are much more efficient in bulkloading non-temporal data. Crescando, in contrast, is quite fast because it bulkloads temporal data in the same way as non-temporal data. The temporal columns are no different than any other column and Crescando creates no data structures that are specific to temporal data. The Timeline Index can be constructed very fast for this small data set; however, for SF=100, it takes more than seven hours to create the Timeline Index and its storage structures whereas it takes only about three hours to bulkload the large TPC-BiH database into Crescando.

6. CONCLUSION

This paper presented ParTime, a parallel temporal aggregation algorithm. ParTime is based on a two step approach. In the first step, the temporal data is scanned, thereby computing the affects of each row in the form of deltas. In the second step, the deltas are merged to compute the result. The first step is embarrassingly parallel. The second step is more difficult to parallelize and we did not parallelize it in our implementation because it is typically cheap. Our experimental results showed that ParTime can achieve linear speed-ups for most queries found in practice. Furthermore, ParTime outperforms any existing algorithm that is not based on pre-computation. With sufficient parallelism, ParTime is even competitive to an approach that is based on a materialized temporal view (i.e., the Timeline Index). In addition to the general ParTime algorithm, this paper presented an important optimization that is applicable to a large class of temporal aggregation queries, so-called windowed temporal aggregation queries.

Arguably, parallelization is the most important feature of ParTime. In addition, ParTime has a number of other advantages. One of the most important advantages is that the first step of the ParTime algorithm integrates nicely into a shared scan. We exploited this feature and integrated ParTime in Crescando, a parallel main-memory database system that relies heavily on shared scans. This way, Crescando with ParTime was able to handle a complex workload from the travel industry with complex temporal aggregate queries and high update rates. To date, no other system is able to sustain this workload. The only way to implement this workload today is to split the workload and to carry out the temporal queries in a separate data warehouse.

There are a number of possible avenues for future work. First, we would like to generalize the ParTime technique and apply it to other temporal operators; e.g., temporal joins. Second, weplan to investigate how ParTime can co-exist with indexes such as the Timeline Index; for instance, would it be possible to partially index historic data that is not updated and to apply ParTime only to fresh and recently appended data in a hybrid way. Third, we would like to develop a cost model in order to compute the optimal degree of parallelism for ParTime.

7. REFERENCES

[1] M. Al-Kateb et al. Temporal query processing in Teradata. In *EDBT*, 2013.

[2] B. Becker et al. An asymptotically optimal multiversion B-Tree. *VLDB J.*, 1996.

[3] M. H. Böhlen, J. Gamper, and C. S. Jensen. Multi-dimensional Aggregation for Temporal Data. In *EDBT*, 2006.

[4] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Temporal statement modifiers. *ACM Trans. Database Syst.*, 25(4), 2000.

[5] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB*, 1999.

[6] G. Candea, N. Polyzotis, and R. Vingralek. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. *PVLDB*, 2009.

[7] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[8] D. Gao et al. Main Memory-Based Algorithms for Efficient Parallel Aggregation for Temporal Databases. *Distributed and Parallel Databases*, 16(2), 2004.

[9] J. A. G. Gendrano et al. Parallel Algorithms for Computing Temporal Aggregates. In *ICDE*, 1999.

[10] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing One Thousand Queries With One Stone. *PVLDB*, 5(6), 2012.

[11] G. Graefe, A. Linville, and L. D. Shapiro. Sort versus hash revisited. *IEEE Trans. Knowl. Data Eng.*, 6(6), 1994.

[12] L. M. Haas et al. Extensible query processing in starburst. In *SIGMOD*, 1989.

[13] M. Kaufmann et al. Timeline Index:A Unified Data Structure for Processing Queries on Temporal Data in SAP HANA. In *SIGMOD*, 2013.

[14] M. Kaufmann et al. TPC-BiH: A Benchmark for Bi-Temporal Databases. In *TPCTC*, 2013.

[15] M. Kaufmann et al. Bi-temporal timeline index: A data structure for processing queries on bi-temporal data. In *ICDE*, 2015.

[16] N. Kline and R. T. Snodgrass. Computing Temporal Aggregates. In *ICDE*, 1995.

[17] L. Lamport. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Trans. Program. Lang. Syst.*, 6(2), Apr. 1984.

[18] D. B. Lomet and B. Salzberg. Access methods for multiversion data. In *SIGMOD*, 1989.

[19] M. A. Nascimento et al. M-IVTT: An Index for Bitemporal Databases. In *DEXA*, 1996.

[20] L. Qiao et al. Main-memory scan sharing for multi-core CPUs. *PVLDB*, 1(1), 2008.

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

[22] B. Salzberg and V. J. Tsotras. Comparison of Access Methods for Time-Evolving Data. *ACM Comput. Surv.*, 31(2), 1999.

[23] C. M. Saracco, M. Nicola, and L. Gandhi. A Matter of Time: Temporal Data Management in DB2 10. Technical report, IBM, 2012.

[24] R. T. Snodgrass et al. TSQL2 Language Specification. *SIGMOD Record*, 23(1), 1994.

[25] P. Unterbrunner et al. Predictable Performance for Unpredictable Workloads. *PVLDB*, 2(1), 2009.

[26] D. Zhang et al. On Computing Temporal Aggregates with Range Predicates. *ACM Trans. Database Syst.*, 33(2), 2008.

[27] M. Zukowski et al. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *VLDB*, 2007.