Adaptable Query Optimization and Evaluation in Temporal Middleware

Giedrius Slivinskas Christian S. Jensen
Department of Computer Science
Aalborg University, Denmark
{giedrius,csj}@cs.auc.dk

Richard T. Snodgrass
Department of Computer Science
University of Arizona, AZ, USA
rts@cs.arizona.edu

ABSTRACT

Time-referenced data are pervasive in most real-world databases. Recent advances in temporal query languages show that such database applications may benefit substantially from built-in temporal support in the DBMS. To achieve this, temporal query optimization and evaluation mechanisms must be provided, either within the DBMS proper or as a source level translation from temporal queries to conventional SQL. This paper proposes a new approach: using a middleware component on top of a conventional DBMS. This component accepts temporal SQL statements and produces a corresponding query plan consisting of algebraic as well as regular SQL parts. The algebraic parts are processed by the middleware, while the SQL parts are processed by the DBMS. The middleware uses performance feedback from the DBMS to adapt its partitioning of subsequent queries into middleware and DBMS parts. The paper describes the architecture and implementation of the temporal middleware component, termed TANGO, which is based on the Volcano extensible query optimizer and the XXL query processing library. Experiments with the system demonstrate the utility of the middleware's internal processing capability and its cost-based mechanism for apportioning the processing between the middleware and the underlying DBMS.

1. INTRODUCTION

In this paper we propose a new approach, that of *temporal middleware*, to evaluating temporal queries that enables significant performance benefits.

Most real-world database applications rely on time-referenced data. For example, time-referenced data is used in financial, medical, and travel applications. Being time-variant is even one of Inmon's defining properties of a data warehouse [10]. Recent advances in temporal query languages [6, 12] show that such applications may benefit substantially from running on a DBMS with built-in temporal support. The potential benefits are several: application code is simplified and more easily maintainable, thereby increasing programmer productivity [21], and more data processing can be moved from applications to the DBMS, potentially leading to better performance.

ACM SIGMOD 2001 May 21-24, Santa Barbara, California USA

In contrast, the built-in temporal support offered by current database products is limited to predefined time-related data types, e.g., the Informix TimeSeries Datablade and the Oracle8 TimeSeries cartridge, and extensibility facilities that enable the user to define new, e.g., temporal, data types [27]. However, temporal support is needed that goes beyond data types. The temporal support should encapsulate temporal operations in query optimization and processing, as well as extend the query language itself.

Developing a DBMS with built-in temporal support from scratch is a daunting task that may, at best, only be feasible by DBMS vendors that already have a code base to modify and have large resources available. This has led to the consideration of a layered, or *stratum*, approach where a layer that implements temporal support is interposed between the user applications and a conventional DBMS [2, 22, 23, 25]. The stratum maps temporal SQL statements to regular SQL statements and passes them to the DBMS, which remains unaltered.

A stratum approach presents difficulties of its own. First, every temporal query must be expressible in the conventional SQL supported by the underlying DBMS, which constrains the temporal constructs that can be supported. Even more problematic is that some temporal constructs, such as temporal aggregation, are quite inefficient when evaluated using SQL, but can be evaluated efficiently with application code that uses a cursor to access the underlying data.

This paper proposes a generalization of the stratum approach, moving some of the query evaluation into the stratum. We term this the "temporal middleware" approach. All previous approaches have consisted entirely of a temporal-SQL-to-SQL translation, effectively a smart macro processor, with all of the work done in the DBMS, and little flexibility in the SQL that is generated. Our middleware approach, in addition to mapping temporal SQL to conventional SQL, performs query optimization and some processing. Moving some of the query processing to the middleware improves query performance because complex operations such as temporal aggregation or temporal duplicate elimination have efficient algorithms in the middleware, but are difficult to process efficiently in conventional DBMSs.

Allowing some of the query processing to occur in the middleware raises the issue of deciding which portion(s) of a query to execute in the underlying DBMS, and which to execute in the middleware itself. Two transfer operations, T^M and T^D, are used to move a relation from the DBMS to the middleware and vice versa. A query plan consists of those portion(s) to be evaluated in the middleware and SQL code for the portion(s) of the query to be processed by the DBMS.

To flexibly divide the processing between the middleware and the DBMS, the middleware includes a query optimizer. Heuristics

are used to reduce the search space, e.g., one heuristic is that the optimizer should consider evaluating in the middleware only those operations that may be processed more efficiently there. Costing is used to determine where to process certain operations, which is not always obvious. For example, whether to process a temporal join in the middleware or in the DBMS depends on the statistics of the argument relations, which are fed into the cost formulas.

This paper makes several contributions. It validates the proposed temporal middleware architecture with an implementation that extends the Volcano query optimizer [8] and the XXL query processing system [1]. The middleware query optimization and processing mechanisms explicitly address duplicates and order in a consistent manner. We provide heuristics, cost formulas, and selectivity estimation methods for temporal operators (using available DBMS statistics); and to divide the processing between the middleware and the DBMS, we use the above-mentioned transfer operators. Performance experiments with the system demonstrate that adding query processing capabilities to the middleware significantly improves the overall query performance. In addition, we show that the cost-based optimization is effective in dividing the processing between the middleware and the DBMS. Thereby, the proposed middleware system captures the functionality of previously proposed stratum approaches and is more flexible.

The presented temporal operators, their algorithms, cost formulas, transformation rules, and statistics-derivation techniques may also be used when implementing a stand-alone temporal DBMS. This makes the presented implementation applicable to both the integrated and the layered architecture of a temporal DBMS, in turn making it relevant for DBMS vendors planning to incorporate temporal features into their products, as well as to third-party developers that want to implement temporal support.

Section 2 presents the architecture of the temporal middleware, and shows how queries flow through the system. The following section presents temporal operators, their implementations in the middleware and the DBMS, and the corresponding cost formulas. For each temporal operation, we propose a method for estimating its selectivity using standard DBMS-maintainable statistics on base relations and attributes. This is needed because standard selectivity estimation does not work well for temporal operations, as we show. Section 4 explains the transformation rules and heuristics used by the middleware optimizer. Performance experiments demonstrate the utility of the shared query processing, as well as of the cost-based optimization.

2. TEMPORAL MIDDLEWARE

We first present the architecture of the temporal middleware, termed TANGO (Temporal Adaptive Next-Generation query Optimizer and processor). Then follows an example of how a query is processed.

2.1 System Architecture

Figure 1 shows TANGO's architecture. The parser translates a temporal-SQL query to an algebra expression, the initial query plan, which is passed on to the optimizer. This plan assigns all processing to the DBMS and specifies that the result is to be transferred to the middleware, by placing a T^M operation at the end.

Optimization occurs in two phases. Initially, a set of candidate algebraic query plans is produced by means of the optimizer's transformation rules and heuristics. Next, the optimizer considers in more detail each of these plans. For each algebraic operation in a plan, it assumes that each of the algorithms available for computing that operation is being used, and it estimates the consequent cost of computing the query. This way, one best *physical* query execution

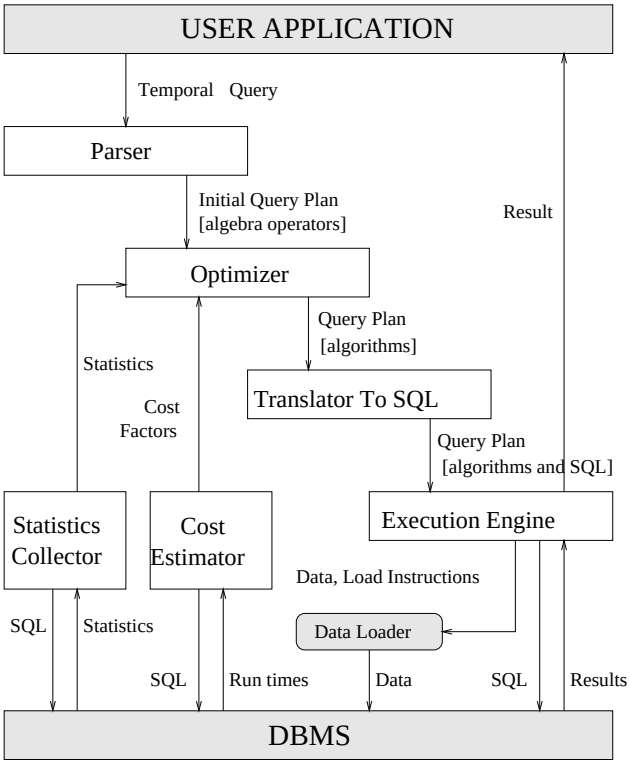


Figure 1: Middleware Architecture

plan, where all operations are specified by algorithms, is found for each original candidate plan. To enable this procedure, the Statistics Collector component obtains statistics on base relations and attributes from the DBMS catalog and provides them to the optimizer. The Cost Estimator component determines cost factors for the cost formulas used by the optimizer. Of the plans generated, the one with the best estimated performance is chosen for execution.

The Translator-To-SQL component translates those parts of the chosen plan that occur in the DBMS into SQL (i.e., parts below T^Ms that either reach the leaf level or T^Ds), and passes the execution-ready plan to the Execution Engine, which executes the plan. The T^M operator results in an SQL SELECT statement being issued, while the T^D operator results in an SQL CREATE TABLE statement being issued, followed by the invocation of a DBMS-specific data loader.

Although both heuristic- and cost-based, this optimizer is lighter weight than a full-blown DBMS optimizer [11], because less information is available to it. While the middleware treats the underlying DBMS as a (quite full featured!) file system, it is not possible for the middleware to accurately estimate the time for the DBMS to deliver a block of tuples from a perhaps involved SQL statement associated with a cursor. This contrasts with a DBMS, which can estimate the time to read a block from disk quite accurately. However, the job of a middleware optimizer is also simpler, in that it does not need to choose among a variety of query plans for the portion of the query to be evaluated by the DBMS. Rather, it just needs to determine *where* the processing of each part of the query should reside. It does so by appropriately inserting transfer operations into query plans.

The optimizer component is an extended version of McKenna and Graefe's Volcano optimizer [8], implemented in C/C++. This optimizer has been enhanced to systematically capture duplicates and order, as well as to support several different kinds of equiv-

alences among relational expressions (e.g., equivalences that consider relations as multisets or lists) [19]. The Execution Engine module is implemented in Java, uses the XXL library of query processing algorithms developed by van den Bercken et al. [1], and accesses the DBMS using a JDBC interface.

Figure 2 describes the main function of the Execution Engine, which receives an execution-ready plan consisting of a sequence of algorithms with their parameters and arguments. For example, an algorithm implementing temporal aggregation takes grouping attributes and aggregate functions as parameters, and a relation as its argument, while an algorithm implementing T^M takes an SQL query as its parameter.

```
ExecuteQuery (Query Plan qp):
    for (i = 0; i < getNumberOfAlgorithms(qp); i++)
        rs[i] = new ResultSet(getAlgorithm(qp, i),
                              getParameters(qp, i),
                              getArg1(qp, i), getArg2(qp, i))
    for (i = 0; i < getNumberOfAlgorithms(qp); i++)
        rs[i].init()
    while(rs[i − 1].hasNext())
        output rs[i − 1].getNext()
```

Figure 2: Pseudo-Code for the Execution Engine

The function first creates result sets for all algorithms in the query plan. Each result set implements iterator interface with $init()$ and $getNext()$ methods, enabling a pipelined query execution. For each result set, its $init()$ method is then invoked. Usually this method just initializes inner structures used by the algorithms, but it does in some cases more: for example, in the case of the algorithm implementing T^D, it fetches all tuples of the argument result set (via its $getNext()$ method) and copies them into the DBMS.

Finally, the $getNext()$ method of the result set for the last algorithm is invoked; in order to collect the result, it invokes the $getNext()$'s of the result sets for the algorithms before it.

2.2 Query Processing Example

An example illustrates how queries are processed. Consider the `POSITION` relation in Figure 3(a), which stores information about the positions of employees. We assume a closed-open representation for time periods and let the time values denote days. For example, Tom occupied position 1 from day 2 through day 19, as indicated by time attributes `T1` and `T2`. We compute the time-variant relation that, for each position tuple, provides the number of employees assigned to that position over time, sorted on the position. This relation is given in Figure 3(b). For example, when Tom occupied position 1 from time 2 to 5, he was the only employee with that position (the count is 1), but from time 5 to 20, Jane also had that position (the count is 2).

Figure 4(a) depicts the initial query plan that the optimizer receives as input. This plan consists solely of algebraic operations and assigns all the processing to the DBMS; and a T^M operation is performed at the end, to deliver the resulting tuples to the middleware, which delivers them to the client. To obtain the desired result, temporal aggregation should be performed first to count the number of employees for each position over time (see its result in Figure 3(c)). This result is then temporally joined with the `POSITION` relation on `PosID` (this join also requires time periods to overlap). The $sort$ operation ensures the desired sorting. Algebraic operators in the initial plan include both regular and temporal operators; temporal operators have their own algorithms for the middleware and are translated into regular SQL if they have to be evaluated in the DBMS.

`POSITION`

PosID	EmpName	T1	T2
1	Tom	2	20
1	Jane	5	25
2	Tom	5	10

(a)

`Query Result`

PosID	EmpName	T1	T2	COUNTofPosID
1	Tom	2	5	1
1	Tom	5	20	2
1	Jane	5	20	2
1	Jane	20	25	1
2	Tom	5	10	1

(b)

`Aggregation Result`

PosID	T1	T2	COUNT
1	2	5	1
1	5	20	2
1	20	25	1
2	5	10	1

(c)

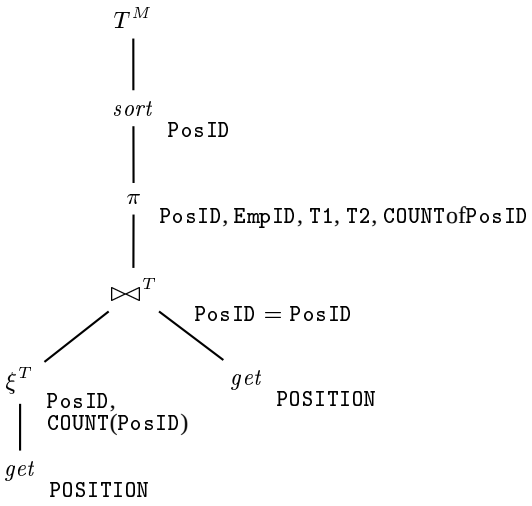
Figure 3: Relation `POSITION` (a), the Query Result (b), and the Aggregation Result (c)

Figure 4(b) shows one of the possible query plans that can be produced by the optimizer. Operations are replaced by actual algorithms for which the optimizer has cost formulas. Superscripts for algorithm names indicate if they have to be evaluated in the DBMS or in the middleware. The given plan states that the `POSITION` relation first should be scanned, with relevant attributes being selected. Then temporal aggregation should be performed in the middleware. Since the temporal aggregation algorithm for the middleware, $TAGGR^M$, requires a sorted argument (see Section 3.4), a $SORT^D$ algorithm is performed before transferring the argument to the middleware. The result of the temporal aggregation is transferred back into the DBMS, which then performs the temporal join (regular join followed by selection and projection). Since the middleware does not know which join algorithm the DBMS will use in each given case, the middleware optimizer uses "generic" cost formula for the DBMS join algorithm (see Section 3.1).

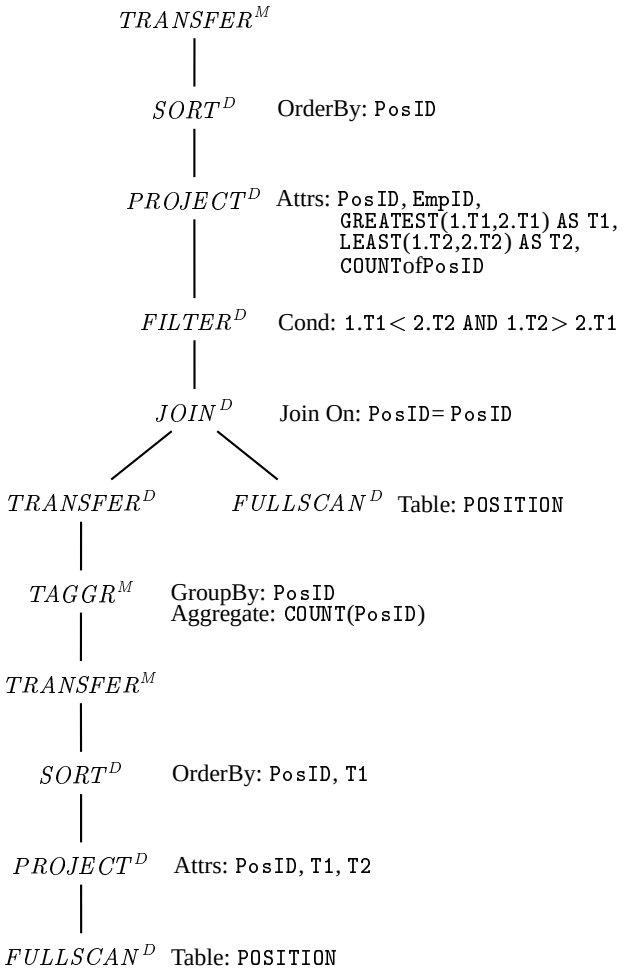
The execution-ready query plan that is passed to the Execution Engine is given in Figure 5. It consists of four algorithms. First, $TRANSFER^M$ issues a SELECT statement to the DBMS in order to obtain the argument for the temporal aggregation. Then, $TAGGR^M$ performs a temporal aggregation, and its result is loaded into the DBMS by $TRANSFER^D$. Finally, $TRANSFER^M$ issues a SELECT statement to the DBMS to obtain the result. In the figure, solid lines represent algorithm arguments, and dashed lines represent algorithm sequence (in this case, the top $TRANSFER^M$ does not take any arguments, but must be preceded by the $TRANSFER^D$ algorithm).

3. STATISTICS AND COST FORMULAS

The availability of statistics on base relations as well as the ability to derive statistics for intermediate relations are important to the query optimizer. The middleware has a separate component that collects statistics from the DBMS, either by querying base relations or by querying the statistics relations that exist in different formats in the various DBMSs. Our middleware uses standard statistics: block counts, numbers of tuples, and average tuple sizes for relations; minimum values, maximum values, numbers of dis-



(a)



(b)

Figure 4: Initial Query Plan (a) and a Possible Selected Query Plan (b)

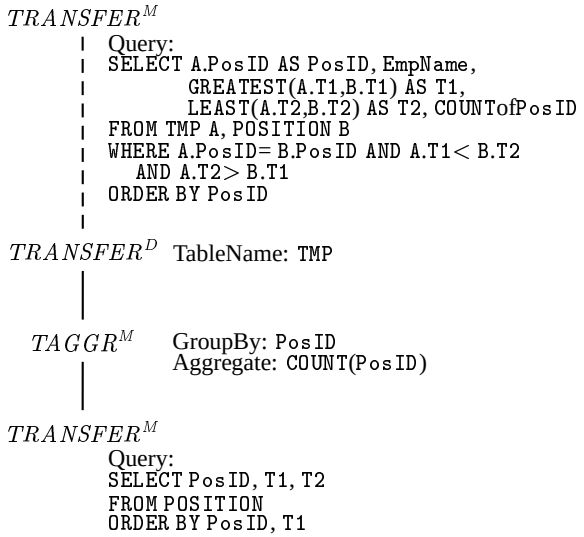


Figure 5: Execution-Ready Query Plan

tinct values, histograms, and index availability for attributes; and clusterings for indexes.

We will shortly describe several algebraic operators and their implementation. For each operator, we will discuss how to derive the cardinality of its result, given the statistics for its argument(s). The main focus is to provide reasonable estimates for the temporal operations, which offers new challenges. For example, standard selectivity estimation does not estimate well the result cardinalities of selections having temporal predicates. Hence, Section 3.3 describes how to obtain more accurate selectivity estimates.

3.1 Cost Formulas

Figure 6 gives cost formulas for the transfer algorithms, the selection and the temporal aggregation algorithms in the middleware, as well as for temporal aggregation in the DBMS. Other algorithms implemented in TANGO include temporal join, join, projection, and sorting; in addition, the middleware optimizer has cost formulas for "generic" implementations of join, Cartesian product, sorting, full table scan, and index scan in the DBMS (see [20] for more details). Additional algorithms may later be added to TANGO, including duplicate elimination, difference, and coalescing. The cost formulas incorporate I/O and CPU costs, and the unit of measure of their return values is microsecond; the formulas are explained when corresponding operators are discussed in Sections 3.2–3.4.

Simplified cost formulas are used in comparison to, e.g., [5], because we generally do not know which algorithms the DBMS might use for queries (hence, we consider only one DBMS implementation of temporal aggregation, even though it may be executed in many different ways).

Conceptually, the cost of an algorithm consists of an initialization cost, the cost of processing the argument tuples, and the cost of forming the output tuples. The initialization costs of all algorithms are set to zero, as are the costs of forming the outputs for sorting, selection, and projection. In addition, we assume a zero cost for selection and projection in the DBMS. Each formula has a number of cost factors p that are used to weigh the statistics, such as *size*(r) (the product of cardinality and average tuple size for relation r); the determination of cost factors is discussed in more detail in [20]. The selection cost formula includes a function that returns a coefficient representing the selection condition.

We now turn to several specific operators and their implementa-

tions, along with the cost formulas and related statistics.

$$
\begin{aligned}
&\mathit{cost}(\mathit{TRANSFER}^M(r)) = p_{tm} \cdot \mathit{size}(r)\\
&\mathit{cost}(\mathit{TRANSFER}^D(r)) = p_{td} \cdot \mathit{size}(r)\\
&\mathit{cost}(\mathit{FILTER}^M{}_P(r)) = p_{sem} \cdot f(P) \cdot \mathit{size}(r)\\
&\mathit{cost}(\mathit{TAGGR}^M{}_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r)) =\\
&\qquad \mathit{cost}(\mathit{SORT}^M{}_{G_1,\ldots,G_n}(r)) +\\
&\qquad p_{taggm1} \cdot \mathit{size}(r) + p_{taggm2} \cdot \mathit{size}(\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r)))\\
&\mathit{cost}(\mathit{TAGGR}^D{}_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r)) = p_{taggd1} \cdot \mathit{size}(r) +\\
&\qquad p_{taggd2} \cdot \mathit{size}(\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r))
\end{aligned}
$$

Figure 6: Cost Formulas

3.2 Transfer Operators

The T^M operator transfers a relation from the DBMS to the middleware. Its implementation, the $\mathit{TRANSFER}^M$ algorithm, is straightforward: it sends an SQL query to the DBMS via the JDBC interface and fetches result tuples.

The performance of this operator depends on the number and size of the tuples transferred. Experiments with Oracle show that the performance is also affected by the row-prefetch setting, which specifies the number of tuples fetched at a time by JDBC to a client-side buffer. We have not included this latter setting in our cost formula because it is DBMS-specific.

The T^D operation transfers data from the middleware to the DBMS. Its algorithm, $\mathit{TRANSFER}^D$, first creates a table in the DBMS and then loads data into it. The data load is specific to the DBMS. For example, the program SQL Loader may be used in Oracle. This program needs a data file with the actual tuples and a control file specifying the structure of the data file. An alternative implementation of the T^D operation could use a sequence of INSERT statements; this solution would be inefficient for large amounts of data.

In Oracle, a number of optimization techniques can be used to speed up the load time of SQL Loader and to minimize the size of the result table. First, direct-path load can be used (which loads data directly into the database as opposed to conventional-path load which uses INSERT statements). Second, since the size of the data to load is known, the initial memory extent allocated for the table can be equal to that size, avoiding the cost of multiple memory allocations. In addition, blocks of the new table do not have to contain any free space because the table will never be updated.

The cost of $\mathit{TRANSFER}^D$ depends on the number and size of the tuples transferred. The name of the table created must be unique, and the table must be dropped at the end of the query.

3.3 Selection

Although DBMSs have efficient selection algorithms, we have also implemented a selection algorithm in the middleware (FILTER^M) because it is sometimes needed. For example, if there is a selection between two temporal algorithms to be performed in the middleware, it would be inefficient to transfer the intermediate result to the DBMS solely for the purpose of selection. The cost of FILTER^M depends on the relation size as well as on the selection predicate.

If the selection predicate is non-temporal, the cardinality of the result relation is estimated using standard methods, as in current DBMSs, by either assuming a uniform distribution between the minimum and maximum values or by using histograms and assuming a uniform distribution within each histogram bucket. (A histogram divides attribute values into buckets; each bucket is assigned to a range of attribute values and stores how many attribute values fall within that range.)

Standard estimation techniques are not directly suitable for temporal predicates. Current DBMSs treat time attributes as any other attributes, storing the same statistics. Straightforward use of these statistics leads to very inaccurate estimates of selections having temporal predicates. However, the statistics available from the DBMS are sufficient to adequately estimate the selectivities of such queries. We elaborate on these points next.

Consider a temporal relation R of 100,000 tuples, where the duration of each time period is 7 days and where time periods are uniformly distributed over the period from January 1, 1995 to January 1, 2000. Consequently, the time period start (`T1`) values are between January 1, 1995 and December 25, 1999, and the time period end (`T2`) values are between January 8, 1995 and January 1, 2000. Both `T1` and `T2` may have 1819 distinct values (the number of days between their minimum and maximum values). Each day then has about 383 tuples with an intersecting time period.

Now consider a query that retrieves all tuples overlapping with the period starting on February 1, 1997 and ending on February 8, 1997 (the predicate would be $\mathit{Overlaps}$(1997-02-01, 1997-02-08); and its SQL condition may be written as `T1` $<$ 1997-02-08 `AND` `T2` $>$ 1997-02-01). Since the distribution of time periods is uniform, histograms are not needed. The number of tuples in the result should be between 383 and 383 $\cdot$ 2 tuples, which is about 0.4%–0.8% of the original relation.

To estimate the selectivity of this query, each predicate is analyzed in turn. The first predicate results in 769/1819 = 42.3% of the original relation, and the second predicate, when applied to the result of the first selection, results in 1064/1819 = 58.5% of the second relation, which is 24.7% of the tuples of the original relation. This is a factor of 40 too high!

As an alternative to this straightforward estimation, we propose to simply take into account that the end time of a period never precedes its start time, which is a simple application of semantic query optimization. The result cardinality for the above-mentioned query can then be estimated by subtracting $\mathit{EndBefore}(\mathtt{A}+1, r)$, the number of tuples ending before or at `A` (here, February 1, 1997), from $\mathit{StartBefore}(\mathtt{B}, r)$, the number of tuples starting before `B` (here, February 8, 1997).

Functions $\mathit{StartBefore}(\mathtt{A}, r)$ and $\mathit{EndBefore}(\mathtt{A}, r)$, where `A` is a time-attribute value in relation r, are defined next. Their definitions depend on whether histograms on `T1` and `T2` are available. For a given histogram H, functions $b1(i, H)$ and $b2(i, H)$ return the start and end values of bucket i; function $bVal(i, H)$ returns the number of attribute values in the i-th bucket, and function $bNo(\mathtt{A}, \mathtt{H})$ return the number of buckets to which attribute value `A` belongs. Functions $minVal(\mathtt{A}, r)$ and $maxVal(\mathtt{A}, r)$ return, respectively, the minimum and maximum values of attribute `A` in relation r, and function $hasHistogram(\mathtt{A}, r)$ returns True if there is a histogram on `A` in relation r.

$$
\mathit{StartBefore}(\mathtt{A}, r) \triangleq
$$

$$
\begin{cases}
\dfrac{\mathtt{A} - minVal(\mathtt{T1}, r)}{maxVal(\mathtt{T1}, r) - minVal(\mathtt{T1}, r)} \cdot cardinality(r) & \\
\qquad\qquad \textbf{if } \neg\, hasHistogram(\mathtt{T1}, r) & \\
\Big(\displaystyle\sum_{i=1,\, i<bNo(\mathtt{A},\mathtt{T1})} bVal(i, \mathtt{T1})\Big) + & \\
\dfrac{\mathtt{A} - b1(bNo(\mathtt{A}, \mathtt{T1}), \mathtt{T1})}{b2(bNo(\mathtt{A}, \mathtt{T1}), \mathtt{T1}) - b1(bNo(\mathtt{A}, \mathtt{T1}), \mathtt{T1})} \cdot bVal(bNo(\mathtt{A}, \mathtt{T1}), \mathtt{T1}) & \\
\qquad\qquad \textbf{otherwise} &
\end{cases}
$$

$$EndBefore(\texttt{A}, r) \triangleq$$

$$\begin{cases} \frac{\texttt{A} - minVal(\texttt{T2}, r)}{maxVal(\texttt{T2}, r) - minVal(\texttt{T2}, r)} \cdot cardinality(r) \\ \qquad \textbf{if } \neg\, hasHistogram(\texttt{T2}, r) \\ (\sum_{i=1,\ i<bNo(\texttt{A},\texttt{T2})} bVal(i, \texttt{T2})) + \\ \frac{\texttt{A} - b1(bNo(\texttt{A}, \texttt{T2}), \texttt{T2})}{b2(bNo(\texttt{A}, \texttt{T2}), \texttt{T2}) - b1(bNo(\texttt{A}, \texttt{T2}), \texttt{T2})} \cdot bVal(bNo(\texttt{A}, \texttt{T2}), \texttt{T2}) \\ \qquad \textbf{otherwise} \end{cases}$$

To compute these functions using histograms, we find the bucket containing attribute value A. Then we sum the number of values in all preceding buckets and add a fraction of the number of values in the bucket containing A, assuming a uniform distribution of the values within the bucket. The formulas are valid for both height-balanced histograms (where each bucket has the same number of values) and width-balanced histograms (where each bucket is of the same length); functions $b1(i, H)$, $b2(i, H)$, $bVal(i, H)$ would return different values for different types of H.

For the given query, $EndBefore(\text{1997-02-02}, R)$ is 769/1819 = 42.3% of the original relation, and $StartBefore(\text{1997-02-08}, R)$ is 755/1819 = 41.5% of the original relation, leading to an estimated size of the result of 0.8% of the original relation, which is close to the actual result.

For a timeslice predicate—such as (T1 $\leq$ A AND T2 $>$ A) which returns all tuples with time periods containing time point A—the result cardinality is $StartBefore(\texttt{A} + 1, r) - EndBefore(\texttt{A} + 1, r)$.

The proposed estimation technique has some resemblance to a previous proposal [18], which uses two temporal histograms: one for the starting points of time periods, and one for "active" time periods (a time period is active during a histogram bucket time period P if it starts before P and overlaps with P). The second histogram is not available from current DBMSs. In contrast, we use only statistics maintained by current DBMSs. The formula for $Overlaps(\texttt{A}, \texttt{B})$ without histograms follows the estimation techniques given in [9].

3.4 Temporal Aggregation

Temporal aggregation (ξ^T) is one of those operators that clearly benefit from running in the middleware versus in the DBMS. We have implemented a middleware implementation, $TAGGR^M$, and a DBMS implementation, $TAGGR^D$, which is a 50-line SQL query (provided in [20]). Below we discuss $TAGGR^M$ as well as how the result cardinality is derived.

For $TAGGR^M$, we require its argument to be sorted on the grouping attribute values and on T1, because if tuples of the same group are scattered throughout the relation, aggregate computation requires scanning of the whole relation for *each* group. Meanwhile, if the argument is ordered on the grouping attributes, only a certain part of the argument relation is needed at a time. The sorting enables reading each tuple only once.

In addition, another copy of the argument is sorted on all grouping attributes and T2. The first sorting is performed by an external algorithm ($SORT^M$ or $SORT^D$), while the second sorting is performed internally by the $TAGGR^M$ algorithm. The algorithm traverses both copies of the argument similarly to sort-merge join and computes the aggregate values group by group. The algorithm's pseudocode is outlined in [20]; it is different from the temporal aggregation algorithms presented in [13], which used aggregation trees in memory or, during computation, maintained lists of constant periods and their running aggregate values.

The cost of temporal aggregation in the middleware depends on the size of the argument and of the result (see Figure 6). For simplicity, the complexity of the actual aggregate functions (such as MIN or AVG) is not included, but experiments show that different such functions do not change the cost significantly. The cost of internal sorting is accounted for.

The upper bound for the cardinality of $\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r)$ is $cardinality(r) \cdot 2 - 1$, and the lower bound (for a non-empty relation) is 1. Knowing the number of distinct values for the grouping and the time attributes allows us to tighten the range between the minimum and maximum.

The minimum cardinality is $\min(distinct(G_1, r), \ldots, distinct(G_n, r), distinct(\texttt{T1}, r) + 1, distinct(\texttt{T2}, r) + 1)$, where $distinct(\texttt{A}, r)$ is the number of distinct values in attribute A in relation r. If there are no grouping attributes, the maximum cardinality is $distinct(\texttt{T1}, r) + distinct(\texttt{T2}, r) + 1$. Otherwise, it is

$$(\frac{cardinality(r)}{\max(distinct(G_1, r), \ldots, distinct(G_n, r))} \cdot 2 - 1) \cdot \max(distinct(G_1, r), \ldots, distinct(G_n, r)),$$

where the fraction represents the average number of tuples for each value of the grouping attribute having the most distinct values, and the factor to the right represents the maximum number of the resulting time periods for each such value. We multiply it by the maximum number of distinct values for the grouping attributes. For experiments, we use 60% of the maximum cardinality if the resulting value is bigger than the minimum cardinality, and the minimum cardinality, otherwise.

4. QUERY OPTIMIZATION HEURISTICS AND EQUIVALENCES

Initial query plans have a single T^M operator at the top, assigning all processing to the DBMS. TANGO's optimizer applies transformation rules to generate candidate query plans. In this section, we outline the transformation rules that drive this process.

Transformation rules derive from equivalences that express that the relations that result from two algebraic expressions are in some sense equal. Specifically, we use two kinds of equivalences, list equivalences and multiset equivalences. Two expressions are *list equivalent* if they evaluate to relations that are equal as lists, and are *multiset equivalent* if they evaluate to relations that are equal as multisets. This latter notion of equal takes into account duplicates, but not order.

List equivalence implies multiset equivalence, and for each transformation rule given below, we only explicitly give the strongest equivalence type that holds. We denote left-to-right transformation rules (also termed *heuristics*) by $\rightarrow_L$ or $\rightarrow_M$ and bidirectional transformation rules by $\equiv_L$ or $\equiv_M$.

These two types of equivalence are essential in a middleware architecture because the location where an operation is processed affects the type of equivalence that holds: while the middleware algorithms are designed to be order preserving, this does not hold for the DBMS algorithms. Therefore, applying a $\rightarrow_L$ rule means that if (1) the relation produced by the left-hand side has some specified order and (2) if it is located in the middleware or if the top operation at the left-hand side is sorting, then the relation produced by the right-hand side will have the same order as the relation produced by the left-hand side. But if either of these two conditions does not hold, only multiset equivalence may be assumed.

4.1 Heuristics

We divide the heuristics into four groups, based on their intended function. We describe two of these groups; the other two—for combining several operations into one and for reducing arguments to expensive operations—are reported in [20].

Heuristic Group 1. Move to the middleware only those operations that may be processed more efficiently there.

An operation is moved to the middleware by introducing the T^M operation below it and the T^D operation above it. Experiments with different DBMSs show that the operations that may benefit from being processed by the special-purpose algorithms in the middleware are temporal aggregation, join, and temporal join. Transformation rules T1–T3 accomplish this move. Note that these rules introduce the sort operator because the algorithms that implement these operations in the middleware require sorted arguments (temporal join and join are implemented as sort-merge joins). In addition, we use rules that enable moving selection, projection, and sorting to the middleware (rules T4–T6); we do not introduce extra T^M and T^D operations in these rules because these operations alone cannot be the reason to partition the processing. Rule T6 has type $\rightarrow_L$ because operation T^M preserves order.

(T1) $\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(r) \rightarrow_M$
$T^D(\xi^T_{G_1,\ldots,G_n,F_1,\ldots,F_m}(T^M(sort_{G_1,\ldots,G_n}(r))))$

(T2) $r_1 \bowtie_{ja_1,ja_2} r_2 \rightarrow_M$
$T^D(T^M(sort_{ja_1}(r_1)) \bowtie_{ja_1,ja_2} T^M(sort_{ja_2}(r_2)))$

(T3) $r_1 \bowtie^T_{ja_1,ja_2} r_2 \rightarrow_M$
$T^D(T^M(sort_{ja_1}(r_1)) \bowtie^T_{ja_1,ja_2} T^M(sort_{ja_2}(r_2)))$

(T4) $T^M(\sigma_P(r)) \rightarrow_M \sigma_P(T^M(r))$

(T5) $T^M(\pi_{f_1,\ldots,f_n}(r)) \rightarrow_M \pi_{f_1,\ldots,f_n}(T^M(r))$

(T6) $T^M(sort_A(r)) \rightarrow_L sort_A(T^M(r))$

Rules T1–T3 are applied only if the top operators of their left-hand sides are assigned to processing in the DBMS. In these and all subsequent rules, r may be a base relation or an operation tree (query expression).

Heuristic Group 2. Eliminate redundant operations.

This group includes rules for removing sequences of T^M and T^D operations (caused be multiple applications of rules T1–T3), and unnecessary projection and sort operations. A sorting operation can be removed if its argument is already ordered as needed, or if only multiset equivalence is required (this may happen, for example, if the relation will be sorted later, or if the end result does not need to be ordered). For each given heuristic, we specify its pre-condition (if any) following the heuristic.

(T7) $T^M(T^D(r)) \rightarrow_M r$

(T8) $T^D(T^M(r)) \rightarrow_M r$

(T9) $\pi_{f_1,\ldots,f_n}(r) \rightarrow_L r$ $\quad \{f_1,\ldots,f_n\} = \Omega_r$

(T10) $sort_A(r) \rightarrow_L r$ $\quad IsPrefixOf(A,\ Order(r))$

(T11) $sort_A(r) \rightarrow_M r$

(T12) $sort_A(sort_B(r)) \rightarrow_L sort_A(r)$ $\quad IsPrefixOf(B, A)$

Rule T9 can be applied for projections on all attributes of the argument relation. We denote the attribute domain of the schema of relation r by Ω_r. Predicate $IsPrefixOf$ takes two lists as argument and returns True is the first is a prefix of the second.

4.2 Equivalences

In addition to the uni-directional heuristics given above, a number of bi-directional equivalences are employed, including moving selections and projections down or up the operation tree and switching the order of Cartesian products. We mark pre-conditions that apply only for the left-to-right and right-to-left transformation by $[\texttt{lr}]$ and $[\texttt{rl}]$, respectively.

(E1) $\pi_{f_1,\ldots,f_n}(\sigma_P(r)) \equiv_L \sigma_P(\pi_{f_1,\ldots,f_n}(r))$
$[\texttt{lr}]\ attr(P) \subseteq attr(f_1,\ldots,f_n)$

(E2) $r_1\ op\ r_2 \equiv_M r_2\ op\ r_1$ $\quad op \in \{\times, \bowtie, \bowtie^T\}$

(E3) $(r_1\ op\ r_2)\ op\ r_3 \equiv_L r_1\ op\ (r_2\ op\ r_3)$ $\quad op \in \{\times, \bowtie, \bowtie^T\}$

(E4) $sort_A(\sigma_P(r)) \equiv_L \sigma_P(sort_A(r))$

(E5) $sort_A(\pi_{f_1,\ldots,f_n}(r)) \equiv_L \pi_{f_1,\ldots,f_n}(sort_A(r))$
$[\texttt{lr}]\ attr(A) \subseteq \Omega_r,\ [\texttt{rl}]\ attr(A) \subseteq attr(f_1,\ldots,f_n)$

Function $attr$ returns the set of attributes present in projection functions or in a selection predicate. Equivalences E4 and E5 are used only when their left-hand side operations are processed in the middleware. Because equivalent query parts assigned to processing in the DBMS are subsequently translated into the same SQL code, it is useful to apply transformation rules to the DBMS parts only when this may help the middleware optimizer to more accurately estimate their costs. Consequently, applicable rules include, e.g., introduction of extra projections or selections. Pushing sorting down or up does not help the optimizer.

5. PERFORMANCE STUDIES

To validate TANGO, we conducted a series of performance experiments. Objectives of the experiments and the data used are described in Section 5.1, and the optimization and processing of four queries is discussed in Section 5.2. Section 5.3 summarizes performance study findings.

5.1 Objectives and Context

We set a number of objectives for performance experiments. First, we wanted to determine if and when it is worth processing fragments of queries in the middleware, and where and when the different operations should be evaluated. In addition, we wanted to evaluate the robustness of the middleware optimizer, i.e., does it return plans that fall within, say, 20% of the best plans. We also attempted to validate the advantages of cost-based optimization, including the proposed selectivity estimation technique for temporal selections. Finally, we sought to determine how significant the overhead of TANGO is.

We performed a sequence of queries, where each query aims to answer a number of the above-mentioned questions. The queries were run on realistic dataset from a university information system [7]. Specifically, two relations were used, namely `EMPLOYEE`, which maintains information about employees, and `POSITION`, part of which was used in Section 2.1 and which provides information on job assignments to employees. The first relation has 49,972 tuples of 31 attributes (about 13.8 megabytes of data) and the second relation has 83,857 tuples of 8 attributes (about 6.7 megabytes of data). We have also used eight other variants of `POSITION` with, respectively, 8,000, 17,000, 27,000, 36,000, 46,000, 55,000, 64,000, and 74,000 tuples from the original relation.

All queries were optimized using the middleware's optimizer and then run via its Execution Engine. All running times in the graphs are given in seconds; for query plans involving middleware algorithms, the middleware optimization time is included. To enable optimization in the middleware, we collected statistics on the (DBMS) relations used via the Statistics Collector module, and we calibrated the cost factors in the cost formulas via the Cost Estimator module; for the latter procedure we used a mechanism similar to that of Du et al. [4] and described it in more detail in [20].

5.2 Queries

We have examined closely the plans for a number of queries to ensure that the optimizer identifies the portions of queries that are appropriate for execution in the DBMS and in the middleware. Here, we consider four such queries in some detail. For each, we show several of the plans that were enumerated by the optimizer, and we measure the evaluation time for these selected plans over a range of data. In most cases, the optimizer does select the best plan among the enumerated ones; we elaborate on how it does so.

The Volcano optimizer [8] is based on a specific notion of equivalence class. Each equivalence class represents equivalent subexpressions of a query, by storing a list of elements, where each element is an operator with pointers to its arguments (which are also equivalence classes). The number of equivalence classes and elements for a query directly correspond to the complexity of the query; we give these measures for each query.

Query 1. "For each position in `POSITION`*, get the number of employees occupying that position at each point of time. Sort the result by the position number."*

This temporal aggregation query was used as subquery in the example query in Section 2.2. Figure 7 shows three of the query evaluation plans for this query. The first sorts the base relation in the DBMS on the grouping attribute and the starting time, then performs the temporal aggregation in the middleware. Since $TAGGR^M$ preserves order on the grouping attributes, additional sorting is not needed at the end. The second plan is similar, but performs sorting in the middleware. The third performs everything in the DBMS. Due to space constraints, we omit the complete SQL query here.

$TAGGR^M$ GroupBy: `PosID`
Aggregate: `COUNT(PosID)`
|
$TRANSFER^M$
Query:
`SELECT PosID, T1, T2`
`FROM POSITION`
`ORDER BY PosID, T1`

Plan 1

$TAGGR^M$ GroupBy: `PosID`
Aggregate: `COUNT(PosID)`
|
$SORT^M$ OrderBy: `PosID, T1`
|
$TRANSFER^M$
Query:
`SELECT PosID, T1, T2`
`FROM POSITION`

Plan 2

$TRANSFER^M$
Query:
[temporal aggregation in SQL]

Plan 3

Figure 7: Plans for Query 1

We compare the three plans for varying sizes of the argument relation. For all queries, the optimizer selects the first plan. The optimizer generated 12 equivalence classes with 29 class elements.

The running times of all plans are shown in Figure 8, where it can be seen that the first two significantly outperform the third. This is because temporal aggregation in the DBMS is very slow. While not reported here, experiments with similar queries, where the the grouping attribute(s) and relation size are also varied, show similar results.

This experiment shows that processing in the middleware can be

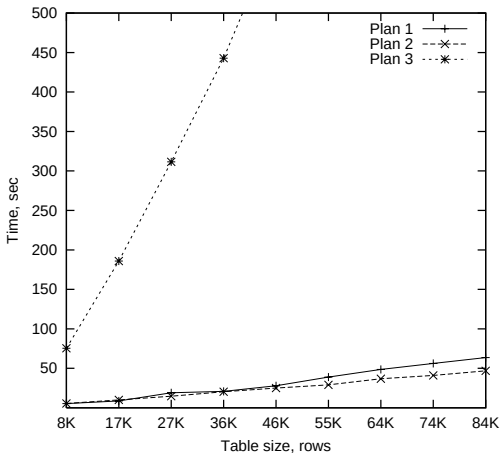


Figure 8: Results of Query 1

up to ten times faster, if a query involves temporal aggregation. Temporal aggregation in the DBMS can compete with temporal aggregation in the middleware only when a very small number of records (a few hundreds) have to be aggregated (see Query 2).

Query 2. "Produce a time-varying relation that provides, for each `POSITION` *tuple with pay rate greater than $10, the count of employees that were assigned to the position. Consider the time period between January 1, 1983 and January 1, 1984 and sort the result by position number."*

This query corresponds to the query presented in Section 2.2, but we introduce the time period and the $10 pay rate condition.

Six plans were used, four of which are given in Figure 9. The first plan performs temporal aggregation in the middleware and the rest in DBMS. The next three plans also assign temporal join to the middleware (Plan 2); temporal join and sorting to the middleware (Plan 3); and temporal join, sorting, and selection to the middleware (Plan 4). The fifth plan (not shown) is the same as the first, but no selection is performed on the argument to the temporal aggregation (this selection is not needed for correctness, but it reduces the argument size). The sixth plan (not shown) performs everything in the DBMS.

We ran all six plans a number of times, each time increasing the end time of the time period given in the query by one year, thus relaxing the predicate. Since most of the `POSITION` data is concentrated after 1992, the running times are similar for the queries with the time period ending before 1992 (see Figure 10), but they increase rapidly after that time (see Figure 10(b)). In Figure 10(a), we also observe that Plans 4 and 5 perform poorly; this is because of the high cost of the $TRANSFER^M$ operation, which takes the whole base relation as its argument (without applying selection first). Plan 6, which performs temporal aggregation in the DBMS, is competitive because the selection predicates are very selective.

For larger time periods (Figure 10(b)), the performances of the plans vary more. Plans 4 and 5 are slow due to the expensive $TRANSFER^M$ operations, and Plan 6 also deteriorates rapidly when the argument to the temporal aggregation increases. Plan 1 deteriorates faster than Plans 2 and 3 because it includes the $TRANSFER^D$ algorithm, which becomes significantly slower when its argument's size increases (due to the increase of the selection time period).

We optimized this query running the middleware optimizer with and without histograms on the time attributes. When used without histograms, the optimizer returned the second plan for the six

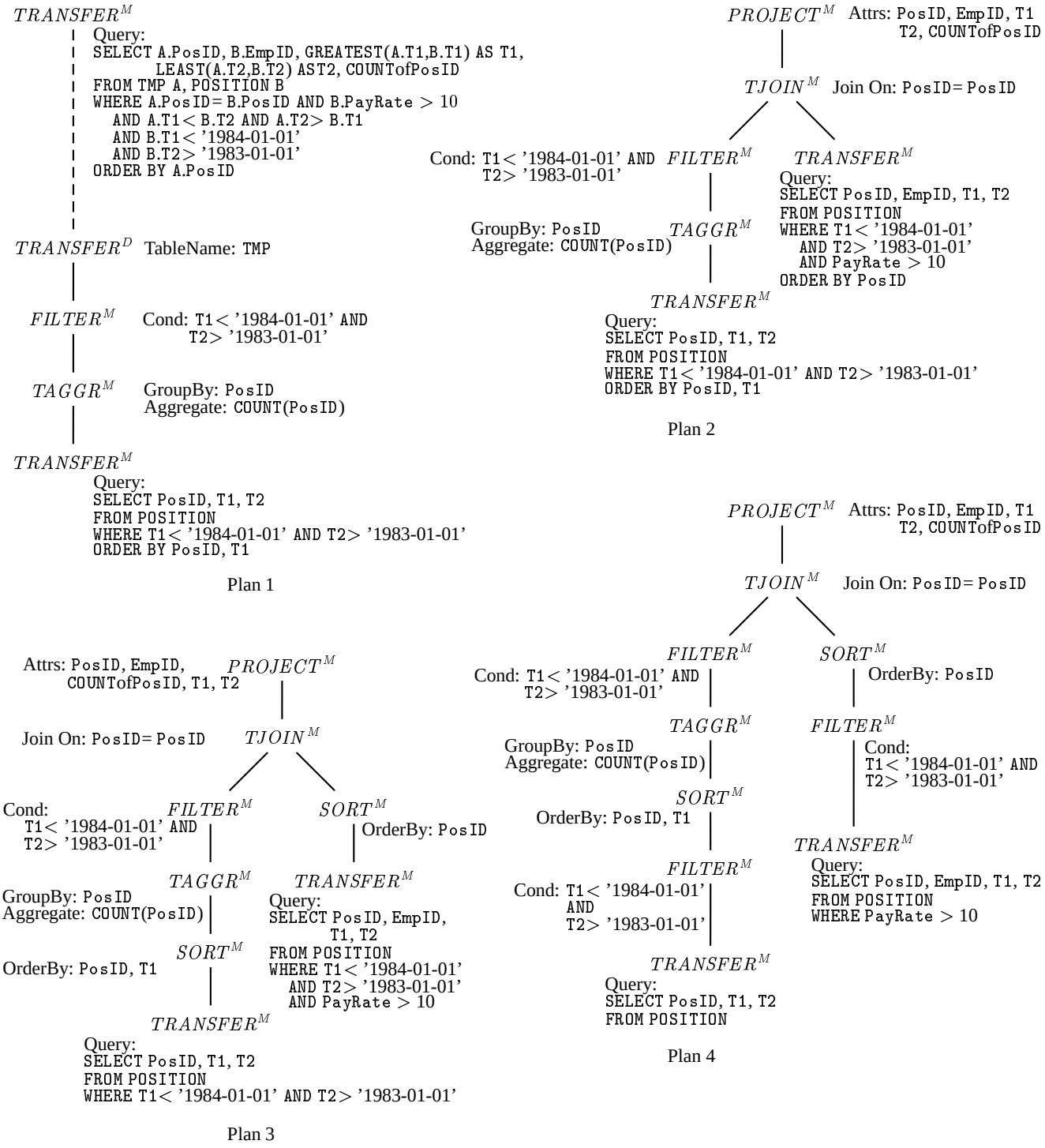


Figure 9: Plans for Query 2

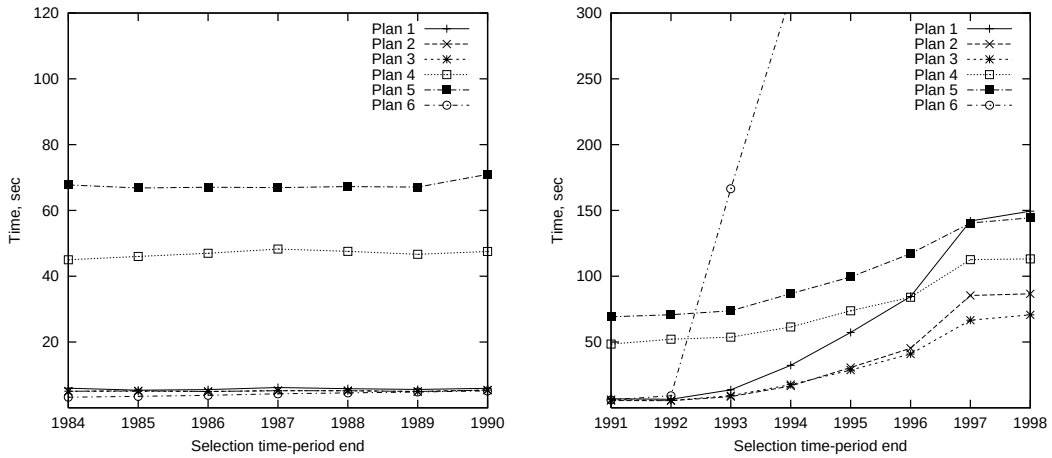


Figure 10: Results of Query 2 when the Selection Time-Period End Value was 1990 or Smaller (a) and 1991 or Bigger (b)

queries with the time-period end varying from January 1, 1984 to January 1, 1989, and the first plan for all other queries. When used with histograms, the optimizer always returned the second plan (which is better than the first plan, as is clear from Figure 10(b)), because it could more accurately estimate the result size of the temporal selection. The optimizer generated 142 classes with 452 elements in total.

This query shows that temporal join can be as much as two times faster in the middleware if at least one of its arguments resides there (as does, in this case, the result of temporal aggregation). Other experiments [20] show that the same holds for regular join. In addition, this experiment confirms that the cost-based selectivity estimation helps the middleware optimizer return better plans.

Query 3. "For each position in `POSITION` *starting before January 1, 1990, show all pairs of employees that occupied that position during the same time. Sort the result by the position number."*

This query is a temporal self-join. We tested two plans: the first performs everything in the DBMS, while the second performs temporal join in the middleware. In the experiment, we have varied the condition constraining the time-period start. The running times are shown in Figure 11(a).

When the maximum allowed time for the time-period start increases, Plan 2 performs better than Plan 1 because the result is bigger than the arguments, leading to high costs of sorting within the DBMS and transfer of the result in Plan 1. The difference in performance becomes obvious when the maximum time-period start reaches year 1996, since about 65% of the `POSITION` tuples have time-periods starting at 1995 or later.

The middleware optimizer returned Plan 1 for the first six queries and Plan 2 for the last three. The errors for the middle three queries—where Plan 2 is already better than Plan 1—occur because the selectivity estimation for join and temporal join assumes uniform distribution of the join-attribute values (`PosID`), which is not the case for the data used. The optimizer generated 104 equivalence classes with 301 element.

This query illustrates that allocating processing (in this case, of temporal join) to the middleware can be advantageous if the result size is bigger than the argument sizes. It also demonstrates that the cost-based optimization leads to selecting a better plan for the last three queries.

Query 4. "For each position, list the employee name and address."

This query is a regular join of the `POSITION` and `EMPLOYEE` relations. We tested three plans: the first plan performs sorting and join in the middleware, the second plan performs a nested-loop join in the DBMS, and the third plan performs a sort-merge join in the DBMS (the DBMS join methods were set explicitly using Oracle hints). We executed the plans while varying the size of the `POSITION` relation. The results in Figure 11(b) show that Plan 2 yields the best performance while the other two plans are competitive. The middleware optimizer suggested to perform the join in the DBMS (plans 2 and 3; since the optimizer does not consider different DBMS join algorithms, both plans were considered as one). It generated 13 equivalence classes with 30 elements in total.

This experiment shows that the DBMS is faster when performing queries involving regular operations. The fact that similar algorithms are competitive in the DBMS and middleware (both plans 1 and 3 include sort-merge joins) indicates that the run-time overhead introduced by TANGO is insignificant.

5.3 Summary of Performance Study Findings

The performance experiments demonstrate that the middleware can be very effective when processing queries involving temporal aggregation. Temporal join is faster in the middleware if at least one its arguments already resides there (Query 2), or if its result size is bigger than its argument sizes (Query 3); other experiments [20] show that there are cases when temporal join is more efficient in the DBMS.

In addition, we showed that the cost-based optimization with its simplified cost formulas is effective in dividing the processing between the middleware and the DBMS. The proposed selectivity estimation techniques for temporal selection was shown to more accurately estimate sizes of intermediate relations, which generally results in better plans being selected. Plans allocating all evaluation for the DBMS (including temporal aggregation) perform well for highly selective queries, but deteriorate rapidly as selection predicates are relaxed (Figure 10(b)).

For the tested queries, the middleware optimization overhead was very small. We have not implemented the parser and Translator-To-SQL middleware modules, but we do not expect them to significantly slow down the processing. They will use standard language technology and are independent of the database size. It should be noted, though, that we have not tested queries involving many joins;

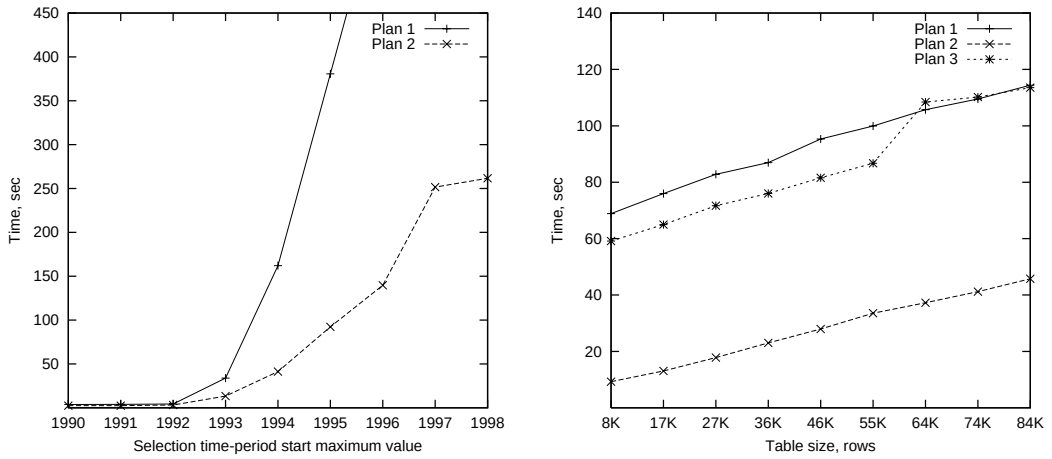


Figure 11: Results of Query 3 (a) and Query 4 (b)

for such queries, it is likely that join-order heuristics should be introduced instead of the join equivalences used (Section 4.2).

6. RELATED WORK

The general notion of software external to the DBMS participating in query processing is classic. Much work has been done on *heterogeneous databases* (e.g., [15]), in which data resident in multiple, not necessarily consistent, databases is queried and updated [28]. There has also been a great deal of work in the related area of *mediators* [26] and, more generally, on *integration architectures*. Roughly, a mediator offers a consistent data model and accessing mechanism to disparate data sources, which may not be traditional databases. At an abstract level, this approach shares much with the notion of temporal middleware: the underlying databases cannot be changed, the data models and query languages exposed to the users may differ from those supported by the underlying databases, the exported schema may be different from the local schema(s), significant query processing occurs outside the underlying DBMS, and a layer (also termed a *wrapper* [16]) is often interposed that changes the data model or allows new query facilities to access the data. However, there are also differences. A heterogeneous database by definition involves several underlying databases, whereas the temporal middleware is connected to but one underlying database and does not need to address issues of data fusion and schematic discrepancies, or of access to semi-structured data.

Several papers discuss layered architectures for a temporal DBMS, e.g., [22], and several prototype temporal DBMSs have been implemented, e.g., [3]. That work is based on a pure translation of temporal query language statements to SQL and does not provide systematic solutions on how to divide the processing of temporal queries between the layer and the underlying DBMS. Vassilakis et al. [25] discuss techniques for adding transaction and concurrency-control support to a layered temporal DBMS; they propose a layer that accepts queries written in VT-SQL, identifies the regular-SQL parts of such queries, and sends these parts to the DBMS for processing. Their layer is able to evaluate temporal constructs at the end of a query, if needed. The TANGO system presented here is more flexible in apportioning the processing.

In this paper, we extend our previous foundation for temporal query optimization [19], which included a temporal algebra that captured duplicates and order, defined temporal operations, and offered a comprehensive set of transformation rules. However, that foundation did not cover optimization heuristics, the implementation of temporal operations, or their cost formulas, which are foci of the present paper. Other work on temporal query optimization [9, 14] primarily considers the processing of joins and semijoins. Perhaps most prominently, Gunadhi and Segev [9] define several kinds of temporal joins and discuss their optimization. They do not delve into the general query optimization considered here. Vassilakis [24] presents an optimization scheme for sequences of coalescing and temporal selection; when introducing coalescing to our framework, this scheme can be adopted in the form of transformation rules. Related work in selectivity estimation for temporal operators includes [9, 17, 18]; we use some of their techniques for estimating the selectivity of temporal predicates (when histograms are not available), and we also show how selectivity can be estimated by using solely statistics available from conventional DBMSs.

Several papers have considered cost estimation in heterogeneous systems. Du et al. [4] propose a cost model for different selections and joins. Cost factors used in the formulas are deduced in a *calibration* phase, where a number of sample queries are run. We use a similar approach, but we assume that we do not know the specific algorithms used by the DBMS.

TANGO is implemented using the Volcano extensible query optimizer [8] and the XXL library of query processing algorithms [1]. Volcano was significantly extended to include new operators, algorithms, and transformation rules, as well as different types of equivalences (Section 4). Available XXL algorithms for regular operators, as well as our own algorithms for temporal operators, were used in TANGO's Execution Engine.

7. CONCLUSIONS

This paper offers a temporal middleware approach to building temporal query language support on top of conventional DBMSs. Unlike previous approaches, this middleware performs some query optimization, thus dividing the query processing between itself and the DBMS, and then coordinates and takes part in the query evaluation. Performance experiments show that performing some query processing in the middleware in some cases improves query performance up to an order of magnitude over performing it all in the DBMS. This is because complex operations, such as temporal aggregation, which DBMSs have difficulty in processing efficiently, have efficient implementations in the middleware.

The paper's contributions are several. It proposes an architecture

for a temporal middleware with query optimization and processing capabilities. The middleware query optimization and processing explicitly and consistently address duplicates and order. Heuristics, cost formulas, and selectivity estimation techniques for temporal operators (using available DBMS statistics) are provided. The temporal middleware architecture is validated by an implementation that extends the Volcano optimizer and the XXL query processing system. Performance experiments validate the utility of the shared processing of queries, as well as of the cost-based optimization.

The result is a middleware-based system, TANGO, which captures the functionality of previously proposed temporal stratum approaches, and which is more flexible.

The proposed transformation rules and selectivity estimation techniques may also be used in an integrated DBMS, e.g., when adding temporal functionality to object-relational DBMSs via user-defined functions. For this to work, the user-defined functions must manipulate relations and must be able to specify the cost functions and transformation rules relevant to them to the optimizer.

Several directions for future work exist. The current middleware algorithms should be enhanced to support very large relations. In addition, new operators may be added to TANGO. To add an operator, one needs to specify relevant transformation rules, formulas for derivation of statistics, and algorithm(s) implementing the operator. If the operator is to be implemented in the middleware, its algorithm has to be added to the Execution Engine.

DBMS query processing statistics, such as the running times of query parts, may be used to update the cost factors used in the middleware's cost formulas. It is an interesting challenge to be able to divide the running time between the $TRANSFER^M$ algorithm and, possibly, several DBMS algorithms. A number of other refinements are also possible. For example, if a query is to access the same DBMS relation twice (even if the projected attributes are different), it would be beneficial to issue only one T^M operation.

Acknowledgments

This research was supported in part by the Danish Technical Research Council through grant 9700780, by the U.S. National Science Foundation through grant IIS-9817798, and by a grant from the Nykredit Corporation.

8. REFERENCES

[1] J. Van der Bercken, J. P. Dittrich, and B. Seeger. javax.XXL: A Prototype for a Library of Query Processing Algorithms. In *Proceedings of ACM SIGMOD*, Dallas, TX, p. 588 (2000).

[2] M. H. Böhlen. Temporal Database System Implementations. *ACM SIGMOD Record*, 24(4): 53–60 (1995).

[3] M. H. Böhlen. The Tiger Temporal Database System. URL: <www.cs.auc.dk/~tigeradm/> (current as of February 23, 2001).

[4] W. Du, R. Krishnamurthy, and M.-C. Shan. Query Optimization in a Heterogeneous DBMS. In *Proceedings of VLDB*, Vancouver, Canada, pp. 277–291 (1992).

[5] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Third Edition. Addison-Wesley (2000).

[6] O. Etzion, S. Jajodia, and S. Sripada (eds.). *Temporal Databases: Research and Practice*. LNCS 1399. Springer-Verlag (1998).

[7] J. A. G. Gendrano, R. Shah, R. T. Snodgrass, and J. Yang. *University Information System (UIS) Dataset.* TIMECENTER CD-1, September, 1998.

[8] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of IEEE ICDE*, Vienna, Austria, pp. 209–218 (1993).

[9] H. Gunadhi and A. Segev. A Framework for Query Optimization in Temporal Databases. In *Proceedings of SSDBM*, Charlotte, NC, pp. 131–147 (1990).

[10] W. H. Inmon. *Building the Data Warehouse*. Second Edition. John Wiley and Sons (1996).

[11] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2): 111–152 (1984).

[12] C. S. Jensen and R. T. Snodgrass. Temporal Data Management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–45, 1999.

[13] N. Kline and R. T. Snodgrass. Computing Temporal Aggregates. In *Proceedings of IEEE ICDE*, Taipei, Taiwan, pp. 222–231 (1995).

[14] T. Y. C. Leung and R. R. Muntz. Stream Processing: Temporal Query Processing and Optimization. In *Temporal Databases: Theory, Design, and Implementation*, A. U. Tansel et al. (eds.), Benjamin/Cummings, pp. 329–355 (1993).

[15] T. M. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Second Edition. Prentice Hall (1999).

[16] M. T. Roth and P. M. Schwarz. Don't Scrap it, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *Proceedings of VLDB*, Athens, Greece, pp. 266–275 (1997).

[17] A. Segev, G. Himawan, R. Chandra, and J. Shanthikumar. Selectivity Estimation of Temporal Data Manipulations. *Information Sciences*, 74(1-2): 111–149 (1993).

[18] I. Sitzmann and P. J. Stuckey. Improving Temporal Joins Using Histograms. In *Proceedings of DEXA*, London/Greenwich, UK, pp. 488–498 (2000).

[19] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Query Plans for Conventional and Temporal Queries Involving Duplicates and Ordering. In *Proceedings of IEEE ICDE*, San Diego, CA, pp. 547–558 (2000).

[20] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Adaptable Query Optimization and Evaluation in Temporal Middleware. TIMECENTER Technical Report TR-56, URL: <www.cs.auc.dk/TimeCenter/> (2001).

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

[22] K. Torp, C. S. Jensen, and R. T. Snodgrass. Stratum Approaches to Temporal DBMS Implementation. In *Proceedings of IDEAS*, Cardiff, Wales, pp. 4–13 (1998).

[23] K. Torp, C. S. Jensen, and R. T. Snodgrass. Effective Timestamping in Databases. *The VLDB Journal*, 8(3-4): 267–288 (2000).

[24] C. Vassilakis. An Optimisation Scheme for Coalesce/Valid Time Selection Operator Sequences. *SIGMOD Record*, 29(1): 38–43 (2000).

[25] C. Vassilakis, N. A. Lorentzos, and P. Georgiadis. Implementation of Transaction and Concurrency Control Support in a Temporal DBMS. *Information Systems*, 23(5): 335–350 (1998).

[26] G. Wiederhold. Mediation in Information Systems. *ACM Computing Surveys*, 27(2): 265–267 (1995).

[27] J. Yang, H. C. Ying, and J. Widom. TIP: A Temporal Extension to Informix. In *Proceedings of ACM SIGMOD*, Dallas, TX, p. 596 (2000).

[28] C. Yu and W. Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann (1998).