Design and Implementation of a Temporal Extension of SQL

Cindy X. Chen
Department of Computer Science
University of Massachusetts at Lowell
Lowell, MA 01854, U.S.A.
cchen@cs.uml.edu

Jiejun Kong
Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90095, U.S.A.
jkong@cs.ucla.edu

Carlo Zaniolo
Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90095, U.S.A.
zaniolo@cs.ucla.edu

Abstract

We present a valid-time extension of SQL and investigate its efficient implementation on an Object-Relational database system. We propose an approach where temporal queries are expressed using a point-based time model, which only requires minimal extensions to SQL:1999. Our prototype system called TENORS (for Temporal ENhanced Object-Relational System) maps the external point-based temporal queries and data model into equivalent internal representations based on time intervals. We describe the mapping of queries from external views to internal relations, and the temporal clustering and indexing methods used to support these queries on DB2.

1. Introduction

Temporal databases have been the focus of much research work and many solutions have been proposed [7]. A critical factor is the degree of extensibility offered by the database system. In the past, relational DBMSs provided no extensibility, and this rigidity means that a major extension of SQL and a new DBMS architecture are needed to add powerful temporal capabilities to DBMSs—as demonstrated by TSQL2 [3].

Recently, Object-Relational (O-R) systems have started offering a limited degree of extensibility which we use to propose an approach where complex valid-time queries are (i) expressed in SQL:1999 using only temporal aggregates and functions, and (ii) supported with reasonable ease and efficiency on current DBMSs as demonstrated by our prototype called TENORS.

The cornerstone of our approach is the use of a temporal language SQL^T [1]. By relying on a point-based temporal model and temporal aggregates, SQL^T minimizes the extensions required from SQL:1999. A second important component of TENORS is the use of a storage schema that supports temporal clustering and indexing using the concept of page $usefulness$. This scheme builds on standard B$^+$-trees, and thus requires no change in the storage modules used by existing DBMSs.

To achieve a simple and yet efficient implementation, we take full advantage of the advanced features provided by O-R systems, particularly table expressions and user-defined functions. Therefore, TENORS achieves good performance by implementing external queries expressed in SQL^T as internal queries that exploit the temporal clustering and indexes supported in the storage scheme.

In summary, we pursue an *evolutionary* and *minimalist's* approach that supports powerful temporal queries in SQL:1999. Our approach also achieves ease of use, and efficient execution on current O-R systems. This is an opportunistic approach that contrasts with the high-road approach taken by TSQL2, where several new temporal constructs were introduced without any attempt to rely on the extensible features of O-R systems.

2. Temporal Model and Query Language

The SQL^T query language [1] supported by TENORS uses a *point-based, explicit* temporal model [6]. In a point-based time model, a database is viewed as a sequence of snapshots from users' standpoint. A more condensed representation of the set of time instants when a tuple is valid, is provided by the interval-based model [2]. However, interval-based time models rely on coalescing—a complex operation that greatly complicates queries. The point-based approach eliminates the need for coalescing after projection. An alternative approach to overcome the coalescing problems is to use an *implicit* temporal model—i.e., one where there is no explicit attribute to represent the time of validity of tuples in temporal relations [3].

We now introduce SQL^T [1] by examples. Consider the following relations and temporal join:

Employee(Name, Title, Title_Level, Misc, VTime)
Position(Title, Title_Level, Salary, VTime)

Example 1 *Salary history for directors in the 1990s.*

SELECT **L.Name, R.Salary, R.VTime**

```
FROM Employee AS L, Position AS R
WHERE L.Title = "Director" AND L.Title = R.Title
   AND L.VTime = R.VTime
   AND L.VTime >= "1/1/1990" AND L.VTime < "1/1/2000"
```

Temporal joins involve equality of points in time. In SQL^T, the notion of "*same time*" is naturally captured by the equality L.VTime = R.VTime.

3. Internal Model

TENORS uses a Temporal Internal Model (TIM) that (i) efficiently supports snapshot queries, intersection queries, and temporal join, and (ii) simplifies the mapping to current O-R systems for both queries and indexing. Figure 1 shows the overall architecture of TENORS.

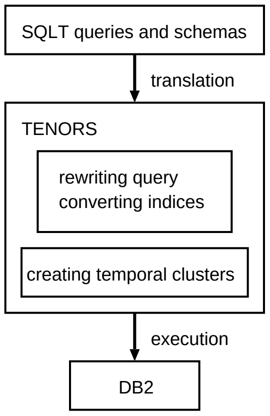

Figure 1. System architecture

Users issue temporal queries in SQL^T, while data is stored in DB2 with additional columns dedicated to temporal elements. TENORS acts as a mediator between the conceptual view of queries written according to a point-based time model and the physical storage of data based on an interval-based time model. SQL^T queries are translated by TENORS into queries that can be executed in DB2. Besides query rewriting, indices defined in SQL^T also get converted into their internal counterparts using techniques introduced by TENORS.

In the internal model of TENORS—TIM, every tuple stores a period of validity for the non-temporal information using two additional attributes, VTstart and VTend. These two attributes represent the start and end time instants of the set of VTime chronons that are contained in the period [VTstart, VTend]. A temporal relation is segmented as follows. Initially, all tuples are valid; thus the initial *usefulness factor* is 100%. As time goes by, some tuples become invalid (i.e., past). So, the usefulness factor of a segment at time t is defined as:

$$U(t) = \frac{V(t)}{T(t)}$$

where $V(t)$ is the count of all tuples still valid (i.e., current or future) at t, and $T(t)$ is the count of all tuples in the segment. TENORS let users specify a minimum usefulness factor U_{min}. When $U(t)$ falls below that threshold (i.e., $U < U_{min}$), the segment is split upon database modification.

Therefore, whenever a temporal segment is instantiated, all tuples are valid and U is 100%. U decreases until it falls below U_{min}, then a *null change* creates a new segment. The history of a relation is partitioned into k consecutive temporal segments, $S_1, S_2, \ldots, S_k$, and TIM uses an additional column $Tseg$ to record the segment number.

This usefulness-based scheme naturally clusters tuples according to their VTstart values, thus confers vast opportunities to speed up snapshot queries and intersection queries. The price is that some tuples are duplicated. Fortunately, the trade-off between storage redundancy and retrieval speedup is user-controllable:

- If $U_{min} = 0$, then null changes never happen and a temporal relation is left unsegmented. Let us denote the size of this unsegmented relation as N_0.
- If $0 < U_{min} < 1$, for statistical reasons, we may ignore the exception that the usefulness factor in some segments may be somewhere between U_{min} and 100%. Assume each segment's usefulness factor is U_{min}. If there are n_i tuples belonging to a temporal segment S_i, then the number of invalid tuples in S_i is $(1 - U_{min}) \times n_i$. Also, let $N_{U_{min}}$ denotes the size of a segmented relation. In the worst case scenario where all the tuples in the original relation become invalid, then we have $N_0 = (1 - U_{min}) \times N_{U_{min}}$ and $N_{U_{min}} = \frac{N_0}{1-U_{min}}$

For example, when $U_{min} = 0.6$ or $U_{min} = 0.5$, the size of the segmented relation obtained is at most 2.5 or 2 times the size of its unsegmented dual, respectively.

The temporal segmentation just described is incorporated in TIM's temporal indexing as follows:

- The VTime column in the external relation is replaced by the three columns: Tseg, VTstart, VTend in TIM where Tseg is the segment number, and VTstart, VTend denote the start and end of the period.

- TENORS also allows the user or database administrator to declare primary (a.k.a., clustered) and secondary (i.e., non-clustered) indices on the external relation—on any combination of attributes *except* VTime. These indices are then automatically translated into indices in TIM as follows:

  1. If K is the primary index on the external relation, then (Tseg,K,VTstart) will be the primary index on the TIM relation.

2. If K is a secondary index on the external relation, then (K,Tseg,VTstart) will be a secondary index on the TIM relation.

- For each external relation R, an auxiliary internal relation R_{Tseg} is created to record start time and end time of every segment of R.

For instance, the relations in Section 2 will be translated into:

Employee (Tseg, Name, Title, Title_Level, Misc, VTstart, VTend)
Position (Tseg, Title, Title_Level, Salary, VTstart, VTend)

4. Equivalent TIM Queries

Besides translation of external relations and their indices into their TIM counterparts, TENORS also translates external queries into TIM queries.

For segmented temporal relations, a temporal segment in "left" L relation is only join-able with overlapping temporal segments in "right" R relations. For temporal join, TENORS introduces a transient *mediator table* $M(L_{TSEQ}, R_{TSEQ})$ computed as

$$\Pi_{L_{TSEQ},R_{TSEQ}}(L\_TSEQ(L_{TSEQ}, L_{START}, L_{END}) \bowtie' R\_TSEQ(R_{TSEQ}, R_{START}, R_{END}))$$

where $\bowtie' = \bowtie_{(L_{START}<R_{END})\wedge(R_{START}<L_{END})}$.

Thus it derives overlapping Tsegs for left relation L and right relation R. Example 1 is then translated into:

Example 2 *Salary history for directors in the 1990s.*

```
SELECT L.Name, R.Salary,
   COALESCE(max(L.VTstart, R.VTstart, "1/1/1990"),
      min(L.VTend, R.VTend, "12/31/1999"))
FROM Employee AS L,
      (SELECT LT.Tseg, RT.Tseg
      FROM Employee_Tseg AS LT, Position_Tseg AS RT
      WHERE LT.VTstart <= "12/31/1999"
         AND LT.VTend >= "1/1/1990"
         AND LT.VTstart <= RT.VTend
         AND LT.VTend >= RT.VTstart
      ) AS M(LTseg, RTseg), Position AS R
WHERE L.Title = "Director" AND L.Title = R.Title
   AND M.LTseg = L.Tseg AND M.RTseg = R.Tseg
   AND L.VTstart <= R.VTend AND L.VTend >= R.VTstart
   AND L.VTstart <= "12/31/1999" AND L.VTend >= "1/1/1990"
   AND R.VTstart <= "12/31/1999" AND R.VTend >= "1/1/1990"
GROUP BY L.Name, R.Salary
```

5. Implementation and Performance

To test the performance of TENORS, we created a testbed similar to the one used at TIME CENTER [5]. We have emulated a company with 100,000 come-and-go employees over 40 years of time. Using the schema shown in Section 2, we have built a database where relation Employee has 864k tuples and Position has 22k tuples. Disk storage is consumed 50M and 1M, respectively. All experiments were carried out on DB2 UDB 5.2 on a Sun UltraSPARC 10 station with 128M main memory, running under Solaris 2.6.

Figure 2 shows the result of Example 2. It shows that usefulness based segmentation speeds up join queries by two-fold. Unlike the partition join method used in [4], no extra partitioning strategies are needed since the tuples are clustered by their VTstart attribute.

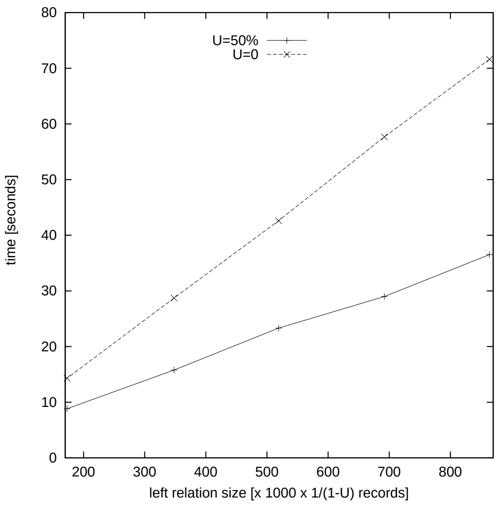

Figure 2. Performance of Temporal Joins

References

[1] C. X. Chen and C. Zaniolo. Universal temporal extensions for data languages. In *Proceedings of the 15th International Conference on Data Engineering*, pages 428–437, 1999.

[2] N. A. Lorentzos and Y. G. Mitsopoulos. Sql extension for interval data. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):480–499, 1997.

[3] R. T. Snodgrass and et al. *The TSQL2 Temporal Query Language*. Kluwer Academic, 1995.

[4] M. D. Soo, R. Snodgrass, and C. S. Jensen. Efficient evaluation of the valid-time natural join. In *Proceedings of the 10th International Conference on Data Engineering*, pages 282–292, 1994.

[5] TimeCenter. *http://www.cs.auc.dk/TimeCenter/*.

[6] D. Toman. A point-based temporal extension of sql. In *Proceedings of the 6th International Conference on Deductive and Object-Oriented Databases*, pages 103–121, 1997.

[7] C. Zaniolo, S. Ceri, C. Faloutsos, R. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Morgan-Kaufman, 1997.