# Efficient Temporal Coalescing Query Support in Relational Database Systems

Xin Zhou[1], Fusheng Wang[2], and Carlo Zaniolo[1]

[1] Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095, USA
{xinzhou,zaniolo}@cs.ucla.edu
[2] Integrated Data Systems Department
Siemens Corporate Research
Princeton, NJ 08540, USA
fusheng.wang@siemens.com

**Abstract.** The interest in and user demand for temporal databases have only increased with time; unfortunately, DBMS vendors and standard groups have not moved aggressively to extend their systems with support for transaction-time or valid-time. This can be partially attributed to the expected major R&D costs to add temporal support to RDBMS by directly extending the database engine. The newly introduced SQL:2003 standards have actually significantly enhanced our ability to support temporal applications in commercial database systems. The long recognized problem of coalescing, which is difficult to support in the framework of SQL:1992, can now be effectively supported in RDBMS. In this paper, we investigate alternatives of temporal coalescing queries under temporal data models in RDBMS. We provide an SQL:2003-based query algorithm and a native relational user defined aggregates (UDA) approach – both approaches only require a single scan of the database. We conclude that temporal queries can be best supported by OLAP functions supported in the current SQL:2003 standards. These new findings demonstrate that the current RDBMSs are mature enough to directly support efficient temporal queries, and provide a new paradigm for temporal database research and implementation.

## 1 Introduction

In this paper, we seek to support historical information management and temporal queries without extending current standards. Our insistence on using only current standards is inspired by the lessons learned from the very history of temporal databases, where past proposals failed to gain much acceptance in the commercial arena, in spite of great depth, breadth [1, 2] and technical elegance [3, 4]. An in-depth review of the technical (and often non-technical) reasons that doomed temporal extensions proposed in the past would provide an opportunity for a very interesting and possibly emotional discussion; but such a discussion is outside the scope of this paper. Here, we simply accept the fact that temporal extensions to existing standards are very difficult to sell, in spite of the growing pull by temporal applications; then, we move on from there by exploring solutions that do not require extending current standards. This low-road approach is hardly as glamorous as the “new temporal standards” approach pursued in the past, but it is not without interesting research challenges and opportunities, as we will show in this paper. In particular, with the introduction of SQL:2003, new opportunities are offered by recent developments that have taken information systems well beyond SQL:1992, and provide new appraoches to support temporal query models with current DBMS engines [5].

Even if temporal database and query model can be well supported in relational database framework, its implementation performance still remains a major problem which hinders commercial vendors from implementation. A very important ruler to validate the efficiency of temporal queries support is the efficiency of supporting some classical difficult queries, such as coalescing and temporal aggregates.

Temporal coalescing is a difficult issue of temporal databases. With SQL:1992, it needs nearly 30 lines of code. Performing coalescing in SQL:1992 is exceedingly inefficient [6]. While alternative algorithms are developed, they have to depend on the extension of the database engine.

As a result, in this paper, we aim at directly using SQL:2003’s OLAP support, as well as relational UDAs – which support single scan of input tuples – to provide efficient temporal coalescing. We show that temporal queries can be nicely expressed, and efficiently supported, and we conclude that RDBMS plus SQL:2003/UDA provides a new paradigm for temporal database research and implementation.

## 2 Temporal Coalescing: the Pain of Temporal Databases

Coalescing is a data restructuring operation applicable to temporal databases, which is similar to duplicate elimination in conventional databases. Coalescing merges timestamps of adjacent or overlapping tuples that have identical attribute values [6]. For instance, consider the snapshot of a temporal relation in Table 1 that records information about employees working in a company. In this table, a new timestamped tuple is generated whenever there is a change in any of the attribute values. The well-know problem with this approach is that coalescing is needed when some of the attributes are projected out [7]. Much research has focused on this problem, and the solutions proposed include the TSQL2 [3] approach, and the point-based temporal model [8].

| EMPNO | SALARY | TITLE | DEPTNO | TSTART | TEND |
|---|---|---|---|---|---|
| 1001 | 60000 | Engineer | d01 | 1995-01-01 | 1995-05-31 |
| 1001 | 70000 | Engineer | d01 | 1995-06-01 | 1995-09-30 |
| 1001 | 70000 | Sr Engineer | d02 | 1995-10-01 | 1996-01-31 |
| 1001 | 70000 | Tech Leader | d02 | 1996-02-01 | 1996-12-31 |

**Table 1. The table EMPLOYEE_HISTORY**

Suppose that a manager of the organization is interested in the history of the salary of employee “1001”. This query could be expressed using TSQL2 (Temporal SQL), which is an extension of the standard relational query language (SQL) enhanced with temporal features and predicates to manipulate temporal databases [3]. In such a query, coalescing is applied on both *EMPNO* and *SALARY* attributes, where *EMPNO* is the primary key of the temporal relation and *SALARY* is the time-varying attribute of the user interest. The TSQL2 statement of the above query looks as follows:

QUERY 1. *TSQL2's expression to query the history of the salary of employee 1001*:

```
SELECT EMPNO, SALARY
FROM EMPLOYEE_HISTORY (EMPNO, SALARY)
WHERE EMPNO = 1001
```

Applying coalescing to these four tuples generates the result shown in Table 2. Because the timestamps of the three tuples with a salary of 70000 are adjacent, the three tuples are coalesced by merging them into a single tuple with new timestamps as shown in Table 2. The *TSTART* value of the new timestamp is the *TSTART* value of the first tuple, where the *TEND* value is the *TEND* value of the last tuple.

| EMPNO | SALARY | TSTART | TEND |
|---|---|---|---|
| 1001 | 60000 | 1995-01-01 | 1995-05-31 |
| 1001 | 70000 | 1995-06-01 | 1996-12-31 |

**Table 2. The result of Query 1**

While TSQL2 provides a concise expression for such coalescing query, it is, however, not yet supported by commercial RDBMS.

In [6], Bohlen et al. propose that coalescing can be implemented through SQL implementation, main memory implementation, or DBMS implementation. The DBMS implementation approach requires modifying the underlying DBMS internals, which is exhaustive and expensive. The main memory implementation approach works by loading a relation into main memory, coalescing it, and then storing it back to the database. This approach suffers from two main problems. First, in many cases, it is impossible to load the whole relation into main memory. Second, it is an expensive task to periodically move a relation from the database to the running application and then store it back to the database. The SQL implementation approach aims at expressing coalescing operation as a set of SQL commands that run on the database and generate a coalesced relation. However, the expression of such coalescing query itself is very complex. Moreover, the query often requires several database scans as well as self-join(s) to the entire temporal relation table, which can be very expensive. The alternatives of implementing coalescing queries are briefly reviewed in Section 3.

In this paper, we provide a novel idea to use SQL:2003's analytical functions to support temporal coalescing. The paper is organized as follows. After a careful review of current coalescing query support with pure SQL:1992 standard in Section 3, we introduce our novel algorithm SSC in Section 4, which only needs one scan of incoming tuples to realize coalescing. We further show that this algorithm can be well supported under SQL:2003's framework, without any external programming extension. Section 5 tackles the problem with another approach – user-defined-aggregates (UDA), a native SQL extension that can be utilized to handle the coalescing queries. Performance study in Section 6 shows that the SQL:2003 method is much more efficient and scalable than the pure SQL:1992 approaches discussed in Section 3. Section 7 concludes our work.

## 3 Support Coalescing with Pure SQL:1992 Queries

There are several alternatives to implement coalescing queries using SQL:1992, either through SQL/PSM, cursors, or entire SQL [9]. The first two require either external programming modules or in-memory cursors, which are inefficient due to the high I/O manipulation cost. However, implementing coalescing entirely in SQL always has the problem that the coalescing query is considerably very complex and often has multiple nested "NOT EXISTS" clauses [7], as well as self-join(s). Query 2 is the pure SQL:1992 expression of the query corresponding to Query 1. Note that this query requires 6 database scans, as well as several self-joins to the entire temporal relation.

QUERY 2. *Pure SQL:1992 implementation of coalescing Query 1*:

```
WITH Temp(Salary, TSTART, END) AS
(SELECT SALARY, TSTART, END
FROM EMPLOYEE_HISTORY
WHERE EMPNO = 1001 )

SELECT DISTINCT F.Salary, F.TSTART, F.TEND
FROM Temp AS F, Temp AS L
WHERE F.TSTART < L.TEND AND F.Salary = L.Salary
AND NOT EXISTS
(SELECT * FROM Temp AS M
WHERE M.Salary = F.Salary
AND F.TSTART < M. TSTART AND M.TSTART < L.TEND
AND NOT EXISTS
(SELECT * FROM Temp AS T1
    WHERE T1.Salary = F.Salary
    AND T1. TSTART < M. TSTART AND M.TSTART <= T1.TEND)
)
AND NOT EXISTS
(SELECT * FROM Temp AS T2
WHERE T2.Salary = F.Salary
AND
    ((T2. TSTART < F. TSTART AND F.TSTART <= T2.TEND)
    OR
    (T2.TSTART < L.TEND AND L.TEND < T2.TEND))
)
```

Other alternatives to implement coalescing entirely in traditional SQL include: i) using COUNT aggregate instead of NOT EXISTS clauses [9], and ii) using recursive SQL queries [10]. Although the coalescing queries in these alternatives are relatively shorter than Query 2 and require fewer accesses to the entire temporal relation, they require heavier joins.

In summary, pure SQL:1992 support for temporal coalescing queries requires multiple table accesses as well as heavy join operations among the tables, which is far from satisfactory to support temporal database model under current SQL framework.

## 4 Support Coalescing with SQL:2003 OLAP Functions

SQL:2003 provides advanced support for analytics on moving windows, with new constructs such as OVER, PARTITION BY, PRECEDING, FOLLOWING, etc. Indeed,

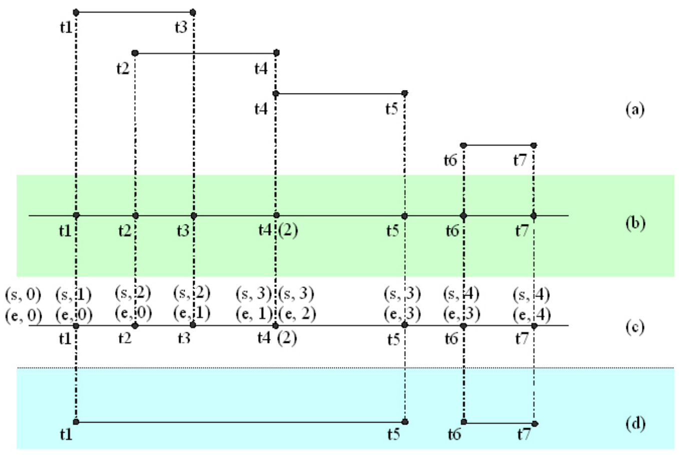

**Fig. 1.** SSC: Single Scan Coalescing Algorithm

coalescing operations can be conveniently supported using SQL:2003 standards without any extension to current SQL framework. In this section, we first show a novel Single Scan Coalescing algorithm (SSC) to support coalescing with one single scan of the input tuples without join, then we propose the SQL:2003 statements to implement this algorithm.

### 4.1 SSC: A Single Scan Coalescing Algorithm

Without loss of generality, suppose we want to coalesce four tuples with the same time-varying attribute value and different time periods, as in Figure 1 (a).

First, we detect all distinct timestamps from the input tuples. Thus in this example, we have eight distinct timestamps, as in Figure 1 (b). Notice that timestamp $t_4$ appears twice.

Next, for each timestamp, we need to keep information of whether it is a *TSTART*, or an *TEND* timestamp. We maintain two values to keep the count of *TSTART* and *TEND* timestamps which have occurred respectively, with initial value (s,0)/(e,0), and update these two values upon every new timestamp, as in Figure 1 (c). For instance, for timestamp $t_1$, since it is a *TSTART* timestamp, we get (s,1)/(e,0). For $t_2$, another *TSTART* timestamp, we increase the count of *TSTART* timestamps, and get (s,2)/(e,0). At timestamp $t_4$ where both *TSTART* and *TEND* occurred, we increase the count by one for both the *TSTART* timestamp and the *TEND* timestamp.

Last, we can output all coalesced periods, which are from timestamp $t_i$ to $t_j$, where $t_{i-1}$ has (s,m)/(e,m), and $t_j$ has (s,n)/(e,n). As in Figure 1 (d), there are two coalesced periods: $t_1$ to $t_5$, and $t_6$ to $t_7$. Intuitively, at timestamp $t_i$, all previous periods have been output as one coalesced period, and a new coalescable period begins from $t_i$. At $t_j$, we have seen equal number of *TSTART* and *TEND* timestamps, which means a coalesced period ends at $t_j$ timestamp, and $t_j$ should be the *TEND* value of our coalesced output.

We observe that if all the timestamps are ordered in the input, we can output all the coalesced periods with a single scan of all the input tuples. And in the reality of transaction time databases, all the timestamps are indeed already ordered with the passage of transaction time, which guarantees the efficiency of SSC algorithm.

## 4.2 SQL:2003 Implementation of SSC

With the introduction of SQL:2003 analytics functions, the SSC algorithm can be supported directly with pure SQL, as shown in the following example.

QUERY 3. *SQL:2003 implementation of coalescing Query 1*:

```
WITH T1 (Start_ts, End_ts, ts, salary) AS (
  SELECT 1, 0, TSTART, SALARY
  FROM EMPLOYEE_HISTORY
  WHERE EMPNO = 1001
  UNION ALL
  SELECT 0, 1, TEND, SALARY
  FROM EMPLOYEE_HISTORY
  WHERE EMPNO = 1001
),

T2 (Crt_Total_ts, Prv_Total_ts, ts, Salary) AS (
  SELECT
       sum (Start_ts) - sum(End_ts)
       OVER (PARTITION BY Salary
             ORDER BY ts, End_ts ROWS UNBOUNDED PRECEDING),
       sum (Start_ts) - sum(End_ts)
       OVER (PARTITION BY Salary
             ORDER BY ts, End_ts
             ROWS BETWEEN 1 PRECEDING AND UNBOUNDED PRECEDING),
       ts,
       Salary
  FROM T1
  WHERE Crt_Total_ts = 0 OR Prv_Total_ts = 0
)

SELECT
  Salary,
  max(ts) OVER (PARTITION BY Salary ORDER BY ts ROWS 1 PRECEDING),
  ts
FROM T2 WHERE Crt_Total_ts = 0;
```

In this implementation, the first temporary table *T1* extracts all *TSTART* or *TEND* timestamps from the input tuples, where "1" in *Start_ts* (or *End_ts*) column denotes a *TSTART* (or *TEND*) timestamp. For a certain timestamp value $t_i$, table *T2* keeps the difference between the total count of *TSTART* and the total count of *TEND* timestamps until $t_i$ (stored as *Crt_Total_ts*). Similarly, it keeps the difference between the total count of *TSTART* and the total count of *TEND* timestamps until $t_{i-1}$ (stored as *Prv_Total_ts*). Thus if *Crt_Total_ts* is 0, it means that there are equal number of *TSTART* and *TEND* timestamps at $t_i$, and $t_i$ should be a *TEND* timestamp in one of the coalesced periods. If *Prv_Total_ts* is 0, it means there are equal number of *TSTART* and *TEND* timestamps before $t_i$, so $t_i$ should be a *TSTART* timestamp for one of the coalesced periods. The final SELECT clause pairs all the result timestamps and output them.

Such an SQL statement can be predefined as a built-in coalescing function, which are transparent to the users. The beauty of this SQL statement is that, it only requires a single scan of the input tuples, since all the timestamps are already ordered. With this approach, we can implement all kinds of coalescing functionalities under the current relational database framework, without any complex extension for temporal applications.

### 4.3 Generalized SQL:2003 Implementation for Coalescing

The basic SQL:2003 implementation for salary coalescing query on employee "1001" can be very easily extended to handle all kinds of complex coalescing queries on different attributes.

– Coalescing on single attribute
  If we remove the condition that employee's *EMPNO* equals to 1001, the SQL:2003 query in Section 4.2 is for single attribute (SALARY) coalescing. For another example, if we want to return the history information of the valid periods for each *DEPTNO*, what we need to modify upon the original query is (1) remove the WHERE condition in *T1*, and (2) replace *Salary* with *DEPTNO* for every sub query.
– Coalescing on multiple attributes
  If we want to return the salary history for every employee, instead of a single employee "1001", the query will become a coalescing query on two attributes, *EMPNO* and *SALARY*. In this case, what we need to change upon the original query is (1) removing the WHERE condition in *T1*, (2) adding *EMPNO* attribute in the return clause of *T1*, and (3) using "PARTITION BY EMPNO, SALARY" to replace "PARTITION BY SALARY" in every sub query.

## 5 Support Coalescing with User-Defined Aggregates

Besides using SQL:2003 to support temporal queries in current relational DBMS, we propose another native approach using user defined aggregates (UDA). UDA is a native SQL extension for many complex data mining applications and streaming queries, which has been introduced in Oracle's latest version [11], as well as the Stream Mill system developed at UCLA [12]. UDAs can be easily used to support windows, time-series queries, and sequence queries on data streams.

As a result, we can implement a similar single scan algorithm as SSC, directly using SQL-compatible UDAs, and integrate it as system predefined aggregates for users to invoke. Due to space limitation, we only list the pseudo-code as in Algorithm 1.

Such a UDA can be correctly evaluated with the input tuples ordered by the *TSTART* values, which is realistic in transaction databases. The first (TSTART, TEND) input tuple is stored in *Temp* table (line 1-2). If the incoming tuple intersects with the current period in *Temp* table, the *TEND* value in the table will be updated to reflect the new *TEND* value (line 4-6). If the input tuple does not intersect, we get one result in the *Temp* table, which needs to be output, and deleted from *Temp* table, and the new input tuple will be stored into *Temp* table (line 7-9). We also return the final coalesced period after the last input tuple (line 12).

**Algorithm 1** UDA Pseudo-code for coalescing query with a single scan

```
1:  Define table Temp (TSTART, TEND) to store the current coalesced period, initially empty;
2:  Insert the first tuple's TSTART and TEND value into Temp;
3:  for every new input tuple T do
4:      if T.TSTART <= Temp.TEND then
5:          //new tuple coalescable with current period
6:          Update Temp.TEND with T.TEND;
7:      else
8:          //current coalesced period ends, a new coalescing period begins
9:          Output the tuple in Temp, then update Temp with T.TSTART and T.TEND;
10:     end if
11: end for
12: Output the tuple in Temp;
```

We will show in next section that this UDA approach beats the traditional pure SQL coalescing queries in performance, although it is not as efficient as the SQL:2003 SSC implementation. Nevertheless, UDA provides a native SQL support for many advanced queries, such as the classical temporal coalescing query, which otherwise needs external programming language to solve SQL's query limitation. Indeed, other complex temporal aggregates, for instance, "return all employees' average salaries along the history", can be efficiently supported with native UDA, which only requires one single scan of the input tuples. Due to the space limitation, we omit the details here.

## 6 Performance Study

We study the performance of coalescing queries with the three approaches: i) SSC approach with SQL:2003, ii) user defined aggregates approach, iii) SQL:1992 approach with "NOT EXIST" clause and iv) traditional SQL approach with recursive queries. We choose Oracle 10g Release 1 as our database server. We executed all of our queries on a personal computer equipped with an AMD Athlon XP 1500+ processor at 1.3 GHz and 512 MB of memory. The operating system we use is Fedora Core Version 3 Linux OS. The Oracle database server support SQL:2003, recursive SQL, and UDA features.

We choose a simulated employee history database as our test data. The data set models the history of employees over 17 years, and simulates the increases of salaries, changes of titles, and changes of departments. The data schema follows that in Figure 1. The total transaction database size is 120MB.

We first test the performance of single-attribute coalescing. We run two queries, coalescing on *DEPTNO* and coalescing on *TITLE*, respectively. The execution time is shown in Figure 2.

The result shows that the performance from our SSC SQL:2003 implementation beats all other approaches. The UDA approach comes close to SSC approach, with an overhead of pre-compilation and initialization time. If we run the experiments multiple times, the overhead can actually be omitted, and the UDA approach then has very close execution time with SSC approach. This proves that UDA approach is another choice

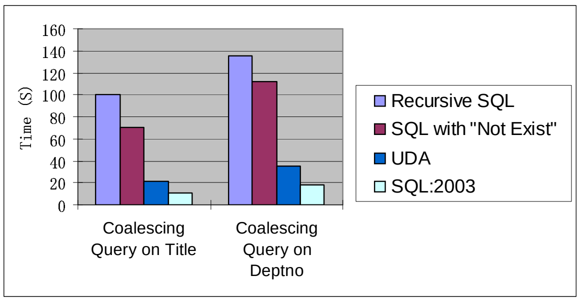

**Fig. 2.** Query Performance on Single Attrbute

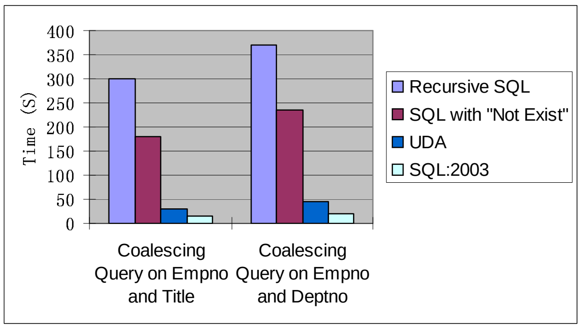

**Fig. 3.** Query Performance on Two Attrbutes

for efficient coalescing. The traditional SQL:1992 implementations such as recursive SQL or SQL with “NOT EXIST” are much slower than the two we proposed.

When it comes to two-attribute coalescing, for example, coalescing on *EMPNO* and *TITLE*, or on *EMPNO* and *DEPTNO*, the traditional SQL:1992 algorithm takes extremely long time to get the result. We have to test the four queries, on one third of the original transaction database size, and the result is shown in Figure 3. The difference ratio is similar to that in Figure 2, except that every query takes longer execution time, due to more returned tuples.

*Scalability of SSC Algorithm* We further test the scalability of our SSC SQL:2003 query, with two-attribute coalescing, on a faction of the original data set: 1/4, 1/2, and 3/4, respectively. Figure 4 shows that our algorithm is linear scalable in term of database size, which conforms to its single scan feature.

## 7 Conclusion

In this paper, we aim at directly supporting efficient temporal coalescing queries within existing commercial RDBMS, without any extension to current systems. We propose two approaches, the approach taking advantage of SQL:2003’s OLAP support, and the

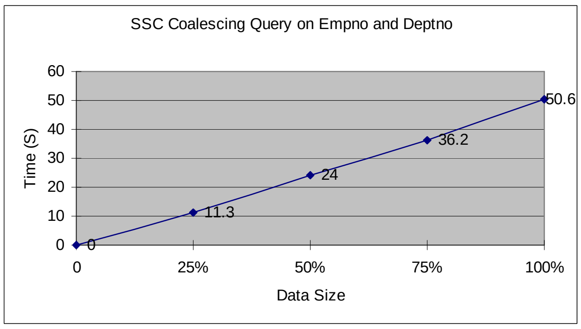

**Fig. 4.** Query Scalability of SQL:2003 Implementation of SSC

approach with user defined aggregates. Both approaches only need to perform a single scan of the database for the query execution and use minimal joins, which makes it possible to provide efficient temporal coalescing. The performance study shows that these SQL:2003 OLAP and UDA approaches hold high efficiency, especially the SQL:2003 approach. Moreover, the single scan characteristic guarantees the scalability of our approaches. The study demonstrates that current commercial RDBMS is mature enough to provide efficient complex temporal queries, and it also provides a direction for commercial temporal database support within the existing RDBMS engine.

## References

1. G. Ozsoyoglu and R.T. Snodgrass. Temporal and Real-Time Databases: A Survey. *TKDE*, 7(4):513–532, 1995.
2. F. Grandi. An Annotated Bibliography on Temporal and Evolution Aspects in the World Wide Web. In *TimeCenter Technique Report*, 2003.
3. R. T. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer, 1995.
4. Richard T. Snodgrass, Michael H. Böhlen, Christian S. Jensen, and Andreas Steiner. Transitioning Temporal Support in TSQL2 to SQL3. *Lecture Notes in Computer Science*, 1399:150–194, 1998.
5. F. Wang, C. Zaniolo, and X. Zhou. Temporal XML? SQL Strikes Back! *Time*, 2005.
6. M. H. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in Temporal Databases. In *VLDB*, 1996.
7. C. Zaniolo, S. Ceri, C.Faloutsos, R.T. Snodgrass, V.S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Morgan Kaufmann Publishers, 1997.
8. D. Toman. Point-based Temporal Extensions of SQL. In *DOOD*, pages 103–121, 1997.
9. R. T. Snodgrass. Developing Time-Oriented Database Applications in SQL. *Morgan Kaufmann*, 1999.
10. T.Y. Leung and H. Pirahesh. Querying Historical Data in IBM DB2 C/S DBMS Using Recursive SQL. In *Recent Advances in Temporal Databases*, 1995.
11. SQL 2003 Standard Support in Oracle Database 10g, otn.oracle.com/products/database/ application_development/pdf/SQL_2003_TWP.pdf.
12. C. Zaniolo, R. Luo, H. Wang, et al. Stream Mill: Bringing Power and Generality to Data Stream Management Systems. World Wide Web, http://wis.cs.ucla.edu/stream-mill/index.html.