CME: A Temporal Relational Model for Efficient Coalescing

Mohammed Al-Kateb
Department of Computer Science
College of Engineering and Mathematics
The University of Vermont
Vermont, USA
malkateb@cs.uvm.edu

Essam Mansour
Department of Computer Science
School of Computing
Dublin Institute of Technology
Dublin, Ireland
essam.mansour@dit.ie

Mohamed E. El-Sharkawi
Department of Information Systems
Faculty of Computers and Information
Cairo University
Giza, Egypt
m.elsharkawi@fci-cu.edu.eg

Abstract

Coalescing is a data restructuring operation applicable to temporal databases. It merges timestamps of adjacent or overlapping tuples that have identical attribute values. The likelihood that a temporal query employs coalescing is very high. However, coalescing is an expensive and time consuming operation. In this paper[1] we present a novel temporal relational model through which coalescing becomes quite simple. The basic idea is to augment each time-varying attribute in a temporal relation with two additional attributes that trace changes in values of the corresponding time-varying attribute. One attribute traces changes in values with respect to each individual instance (i.e. tuples having the same key value), while the other attribute traces changes in values globally for all instances (i.e. all tuples in the temporal relation). Using these tracing attributes, coalescing could be easily implemented through a quite simple join-free group-by query. The coalescing query is fully processed and optimized by the underlying database management system.

1. Introduction

Coalescing is a data restructuring operation applicable to temporal databases. It is similar to duplicate elimination in conventional databases. Coalescing merges timestamps of adjacent or overlapping tuples that have identical attribute values [2]. For instance, consider the snapshot of a temporal relation in Table 1 that shows data about employees working in a company, and assume that the manager of the organization is interested in the history of the salary of Peter.

Name	Department	Salary	Start	End
Peter	Software	200	1/1/1990	1/1/1995
Alberto	Hardware	100	2/1/1990	1/1/1996
Peter	Software	100	1/2/1995	1/1/1998
Alberto	Sales	200	1/2/1996	Now
Thomas	Hardware	100	1/1/1997	Now
Peter	Sales	100	1/2/1998	Now

Table 1. Emp relation.

Name	Salary	Start	End
Peter	200	1/1/1990	1/1/1995
Peter	100	1/2/1998	Now

Table 2. The result of Query 1.

There are three tuples for Peter. The first tuple represents Peter when he was earning 200 dollars, whereas the second and third tuples reflect the fact that Peter is earning 100 dollars. Applying coalescing to these three tuples generates the result shown in Table 2. Because the timestamps of the two tuples with a salary of 100 dollars are adjacent, coalescing merged both tuples into a single tuple with a new timestamp. The Start value of the new timestamp is the Start value of the earlier tuple, where the End value is the End value of the recent tuple.

The semantic of the above query could be expressed

[1]Part of this work was done while the first two authors were at The Faculty of Computers and Information, Cairo University, Egypt.

Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME'05)
1530-1311/05 $20.00 © 2005 **IEEE**

Department	Start	End
Software	1/1/1990	1/1/1995
Hardware	2/1/1990	1/1/1996
Hardware	1/1/1997	Now
Sales	1/2/1996	Now

Table 3. The result of Query 2.

using TSQL (Temporal SQL). TSQL [10] is an extension of the standard relational query language (SQL) enhanced with temporal features and predicates to manipulate temporal databases [15]. In such a query, coalescing is applied on both Name and Salary attributes, where Name is the primary key of the temporal relation and Salary is the time-varying attribute of the user interest. The TSQL statement of the above query looks as follows:

Query 1:

SELECT EMP.NAME, EMP.SALARY
FROM EMP (NAME, SALARY)
WHERE EMP.NAME = Peter

Moreover, in some not unusual cases, time intergaps between identical values of a time-varying attribute might exist. Therefore, coalescing has to consider this situation and reflect this fact in the coalesced result. This is illustrated in the following example. Assume that the manager is asking about the employment history of the various departments in the whole organization. As there were no employees in the Hardware department between 1/1/1996 and 1/1/1997 (i.e. an intergap), the coalesced output should be as shown in Table 3.

In this output, the Hardware department is represented in two separate tuples to reflect the fact that there is a time intergap in this value. The TSQL statement of this query looks as follows:

Query 2:

SELECT EMP.DEPARTMENT
FROM EMP (DEPARTMENT)

In both Query 1 and Query 2, the attribute(s) on which coalescing is applied are placed between parentheses in the FROM clause.

The likelihood that a temporal query employs coalescing is very high. However, coalescing is an expensive and time consuming operation. The complexity of coalescing is affected by the number of time-varying attributes on which coalescing is being applied. The more the number of time-varying attributes involved in coalescing, the more the coalescing becomes complex and expensive.

In this paper we present CME (Coalescing Made Easy), a novel temporal relational model through which coalescing becomes quite simple. The basic idea is to augment each time-varying attribute in a temporal relation with two additional attributes that trace changes in values of the corresponding time-varying attribute. One of these attributes traces changes in values with respect to each individual instance (i.e. tuples having the same key value), while the other attribute traces changes in values globally for all instances (i.e. all tuples in the temporal relation). Using these tracing attributes, coalescing could easily be implemented through a quite simple join-free group-by query. CME model has the following key distinguishing features:

1. In CME model, tuples which are candidates to coalescing are identified during insert and update operations, not at query time. That is, when an existing tuple is updated or a new tuple is inserted into a temporal relation, it is identified how this tuple will be coalesced with respect to each time-varying attribute. Hence, intuitively, query processing time is reduced.

2. CME model is extensible to support unlimited number of time-varying attributes. Regardless of the number of time-varying attributes on which the coalescing is applied, the tight number of accesses to the entire temporal relation is always *1*. That is, considering the number of accesses to the entire temporal relation as the analysis metric, for any temporal query with a coalescing operation *cop*, $cop\ (n) = \theta\ (1)$, where n is the number of time-varying attributes on which *cop* is applied.

3. In CME model, coalescing is done through a quite simple join-free group-by query, which is fully processed and optimized by the underlying DBMS. This guarantees an efficient execution of the query.

4. Because of its simplicity, CME model is easy to implement. It could be used to build a temporal stratum to conventional databases. Recently, we have implemented a beta version of a temporal stratum, named TIME (TIme Made Easy), that utilizes CME model. We give an overview of this stratum in this paper and defer the details to an incoming publication.

The rest of this paper is organized as follows. Section 2 briefly reviews the temporal data model. Section 3 discusses existing approaches for handling coalescing in temporal databases. Section 4 presents CME model and discusses its main concepts. Section 5 explains how coalescing is addressed in CME model. Section 6 presents an overview of TIME stratum as a prototype system that utilizes CME model. Finally, Section 7 concludes this paper with some further discussion.

2. Temporal Data Model

Time is an important aspect of all real-world phenomena. The ability to model the time dimension is essential to many real-world applications, such as banking, inventory control, health-care, and geographical information systems [6, 1, 12].

Temporal databases provide the ability to store the history of current data in the database so that it could be used for various querying purposes. For example, it is possible to ask about the history of an employee's salary so that the result would be all salaries the employee was earning associated with timestamps that represent the time validity for each corresponding salary value. This process of history keeping adds special characteristics to the basic database operations. For example, in update operations, the new value does not replace the old one. Rather, it does cause an insertion of a new tuple that contains the new values associated with a new timestamp. Similarly, the deletion of tuples does not cause an actual deletion from database tables. Rather, it marks the instance as removed from the database and its history is maintained as part of the history of the modeled reality.

In the context of temporal databases, two time dimensions are of interest. They are the valid time and the transaction time [11]. The valid time of a fact is the time when the fact is true in the modeled reality, whereas the transaction time is the time when the fact is actually stored in the database [3]. The database that supports only valid time is termed a valid-time database. The database that supports only transaction time is termed a transaction-time database. The database that supports both valid and transaction time is termed a bitemporal database.

Temporal relational model can support either attribute timestamping or tuple timestamping [6]. In attribute timestamping, each attribute in a temporal relation has its own timestamp that represents the time interval for the validity of the attribute value. In tuple timestamping, each tuple in a temporal relation has one timestamp that represents the time interval for the validity of values of the corresponding tuple as whole.

In this work, we assume that the underlying temporal data model supports tuple timestamping and the valid time dimension. We also assume that the insert and the update operations preserve the temporal order of tuples in the modeled reality.

3. Coalescing Approaches

3.1. Run-time and Update Coalescing

In [4], Dyreson mentions that there are two extremes to address coalescing. They are run-time coalescing and update coalescing. The run-time strategy defers coalescing to query execution time, where tuples are stored uncoalesced and during query execution, tuples are coalesced as needed. On the other hand, the update strategy performs coalescing during data update , where tuples are recoalesced when new data is inserted or existing data is modified. The intuitive merit of the update strategy over the run-time strategy is that the execution time of temporal queries is relatively smaller.

3.2. SQL, Main Memory, and DBMS Implementation

In [2], Bohlen et al. propose that coalescing could be implemented through either SQL implementation, main memory implementation, or DBMS implementation. The DBMS implementation approach requires modifying the underlying DBMS internals, which is something exhaustive and expensive. The main memory implementation approach works by loading a relation into main memory, coalesce it, and then store it back to the database. This approach suffers from two main problems. First, it might be impossible, in many cases, to load the whole relation in 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 runs on the database and generates a coalesced relation. However, usually the coalescing query is very complex and it requires several scans, as well as, self-join(s) to the entire temporal relation. The following section shortly reviews the idea of several alternatives to implement coalescing using SQL.

3.2.1 SQL Implementation

The SQL implementation approach aims at expressing coalescing operation as a set of SQL commands that runs on the database and generates a coalesced relation. However, usually the coalescing query is very complex and it requires several scans, as well as, self-join(s) to the entire temporal relation. In [9], Snodgrass presents several alternative to implement coalescing using SQL, either through SQL/PSM, cursors, or entire SQL.

SQL/PSM is the part of the SQL standard that specifies how to write persistent stored modules. It includes statements to create functions and procedures and also includes additional programming constructs to enhance the power of SQL [5]. This alternative uses the LOOP construct of PSM to express coalescing. The loop maximally extends the end date of each row and it ends when no rows are updated. Then, the algorithm removes the rows that have non-maximal periods of validity. Finally, the algorithm removes

non-sequenced duplicates. The main disadvantage of this alternative is that it requires the use of some constructs outside the SQL [9].

To implement coalescing using cursors, the SQL is used only to open a sorted cursor on the table. Although only a single scan of the entire temporal relation is required in this alternative, and although the coalescing query is a join-free one, the disadvantage here is that it is a must to pull rows into the running application via cursors, manipulate them, and then push them back into the database.

On the other hand, 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 [15], as well as, self-join(s). To express Query 1 entirely in SQL, it would be expressed through the query shown in Figure 1. The origin of this query is cited in [15]. Note that this query requires 6 accesses, as well as, several self-joins to the entire temporal relation.

Another alternative to implement coalescing entirely in SQL is to use COUNT aggregate instead of NOT EXISTS clauses [9]. Although the coalescing query in this alternative is relatively shorter than the one that is in the previous alternative, and although it requires only 3 accesses to the entire temporal relation, the order of the join operation is higher.

CME model follows the update coalescing strategy and the SQL implementation approach. However, the deferences are 1) although CME model employs the update coalescing strategy, at each insert or update operation, only the most recent tuples in the temporal relation are operated for coalescing. That is, it is not needed to recoalesce any data in the history; 2) although CME model uses the SQL implementation approach, the coalescing query is a quite simple query. It is also a join-free query. That is, no self-join is done on the entire temporal relation.

```
CREATE TABLE Temp(Salary, START, END) AS
SELECT Salary, START, END
FROM EMP
WHERE Name = Peter

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

Figure 1. An entire SQL block, with multiple nested NOT EXISTS clauses, for Query 1.

4. CME Model

First of all, unlike the efforts done in [8, 14, 7], this paper does not address an optimization problem for coalescing. Rather, it presents a novel temporal relational model through which coalescing, regardless of the number of time-varying attributes involved, could be easily implemented via a quite simple join-free group-by query. This query is fully processed and optimized by the underlying DBMS.

The idea behind CME model is to augment any time-varying attribute in a temporal relation with two additional attributes that trace changes in values of the corresponding time-varying attribute. The first attribute traces changes in values with respect to each individual instance (i.e. tuples having the same key value). The second attribute traces changes in values globally for all instances (i.e. all tuples in the temporal relation). We begin the explanation of CME model by defining both user-defined temporal relation and CME temporal relation.

Definition 1: User-defined Temporal Relation (TR)

A user-defined temporal relation; TR, is a relation with a schema: TR(K, A_i,..., A_n, TA_j,..., TA_m), where K is the primary key of TR, A_i is a time-invariant attribute such that $0 \leq i \leq n$, and TAj is a time-varying attribute such that $1 \leq j \leq m$.

Definition 2: CME Temporal Relation (CME-TR)

CME temporal relation; CME-TR, is a user-defined temporal relation augmented with two attributes; START and END to indicate the validity of values in each tuple[2]. In addition, each attribute TAj in is augmented with two attributes TAj-TL and TAj-TG. Hence, the actual schema for

[2]We assume CME model supports tuple timestamping, as mentioned in Section 2.

Name	Department	Department-TL	Department-TG	Salary	Salary-TL	Salary-TG	Start	End
Peter	Software	1	1	200	1	1	1/1/1990	1/1/1995
Alberto	Hardware	1	1	100	1	1	2/1/1990	1/1/1996
Peter	Software	1	1	100	2	1	1/2/1995	1/1/1998
Alberto	Sales	2	1	200	2	2	1/2/1996	Now
Thomas	Hardware	1	2	100	1	1	1/1/1997	Now
Peter	Sales	2	1	100	2	1	1/2/1998	Now

Table 4. Emp relation enhanced with CME model.

CME-TR becomes: TR(K, A_i,..., A_n, TA_j, TA_j-TL, TA_j-TG,..., TA_m, TA_m-TL, TA_m-TG, START, END).

When a user defines a temporal relation TR as in Definition 1, CME model converts it to a CME-TR and then the remaining processing is done on CME-TR. All of these details are totally transparent to the user.

Values of TAj-TL and TAj-TG are assigned during the insertion of a new instance or the update of an existing tuple. The massive effort to determine coalesced tuples is done at insert and update time, rather than at the query execution time. Therefore, coalescing is not repeated with each temporal query. Intuitively, such an important feature reduces the execution time of temporal queries.

First, consider the insertion of a new tuple (i.e. new instance); say *nt*. When *nt* is inserted, the value of all TAj-TL is set to *1* to reflect the fact that this value did not appear before at this particular time-varying attribute for the corresponding instance . For each TAj-TG, the system checks the most recent tuple; say *t*, whose TAi value is equivalent to TAi value in *nt*; say *tvi*. There are three cases to be considered. First, if there is no such a tuple *t*, the value of TAj-TG of *nt* is set to *1* to reflect the fact that this *tvi* appears for the first time in the whole temporal relation with respect to TAi. Second, if the END value of *t* is equal to NOW[3], it means that there is no intergap between *tvi* in *nt* and the value of TAj of *t*. Then, in this case, the value of TAj-TG of *nt* is set with the same value of TAj-TG in the tuple *t*. The third case occurs when the END value of *t* is less than NOW. It means that there is an intergap between *t* and *nt* with respect to *tvi*. Consequently, in order to reflect the existence of this intergap, the value of TAj-TG of *nt* is assigned the value of TAj-TG of *t* incremented by *1*. Figure 2 shows the insert algorithm.

Second, consider the update operation. To manipulate TAj-TG attributes, the update operation follows the same mechanism used in insert operations. However, manipulating TAj-TL attributes is slightly different. Once again, recall that the update operation causes an insertion of a new tuple; say *ut* for each updated instance (i.e. the instances that satisfy WHERE condition in the update statement). Assume that the general form of the update statement is as follows:

[3]We assume that NOW is the time when the operation is performed.

UPDATE TR
SET TAi = tvi
WHERE Condition

where TAi is the name of the time-varying attribute to be updated and *tvi* is the new value. For each updated instance, to set values of TAj-TL of *ut*, the system compares *tvi* with the corresponding value of the most recent tuple (again, say *t*) of this instance. If the two values are identical, it means that the update statement does not actually change the value of the time-varying attribute, so the value of TAj-TL of *ut* is set to the same value of TAj-TL of *t*. Otherwise, the value of TAj-TL of *ut* is set to the same value of TAj-TL of *t* incremented by *1*. If the value *tvi* appeared for the updated instance anytime before, this means that there is an intergap. Hence the increment of the value of TAj-TL reflects this fact.

```
Input: A new tuple nt (A1:V1,..., An:Vn , TA1:TV1,...,TAm:TVm).

BEGIN
  IF CME-TR is empty THEN
    FOR all nt.TAj-TL and nt.TAj-TG set value to 1
  ELSE
    BEGIN
      FOR each nt.TAj-TL set value to 1;
        FOR each nt.TAj-TG
          BEGIN
            SELECT MAX(End)
            FROM CME-TR
            WHERE CME-TR.TAj = TVj
            IF MAX(End) = null THEN
              nt.TAj-TG = 1
            ELSE IF MAX(End) ≠ NOW THEN
              nt.TAj-TG = CME-TR.TAj-TG + 1
            ELSE
              nt.TAj-TG = CME-TR.TAj-TG
          END
    END
END;
```

Figure 2. The insert algorithm.

Table 4 shows the result of applying insert and update algorithms to the Emp relation in Table 1. When inserting the first tuple, the value "1" is assigned to all TAj-TL and TAj-TG attributes. For the second tuple, which is an insertion of a new instance, the TAj-TL attributes are assigned the value "1". The TAj-TG attributes are assigned the value

"1", as well, because both the values "Hardware" and "100" of Department and Salary attributes, respectively, appear for the first time in the whole temporal relation. The third tuple is inserted because the salary of Peter has changed. To reflect this fact, the value of Salary-TL is incremented by *1* (i.e. it becomes "2"). No changes appear in the value of Department-TL because no changes happened in the value of Department attribute for Peter. Also, for the TA-TG attributes, the values remain with no changes because there are no intergaps in the timestamps of these values. Alberto has been transferred to the "Sales" department and his salary was increased to "200" . This is shown in the fourth tuple. Because both Department and Salary values have changed, for Alberto, the values of both Department-TL and Salary-TL is incremented by *1* (i.e. both become "2"). No changes are in the value of Department-TG because there is no intergap. But, for the Salary-TG, the value is incremented by 1 because of the intergap in the value "200" between 1/1/1995 and 1/2/1996. The similar mechanism is followed in the insertion of the new instance; Thomas, and the last update of Peter, shown in the fifth and the sixth tuples, respectively.

5. Coalescing In CME

Generally, two tuples in a temporal relation are candidates to coalescing if they have identical attribute values and they have adjacent or overlapping timestamps [2]. Coalescing is necessary to ensure the semantics of some temporal operators such as selection operator. Though its importance, handling coalescing has some problems. First, coalescing is relatively an expensive operation. Another problem is the problem of intergaps, introduced above, which adds more complications to the coalescing. Now, based on the data in Table 4, Query 1 is processed by the following standard SQL query:

Query 3:

SELECT Name, Salary, Salary-TL, **MIN**(START), **MAX**(END)
FROM EMP
WHERE NAME = Peter
GROUP BY Name, Salary, Salary-TL

The result of Query 3 is shown in Table 5. It should be mentioned here that as Query 1 is associated with a certain instance (i.e. the primary key "Name" of the temporal relation is embedded in the query), CME model uses the corresponding TAj-TL attribute (Salary-TL) in the group-by clause. Because the value of Salary-TL is the same for the two tuples of Peter with the salary 100 dollars, both tuples are combined in a single tuple in the output result. Now, considering Query 2, it is processed by the following standard SQL query:

Name	Salary	Salary-TL	Start	End
Peter	200	1	1/1/1990	1/1/1995
Peter	100	2	1/2/1998	Now

Table 5. The result of Query 3.

Department	Department-TG	Start	End
Software	1	1/1/1990	1/1/1995
Hardware	1	2/1/1990	1/1/1996
Hardware	2	1/1/1997	Now
Sales	1	1/2/1996	Now

Table 6. The result of Query 4.

Query 4:

SELECT Department, Department-TG, **MIN**(START), **MAX**(END)
FROM EMP
GROUP BY Department, Department-TG

The result of Query 4 is shown in Table 6. In Query 4, CME model uses the corresponding TAj-TG attribute (Department-TG) in the group-by clause because we need to trace changes of Department attribute in the whole organization. In other words, the primary key of the temporal relation is not embedded in the query. In both Query 3 and Query 4, the attributes defined by CME model (Salary-TL and Department-TG, respectively) appear in the SELECT clause to preserve SQL syntax. Consequently, they should be removed from the output shown to the user.

From the above examples, it is clear that the coalescing could be done in a quite simple, short and readable SQL query. Interesting is, regardless of the number of the time-varying attributes embedded in coalescing, the number of accesses to the entire temporal relation is always *1*. The only thing to do is to add the suitable corresponding TAj-TG and/or TAj-TL attributes to the group-by clause.

It should be mentioned here that because the majority of operations done on TIME involves the use of TAj-TL and TAj-TG attributes, we propose to create an index for each of these attributes to speed up operations and enhance performance. Through the use of these indices, locating data of interest is relatively quick.

6. TIME: A Temporal Stratum To Relational Databases

TIME is a temporal stratum to relational databases that utilizes CME model. Torp et al. conclude in [13] that the stratum approach is the best short and medium term ap-

proach to build a temporal DBMS. TIME has been implemented in JAVA and has been tested on ORACLE 8i DBMS. TIME is a platform-independent and DBMS-independent application.

The user of the stratum has the illusion that the database is managed by a TDBMS (Temporal DBMS). That is, all implementation details are transparent to the user. The user can create two types of relations, either snapshot or temporal relations. In the case of creating a snapshot relation, the system does not store the history of this relation. In the case of creating a temporal relation, the system does store the current snapshot as well as the history of the data. The user can submit queries either in SQL to access the current snapshot of the data, or in TSQL to access temporal data.

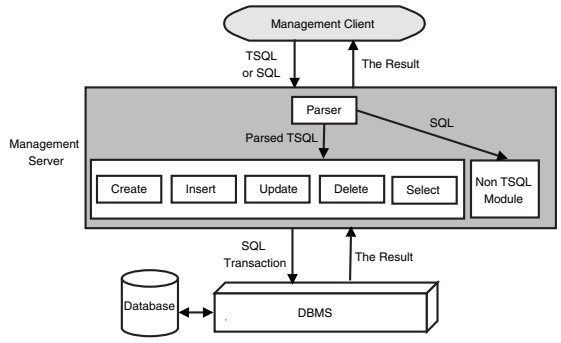

Figure 3. The conceptual architecture of TIME.

The parser module in TIME determines the type of the command issued by the user. If the command is a non-temporal command, the system passes this command directly to the underlying DBMS. Otherwise, if the command is a temporal one, the parser determines its type (either create, insert, update, delete, or select statement), and passes this command to its appropriate module. All temporal commands are transformed into non-temporal SQL commands that are executed by the underlying DBMS as an atomic transaction.

Inherited from CME model, TIME supports tuple timestamping. That is, each tuple in a temporal relation has a timestamp, which represents validity of the tuple in reality. It is also assumes that valid time and transaction time are the same. The time granularity in the current prototype is a *day*. However, TIME supports temporal casting operations, and hence, queries which use other time granularities (e.g. *month* and *year*) are also supported. A worthy merit of TIME, that comes from utilizing CME model, is that all data manipulation and query optimization are totally processed by the underlaying DBMS. Figure 3 shows the conceptual architecture of TIME.

7. Conclusion

In this paper we presented CME, a novel temporal relational model through which coalescing, regardless of the number of time-varying attributes involved, could be easily implemented via a quite simple join-free group-by query. This query is fully processed and optimized by the underlying DBMS.

We also introduced an overview of a temporal stratum, named TIME, that utilizes concepts of CME model. TIME transforms TSQL-2 query to a set of simple SQL queries that can be directly handled by underling DBMS.

The benefits of CME model are not restricted to coalescing in temporal databases, but could be extended to some other features. Currently, we are doing some research in different related areas to make a good use of CME model. The current areas of our interest are temporal XML and temporal data mining.

There are still some points which have not been addressed in this paper. The overhead added to insert and update operations should formally be estimated and verified through experimental results. The fact that each time-varying attribute requires two additional attributes, thus making the size of temporal relation larger, should be addressed. The fact that some events might occur in the modeled reality, but the reflection of these events to the database is delayed, is out of scope in this paper. Therefore, the relaxation of the assumption that insert and update operations preserve the temporal order of tuples in modeled reality is worth considering. We defer the discussion of these points to a future work that discusses TIME stratum in detail.

Acknowledgments

The authors would like to thank Professors Byung S. Lee and X. Sean Wang for their advice on this research. This research has been partially supported by the NSF under Grant No. IIS-0415023.

References

[1] T. Abraham and J. F. Roddick. Survey of spatio-temporal databases. *Geoinformatica*, 3(1):61–99, 1999.

[2] M. H. Bohlen, R. T. Snodgrass, and M. D. Soo. Coalescing in temporal databases. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 180–191. Morgan Kaufmann Publishers Inc., 1996.

[3] C. Dyreson, F. Grandi, W. Kafer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J. F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, P. Tiberio, and G. Wiederhold. A consensus glossary of temporal database concepts. *SIGMOD Rec.*, 23(1):52–64, 1994.

[4] C. E. Dyreson. Temporal coalescing with now granularity, and incomplete information. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 169–180. ACM Press, 2003.

[5] R. A. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[6] I. A. Goralwalla, A. U. Tansel, and M. T. Ozsu. Experimenting with temporal relational databases. In *CIKM '95: Proceedings of the fourth international conference on Information and knowledge management*, pages 296–303, New York, NY, USA, 1995. ACM Press.

[7] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Query plans for conventional and temporal queries involving duplicates and ordering. In *ICDE '00: Proceedings of the 16th International Conference on Data Engineering*, page 547. IEEE Computer Society, 2000.

[8] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. A foundation for conventional and temporal query optimization addressing duplicates and ordering. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):21–49, 2001.

[9] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Publishers, 2000.

[10] R. T. Snodgrass, I. Ahn, G. Ariav, D. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. K. fer, N. Kline, K. Kulkarni, T. Y. C. Leung, N. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. M. Sripada. Tsql2 language specification. *SIGMOD Rec.*, 23(1):65–86, 1994.

[11] A. Tansel. *Temporal databases theory, design, and implementation*. The Benjamin/Cummings Publishing Company, INC, 1993.

[12] P. Terenziani, F. Mastromonaco, G. Molino, and M. Torchio. Executing clinical guidelines: temporal issues. In *AMIA*, pages 848–852., 2000.

[13] K. Torp, C. S. Jensen, and R. T. Snodgrass. Stratum approaches to temporal dbms implementation. In *IDEAS '98: Proceedings of the 1998 International Symposium on Database Engineering & Applications*, page 4. IEEE Computer Society, 1998.

[14] C. Vassilakis. An optimisation scheme for coalesce/valid time selection operator sequences. *SIGMOD Record*, 29(1):38–43, 2000.

[15] C. Zaniolo, S. Ceri, C. Faloutsos, and R. T. Snodgrass. *Advanced Database Systems*. Morgan Kaufmann, 1997.