Volume : 2 | Issue : 4 | April 2013 • ISSN No 2277 - 8160

Research Paper | Engineering

A Comparison of Query Processing Architectures and Transaction Processing.

Sudhakara Reddy Mallidi — CSE department, Krishna Murthy Institute of Technology & Engineering. Affiliated to JNTUH ,Approved by AICTE, Hyderabad, INDIA

Gottipalla Ashok Kumar — CSE department, Krishna Murthy Institute of Technology & Engineering. Affiliated to JNTUH ,Approved by AICTE, Hyderabad, INDIA

ABSTRACT

A wide range of database applications manage time-varying data, and it is well known that querying and correctly updating time-varying data is difficult and error-prone when using standard SQL. Temporal extensions of SQL offer substantial benefits over SQL when managing time-varying data. The topic of this paper is the effective implementation of temporally extended SQL's. Traditionally, it has been assumed that a temporal DBMS must be built from scratch, utilizing new technologies for storage, indexing, query optimization, concurrency control, and recovery. In contrast, this paper explores the concepts and techniques involved in implementing a temporally enhanced SQL while maximally reusing the facilities of an existing SQL implementation. The topics covered span the choice of an adequate timestamp domain that Includes the time variable "NOW," a comparison of query processing architectures, and transaction processing, the latter including how to ensure ACID properties and assign timestamps to updates.

KEYWORDS: temporal databases, SQL, layered architecture, legacy issues.

1. Introduction

A wide variety of existing database applications manages time-varying data. Examples include medical, banking, insurance, and data warehousing applications. At the same time, it is widely recognized that temporal data management in SQL–92 is a complicated and error-prone proposition. Updates and queries on temporal data are complex and are thus hard to formulate correctly and subsequently understand. This insight is also not new, and following more than a decade of research, advanced query languages with built-in temporal support now exist that substantially simplify temporal data management. To be applicable in practice, a temporal language must meet the challenges of legacy code. Specifically, a temporal query language should be upward compatible with SQL–92, meaning that the operation of the bulks of legacy code is not affected when temporal support is adopted. In addition, it is desirable that a language permits for incremental exploitation of the temporal support. When a temporal language is first adopted, no existing application code takes advantage of the temporal features. Only when converting old applications and developing new ones are the benefits of temporal support achieved. To be able to make this transition to temporal support, temporal and old, "non-temporal" applications must be able to coexist smoothly. Temporal query languages effectively move complexity from the user's application to the implementation of the DBMS. The usual architecture adopted when building a temporal DBMS is the integrated architecture also used for implementing commercial relational DBMS's. This architecture allows the implement or maximum flexibility in implementing the temporal query language. This flexibility may potentially be used for developing an efficient implementation that makes use of, e.g., special-purpose indices, query optimizers, storage structures, and transaction management techniques. However, developing

a temporal DBMS with this approach is also very time consuming and resource intensive. A main reason why the layered architecture has received only little attention so far is that the ambitious performance goal has been to achieve the same performance in a temporal database (with multiple versions of data) as in a snapshot database (without versions and thus with much less data). This paper explores the implementation of temporal query languages using a layered architecture. This architecture implements a temporal query language on top of an existing relational DBMS. Here, the relational DBMS is considered a black box, in that it is not possible to modify its implementation when building the temporal DBMS. With this architecture there is a potential for reusing the services of the underlying DBMS, e.g., the concurrency control and recovery mechanisms, for implementing the extended functionality, and upward compatibility may potentially be achieved with a minimal coding effort. The major disadvantages are the entry costs that a DBMS imposes on its clients, as well as the impossibility of directly manipulating DBMS-internal data structures.

Our main design goals are upward compatibility and maximum reuse of the underlying DBMS. Another goal is that no queries should experience significantly lower performance when replacing an existing DBMS with a temporal DBMS. Throughout, we aim to achieve these goals. We consider the alternatives for a domain for timestamps, including the possible values available for representing the time variable "NOW." We show how partial parser architecture can be used for achieving upward compatibility with a minimal effort, and we discuss issues involved in implementing temporal-transaction processing.

2. The Layered Architecture

The layered architecture implements new temporal query facilities in SQL–92. The queries written in the new temporal language are then converted to SQL–92 queries that are subsequently executed by the underlying DBMS. No conversion is needed for plain SQL–92 queries. The layered temporal database architecture is shown in the given figure.

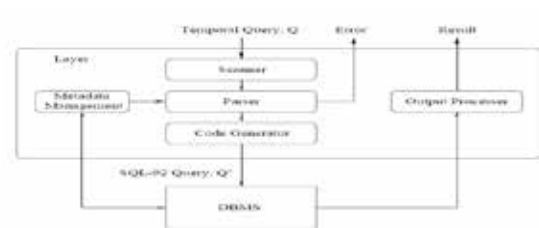


Figure: The Layered Temporal Database Architecture

3. Representing the Time Domain

The domain of the time stamped attributes can be one of the SQL–92 date time data types (DATE or TIMESTAMP). The advantage of using one of the built-in types is maximum reuse. The disadvantage is that the domain is limited to represent the Years 0001 to 9999. If the limits of the SQL–92 data types is a problem to the applications, the domain of the time attributes can be represented using a new temporal data type handled by the layer, and stored as a BIT(x) in the DBMS. The advantage of using a new temporal data type is that it can represent a much wider range of times with a finer precision. The most obvious disadvantage is

that the layer will be thicker, because all handling of the new data type must be implemented in the layer. Further, because dates are irregular, e.g, there are different numbers of days in different months, and because the arithmetic operators defined on the BIT(x) data type are regular, we cannot easily use the BIT(x) arithmetic operators in the DBMS to manipulate the new data type. As an example of these problems, notice that adding one month to a date depends on which month the date is in. The addition routine must add 31 days to a March date and 30 days to an April date. Thus, addition must be performed in the layer. Indeed, the manipulation of time attributes must to a large extent be handled by the layer. This means more tuples have to be sent from the DBMS to the layer to be processed. This again leads to a performance penalty. We have here reached one of the limitations on building on top of an existing DBMS: Adding a new data type in the layer is a major modification when the underlying DBMS does not support abstract data types. In conclusion, we recommend using the built-in data type TIMESTAMP for timestamp attributes.

4. Query Processing

This section describes different strategies for processing queries in a layered architecture. The main idea is to reduce product development time. For this purpose, several variants of partial parsers are investigated. Partial parser approaches are useful in two situations. First, they can significantly reduce the time it takes to release the first version of a new product. Today, this factor often decides whether a product is successful or not. Second, a partial parser approach is useful if many statements of a language are not affected by the (temporal) extension. The parsing of such statements does not have to be implemented.

5. Transaction Processing

In this section, we discuss how to implement ACID properties [8] of transactions in the layer by exploiting the ACID properties of the DBMS. Specifically, we show how concurrency controls and recovery mechanisms can be implemented using the services of the DBMS. Finally, the effective time stamping of database modifications is explored.

5.1 ACID Properties of Transactions

One of our design goals is to retain the desirable properties of the underlying DBMS. The ACID properties of transactions are examples of such desirable properties. The ACID properties of temporal SQL transactions are retained by mapping each temporal transaction to a single SQL–92 transaction. The alternative of allowing the layer to map a temporal SQL transaction to several SQL–92 transactions, while easing the implementation of temporal SQL transactions, leads to hard-to solve problems. To illustrate, assume that a temporal SQL transaction is mapped to two SQL–92 transactions. During execution it may then happen that one SQL–92 transaction commits but the other fails, meaning that the temporal SQL transaction fails and should be rolled back. This, however, is not easily possible—other (e.g., committed) transactions may already have seen the effects of the committed SQL–92 transaction. Next, it is generally not sufficient to simply require that each temporal SQL transaction is mapped to a single SQL–92 transaction. It must also be guaranteed that the SQL–92 transaction does not contain DDL statements. This is so because the SQL–92 standard permits DDL statements to issue implicit commits. Thus the SQL–92 transaction becomes several SQL–92 transactions, yielding the same problem as before.

Recovery is an important part of a DBMS that normally is transparent to end users. When constructing the layered approach, we are not different from end users and can rely on the recovery mechanisms implemented in the DBMS. We see no reason why recovery should be faster or slower using a layered approach. The conclusion is that the ACID properties of temporal SQL transactions are guaranteed if the SQL–92 transactions satisfy the ACID properties and if we map each temporal SQL transaction to exactly one SQL–92 transaction that does not contain DDL statements.

5.2 Timestamping of Updates

When supporting transaction time, all previously current database states are retained. Each update transaction transforms the current database state to a new current state. In practice, this is achieved by associating a pair of an insertion and a deletion time with each tuple. These times are managed by the DBMS, transparently to the user. The insertion time of a tuple indicates when the tuple became part of the current state of the database, and the deletion time indicates that the tuple is still current or when it ceased to be current.

6. Conclusion

We have investigated concepts and techniques for implementing a temporal SQL using a layered approach where the temporal SQL is implemented via a software layer on top of an existing DBMS. The layer reuses the functionality of the DBMS in order to support aspects such as access control, query optimization, concurrency control, indexing, storages, etc. While developing a full-fledged DBMS that supports a superset of SQL is a daunting task that only the major vendors can expect to accomplish, this layered technology promises much faster development. Assuming that the underlying DBMS is an SQL–92 compliant black box makes this technology inherently open and technology transferable. It may be adopted by a wide range of software vendors that would like to provide more advanced database functionality than offered by current products. With specific design goals in mind, we explored what we believe to be central issues in the layered implementation of temporal functionality on a relational SQL–92 platform. We considered the options for the domain of timestamps, and for representing the temporal database variable NOW. Then followed an exploration of different query processing architectures. We showed how the partial-parser architecture may be used for achieving upward compatibility with a minimal effort and for satisfying additional goals. Finally, we considered the processing of temporal transactions.

REFERENCES

[1] I. Ahn and R. Snodgrass. Performance Analysis of Temporal Queries. Inf.Syst., 49:103–146, 1989. [2] M. H. Böhlen. The Temporal Deductive Database System Chronolog. Ph.D.thesis, Departement Informatik, ETH Zurich, 1994. [3] M. Böhlen and C. S. Jensen. Seamless Integration of Time into SQL.TR-96–2049, Department of Computer Science, Aalborg University, Dec.1996. [4] M. Böhlen, C. S. Jensen, and R. T. Snodgrass. Evaluating and Enhancingthe Completeness of TSQL2. TR 95-05, Department of Computer Science,University of Arizona, June 1995. [5] J. Clifford and A. Tuzhilin, editors. Recent Advances in Temporal Databases.Workshops in Computing Series, Springer-Verlag, Nov. 1995. [6] C. Davies, B. Lazell, M. Hughes, and L. Cooper. Time is Just AnotherAttribute—or at Least, Just Another Dimension, pp. 175–193. In [5]. [7] M. Finger and P. McBrien. On the Semantics of 'Current-Time' In Temporal Databases. 11th Brazilian Symposium on Databases, pp. 324–337, Oct. 1996. [8] J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, 1993. [9] T. Y. C. Leung and H. Pirahesh. Querying Historical Data in IBM DB2 C/S DBMS Using Recursive SQL, pp. 315–331. In [5]. [10] J. Melton and A. R. Simon. Understanding the New SQL: A Complete Guide. Morgan Kaufmann Publishers, 1993. [11] Oracle Corp. Oracle7 Server Concepts Release 7.2, March 1995. [12] B. Salzberg. Timestamping After Commit. In Proceedings of the Third International Conference on Parallel and Distributed Information Systems,pp. 160–167, Sep. 1994. [13] A. R. Simon. Strategic Database Technology: Management for the Year 2000. Morgan Kaufmann Publishers, 1995. [14] R. T. Snodgrass. The Temporal Query Language TQuel. ACM Trans. On Database Systems, 12(2):247–298, June 1987. [15] R. T. Snodgrass and I. Ahn. Temporal Databases. IEEE Computer, 19(9):35–42, Sep. 1986. [16] R. T. Snodgrass. A Road Map of Additions to SQL/Temporal. ANSI, Feb.1996. [17] R. T. Snodgrass, editor. The TSQL2 Temporal Query Language. Kluwer Academic Publishers, 1995. [18] M. Stonebraker and G. Kemnitz. The Postgres Next-generation Database Management System. Comm. of the ACM, 34(10):78–92, Oct. 1991. [19] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass,editors. Temporal Databases: Theory, Design, and Implementation. Benjamin/Cummings Publishers, 1993.