Maintaining Temporal Consistency: Issues and Algorithms

Ming Xiong, John A. Stankovic, Krithi Ramamritham, Don Towsley, Rajendran Sivasankaran

Department of Computer Science
University of Massachusetts
Amherst MA 01003
e-mail: {xiong, stankovic, krithi, towsley, sivasank}@cs.umass.edu

Abstract

Although transaction scheduling and concurrency control issues that arise in real-time databases have been studied in detail, insufficient attention has been paid to issues that arise when real-time transactions access data with temporal validity. Such transactions must not only meet their deadlines but also read and use data that correctly reflects the environment. In this paper, we discuss the issues involved in the design of a real-time active database which maintains data temporal consistency. The concept of data-deadline is introduced and time cognizant transaction scheduling algorithms which exploit the semantics of data and transactions based on data-deadline are proposed.

1 Introduction

A real-time database system is a transaction processing system designed to handle workloads where transactions have deadlines. However, many real-world applications involve not only transactions with time constraints, but also data with time constraints. Such data, typically obtained from sensors, become inaccurate with the passage of time. Examples of such applications include autopilot systems, robot navigation, and program stock trading [13]. While considerable work has been done on real-time databases, most of it only assumes that transactions have deadlines [1, 7, 8, 10].

Database systems in which time validity intervals are associated with the data were discussed in [4, 16, 17, 9]. Such systems introduce the need to maintain data temporal consistency in addition to logical consistency. The performance of several concurrency control algorithms for maintaining temporal consistency is studied in [16, 17]. In the model introduced in [16, 17], a real-time system consists of periodic tasks which are either read-only, write-only or update (read-write) transactions. Data objects are temporally inconsistent when their ages or dispersions [18] are greater than the absolute or relative thresholds allowed by the application. Two-phase locking and optimistic concurrency control algorithms, as well as rate-monotonic and earliest deadline first scheduling algorithms are studied [17, 18]. It is pointed out in [18] that it is difficult to maintain the data and transaction time constraints due to the following reasons:

1. A transient overload may cause transactions to miss their deadlines.
2. Data values may become out of date due to delayed updates.
3. Priority based scheduling can cause preemptions which may cause the data read by the transactions to become temporally inconsistent by the time they are used.
4. Traditional concurrency control ensures logical data consistency, but may cause temporal data inconsistency.

In [9], the semantics of data-intensive real-time applications that should maintain data temporal consistency is discussed, and the notion of similarity which is used to provide more flexibility in concurrency control is formalized. Weaker consistency requirements based on the similarity notion are proposed.

Other work [2] discusses how to keep the database "fresh", and at the same time ensure that transactions read valid data and meet their deadlines.

Two real-time active database applications are discussed and the applicability of ECA (Event-Condition-Action) paradigm from active databases in these applications is explored in [11]. In this paper, we examine some of the questions associated with maintaining temporal consistency in real-time active databases so as to achieve better performance. These include:

1. how to maintain data temporal consistency to maximize system performance?
2. how to handle overloads?
3. what are the proper mechanisms for time cognizant conflict resolution and scheduling?

In this paper, after discussing the various considerations that are involved in answering these questions, we propose new priority assignment algorithms which handle both transaction deadlines and data validity constraints.

2 Model and Transaction Correctness in Real-Time Databases

2.1 The Model

We consider a firm real-time active database system which handles triggering and triggered transactions. Each transaction can be either a read-only, a write-only or an update transaction. A *database* consists of a set of objects and a set of ECA (Event-Condition-Action) [6] rules. An *object* models a real world entity, for example, the position of an aircraft. The objects in the database can be either temporal objects or non-temporal objects. Here a temporal object is one whose state may become invalid with the passage of time. Associated with the state is a temporal validity interval. An object whose state does not become invalid with the passage of time is a non-temporal object. Thus, there is no temporal validity interval associated with the state of non-temporal objects. Two approaches to model temporal data have been proposed in the literature: attribute versioning and object versioning. In attribute versioning, each attribute of an object is associated with a validity interval, whereas in object versioning, the aggregate object is associated with a validity interval. We use object versioning which maintains multiple versions of each object. Each state of a temporal object is associated with a time-stamp and a validity interval during which the state is valid. We categorize temporal objects into two types: *base objects* and *derived objects*. Base objects are those that reflect specific objects in the environment. They are updated periodically by transactions that read sensors. Other transactions may derive new data objects from these base objects. Such objects are called derived objects. In our model, a transaction may be triggered according to the ECA rules in the database when a base object is updated by a sensor transaction. The triggered transaction may update the states of derived objects whose states are derived from that base object. There are three types of transactions:

- **Sensor transactions**: These are the periodic transactions which write to base objects.

- **Triggered update transactions**: These are transactions triggered by the updates of base objects. They update the derived objects.

- **User transactions**: These are user-level transactions with deadlines. They can be read-only, write-only or update (read-write) transactions.

The deadlines of sensor transactions and triggered update transactions are derived from the requirement that transactions should update an object before the end of the validity interval associated with the data in order to keep the data in the database fresh. Here fresh means absolute consistency which will be introduced in the next subsection.

2.2 Transaction Correctness

In real-time applications, the values of objects in a database must correctly reflect the state of the environment. Otherwise, the decisions made based on the data in the database may be wrong, and potentially disastrous. For example, not only must the data read by transactions be fresh, but also be temporally correlated. This leads to the notion of temporal consistency which consists of two components [4, 16, 17, 18]: absolute consistency and relative consistency. To define temporal consistency formally, we use definitions from [13] which denote a data item d, in the real-time database by

$$(d_{value}, d_{timestamp}, d_{avi})$$

where d_{value} denotes the current state of d, and $d_{timestamp}$ denotes the time when the observation relating to d was made. d_{avi} denotes d's *absolute validity interval*, i.e., the length of the time interval following $d_{timestamp}$ during which d is considered to be absolutely consistent.
A set of data items, R, used to derive a new data item is a *relative consistency set*. Associated with R is a *relative validity interval* denoted by R_{rvi}.
R has a correct state at time $t > 0$ iff

1. $\forall d \in R$, d_{value} is logically consistent - satisfies all integrity constraints.
2. R is temporally consistent:
   (a) $\forall d \in R, d$ is absolutely consistent, i.e., $(t - d_{timestamp}) \leq d_{avi}$.
   (b) R is relatively consistent, i.e., $\forall d, d^{'} \in R, \mid d_{timestamp} - d^{'}_{timestamp} \mid \leq R_{rvi}$.

Therefore, a transaction in real-time databases can commit iff

1. it is logically consistent,
2. it meets its deadline, and
3. it reads temporally consistent sets of data, and the data it read are still fresh when it commits.

3 Ensuring Data Temporal Consistency

Maintaining temporal consistency raises many issues including the determination of optimal period values for sensor transactions, version selection, handling bursty arrivals of sensor transactions, timely update of derived data, issues of forced delay, and concurrency control.

3.1 Period for Sensor Transactions

Consider the freshness, or absolute consistency of data in the database. Absolute consistency of data is provided by the periodic sampling of the environment, that is, by executing the sensor transaction that obtains data from a sensor and updates periodically the database. The issue here, as pointed out in [13], is how the period of the sensor transaction should be set.

Consider one of the many possible semantics of sensor transactions with period P: One instance of the sensor transaction must execute every period and, so long as the start time and the completion time lie within a period, its execution is considered to be correct. Suppose a sensor transaction takes at most e units of time to complete $(0 \leq e \leq P)$. If an instance starts at time t and ends at $(t + e)$, and the next instance starts at $(t + 2P - e)$ and ends at $(t + 2P)$, then we have two instances, which are separated by $(2P - e)$ units of time in the worst case. This, for example, will be the case if the rate monotonic static priority approach is used. It follows from the example that the maintenance of data absolute consistency requires the period of the corresponding sensor transactions to be no more than half of the length of the absolute validity interval. This indicates that more than one version of a temporal object may exist simultaneously in the database.

3.2 Version Selection

Since there may be more than one version of a temporal object in the database, the transaction should select a version to maintain the relative consistency of data it reads. If a transaction reads the most recent version of an object, it will have more time to commit since this version will have the longest validity. On the other hand, if the transaction reads the most recent version of an object, it may violate the relative consistency requirement. Consider the following opportunistic version selection approach: if x is the first data item in a relative consistency set read by T, then transaction T reads the most recent version of x; otherwise, transaction T reads the most recent version of x which satisfies the relative consistency requirement. An alternate approach is to predeclare the read set of a transaction and choose the versions of data that satisfy temporal consistency. However, neither of the approaches guarantees that a transaction can maintain temporal consistency.

3.3 Bursty Arrival of Sensor Transactions

Overload can easily lead to temporal inconsistency. Consider n periodic sensor transactions $T_1, T_2, ..., T_n$ with periods $P_1, P_2, ..., P_n$ respectively. If they start *in phase*, i.e., the first periods of all the transactions begin at time $t = 0$, then the instances of those sensor transactions simultaneously start at $i * LCM(P_1, P_2, ..., P_n)$ $(i = 1, 2, ...)$ where $LCM(P_1, P_2, ..., P_n)$ is the least common multiple of $P_1, P_2, ..., P_n$. Sensor transactions along with user transactions may cause the system to be overloaded. In such situations, one solution is to *update on demand*: Give higher priority to user transactions and lower priority to sensor transactions. When a user transaction needs to read an object and finds it invalid, the corresponding sensor transaction which updates the object inherits the priority of the user transaction and updates the object. Then the user transaction can read valid data and proceed.

3.4 Timely Update of Derived Data

Since a base object and its corresponding derived objects are not updated by the same transaction, there may be a time interval during which the base object has already been updated but the derived objects, whose values depend on the base object have not yet been updated. If a user transaction attempts to read derived objects within that time interval, it may read invalid values. Since an update to a base object may trigger the updates of more than one derived object, and the value of one derived object may depend on more than one base object, an eager policy to update derived objects may cause considerable repetitive computation, and a lazy policy to update derived objects may cause user transactions to read invalid values. An alternative is to use an update on demand policy as discussed before. However, the drawback of this approach is its latency. Adelberg et. al. [3] study the intricate balance between recomputing derived data and transaction execution. They propose a policy which does not permit the recomputation of derived data until a fixed amount of time passes. Thus the recomputations triggered within that time can be batched together resulting in a reduction of the recomputation load.

Because an update to a base object may trigger updates to more than one derived object, two alternatives exist. One is to treat each update as a transaction. This increases transaction overhead although it permits the earlier release of data locks. The other is to combine all of the updates into one transaction, which can reduce transaction overhead. In this case, if the updated objects are not related to each other, an object lock can also be released as soon as the update is done. We discuss this further later in this paper.

3.5 Forced Delay

To maintain data temporal consistency, a transaction must read valid data. Otherwise, it is aborted. However, a transaction that reads valid data may not be able to commit because the validity interval of a data item it reads expires before it can commit. Although it is impossible to predict the exact time when a transaction will commit, it is possible to know the minimum execution time needed for a transaction to commit. If a transaction reads a data item whose remaining validity interval is less than the remaining execution time of the transaction, there is no chance for the transaction to commit before the validity interval of that data item expires. Such a validity interval is called an *infeasible validity interval* with respect to the transaction. To prevent a transaction from reading invalid data, or data with an infeasible validity interval, it can be forced to wait until the data is updated. That is, there must be a mechanism to check the validity of data before the transaction reads it. This approach saves the work a transaction has already done and avoid recovery overhead. However, this method has the disadvantage of having considerable overheads which in turn may unnecessarily delay the commitment of transactions.

3.6 Concurrency Control

Song and Liu [18] study the problem of how to maintain temporal consistency in real-time transaction scheduling. They use earliest deadline first and rate monotonic algorithms to schedule transactions. Lock based and optimistic concurrency control are used to maintain data integrity. They observed that optimistic concurrency control is generally worse at maintaining temporal consistency of data than lock based concurrency control, even though the former allows more transactions to meet their deadlines.

In our model, base objects are only written by sensor transactions, which are write-only transactions. Furthermore, sensor transactions do not read any data in the database; their write sets are disjoint from each other and from the write sets of update transactions. When a sensor transaction writes a base object in each period, it creates a new version of the base object. In this case, there is no need for a sensor transaction to obtain database locks in order to write. On the other hand, no transactions other than sensor transactions can write base objects. There may be multiple valid versions of an object in the database at one instant. As soon as a version becomes invalid, it is discarded. Thus, non sensor based transactions do not need to obtain database locks in order to read a valid version of an object. Therefore, there is no database concurrency control for base objects. However, to ensure that data is not read while it is being updated, latches (i.e., short term locks or semaphores) must be used.

Concurrency control is necessary for derived objects if data integrity is to be maintained. However, in some cases, the atomicity and isolation of transactions can be relaxed. Consider a triggered update transaction which does several updates. For each update, a lock is needed to maintain the consistency of the object. But the lock can be released immediately after the update is finished, if those updates are not correlated to each other. In this case, the transaction does not have to obey two phase locking rules. On the other hand, if a triggered update transaction aborts after it performs some updates, it does not have to undo the updates it has done so far since they are valid updates. When this transaction is restarted, it only needs to resume from the point it was aborted. Thus the isolation and atomicity of each update are preserved, even though the isolation and atomicity of the whole transaction are relaxed. An interesting open research question is whether it's possible to relax data logical consistency, that is, integrity constraints among interrelated data.

4 Scheduling Transactions in Real-Time Databases

The transactions in the system can be classified into two classes: user transactions and non-user transactions which include both sensor transactions and triggered update transactions. The scheduling algorithm should maximize the number of user transactions which meet their deadlines while maintaining temporal consistency. Policies that assign priorities to different transaction classes and to transactions within the same class are studied. Similar to [3], four policies can be considered to assign priority to different transaction classes:

1. Assign higher priority to user transactions.
2. Assign lower priority to user transactions.
3. Update on demand - Assign higher priority to user transactions, but if a transaction encounters stale data, the corresponding sensor transaction will be executed first to get fresh data.
4. Assign the same priority to both user and non-user transactions.

Since data read by a transaction must be valid when the transaction completes, this leads to another source of deadlines, *data-deadlines*. Within the same transaction class, the scheduling algorithm should be aware of the *data-deadline* of a transaction, that is, the time after which the transaction will violate temporal consistency. The data-deadline of a transaction differs from the deadline of the transaction because the transaction may violate temporal consistency after the data-deadline, although it has not yet missed its deadline. The scheduling algorithm should account for data-deadlines when it schedules transactions if a data-deadline is less than the corresponding transaction deadline. Consider the following example which illustrates the concept of data-deadlines.

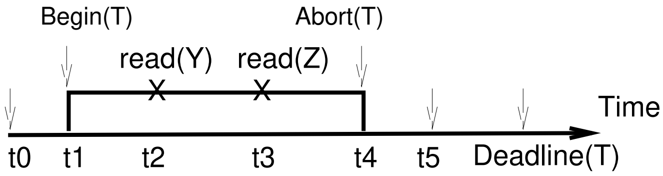

Figure 1. An example to illustrate data-deadline

- Transaction T needs to read two data items, Y and Z, to produce results. Consider the scenario in figure 1. Each $\times$ indicates a read from transaction T. The deadline of transaction T is denoted as Deadline(T). Data Y is valid in the interval $[t_0, t_5]$, and data Z is valid in the interval $[t_0, t_4]$. Transaction T starts at time t_1, and the data-deadline of T equals its deadline at this time. At time t_2, transaction T reads data Y. The data-deadline of transaction T becomes t_5 since it will violate temporal consistency after time t_5. In order to satisfy temporal consistency, T has to be scheduled to commit before time t_5, i.e., before the value of Y it read becomes invalid. Notice the deadline of transaction T is far later than time t_5. Next, transaction T proceeds and reads data Z with a value which will be invalid at time t_4. Now the data-deadline of T is adjusted to t_4. At time t_4, transaction T hasn't completed. Thus it aborts. However, the scheduler can start it again if there is sufficient time for the transaction to meet its deadline.

The data-deadline of transaction T, which is denoted as data-deadline$_T$, is assigned by the following algorithm:

$Begin_T$: /* At the beginning of transaction T */
data-deadline$_T$ = deadline$_T$;
$Read(T, X)$: /* After transaction T reads data X with validity interval $[begin_X, end_X]$ */
data-deadline$_T$ = min(data-deadline$_T$, end$_X$);

Consider the following priority assignment policy:

$$\text{priority}_T = \alpha * \text{data-deadline}_T + (1 - \alpha) * \text{deadline}_T;$$

where scheduling policies depend on $\alpha (0 \leq \alpha \leq 1)$. The following scheduling policies are defined by specific values of α.

1. *Earliest Deadline First* (EDF): $\alpha = 0$.
   This algorithm only takes deadline information into account and neglects the temporal consistency constraints. Therefore, a transaction might be aborted due to temporal inconsistency and restarted again.
2. *Earliest Data Deadline First* (EDDF): $\alpha = 1$.
   To take into account temporal consistency constraints, transactions are scheduled according to data-deadline information. The risk is that a transaction T_1 with a long deadline, but tight data-deadline is given higher priority than another transaction T_2 with a tight deadline. This is because T_2 has a longer data-deadline than T_1 does. Therefore, transaction T_2 might not be able to complete before its own deadline. But if transaction T_2 is given higher priority, both T_1 and T_2 may be able to commit.
3. *Work Proportional Approach*(WP): α is the fraction of work that has been done.
   This algorithm is a compromise between the first two. Here α reflects the percentage of work that a transaction has completed. The algorithm indicates that the more work a transaction has done, the heavier the weight of the transaction's data-deadline in the calculation of its priority.
4. *Median Approach*:

$$\alpha = \begin{cases} 0 & \text{if fraction of work has been done} \leq 0.5 \\ 1 & \text{Otherwise} \end{cases}$$

   This algorithm is another compromise between the first two. It assigns the priority of a transaction according to its deadline if it has not completed half of its work. Otherwise, the priority is assigned according to the data-deadline of the transaction.

Notice that the WP and Median approaches can only be used if the total amount of work of a transaction is known a priori. They do require that the system keep track of the amount of work a transaction has done. The motivation behind those two approaches is to save the work a transaction has done if it has already completed most of its work, and to avoid recovery overhead.

5 Conclusions

Although transaction scheduling and concurrency control aspects of real-time databases have been studied in detail, not much attention has focused on maintaining temporal consistency. The question of how to improve system performance while transactions maintain data temporal consistency and meet their deadlines poses a new and challenging problem. In this paper, we address issues involved in designing a real-time active database which ensures data temporal consistency. We define transaction correctness based on its logical consistency, temporal consistency and deadline constraint. We also discuss problems in updating base objects and derived objects, and in maintaining data temporal consistency of user transactions as well. We have proposed a number of algorithms for scheduling transactions based on *data-deadlines*.
We are developing a real-time active database simulator to study the different issues and scheduling policies presented in this paper, and their tradeoffs. The simulator has the following components :

- DBManager - specification of the data model.
- Workload Generator - simulates the application or the environment.
- Transaction Manager - schedules the transactions.
- Object Manager - responsible for concurrency control.
- Rule Manager - triggers rules depending on the database events.
- Resource Manager - simulates CPU usage.

In addition, we also intend to study buffer management as well as logging and recovery issues involved in maintaining data temporal consistency in real-time active databases.

References

[1] Robert Abbort and Hector Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *ACM Transactions on Database Systems*, Vol. 17, No. 3, pp. 513-560, September 1992.

[2] B. Adelberg, H. Garcia-Molina and B. Kao, "Applying Update Streams in a Soft Real-Time Database System," *Proceedings of the 1995 ACM SIGMOD*, pp. 245 - 256, 1995.

[3] B. Adelberg, H. Garcia-Molina and B. Kao, "Database Support for Efficiently Maintaining Derived Data," Technical Report, Stanford University, 1995.

[4] N. Audsley, A. Burns, M. Richardson, and A. Wellings, "A Database Model for Hard Real-Time Systems," Technical Report, Real-Time Systems Group, Univ. of York, U.K., July 1991.

[5] Michael J. Carey, Rajiv Jauhari and Miron Livny, "On Transaction Boundaries in Active Databases: A Performance Perspective," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 3, No. 3, pp. 320-336, September 1991.

[6] U. Dayal et. al., "The HIPAC Project: Combining Active Databases and Timing Constraints," *SIGMOD Record*, Vol. 17, No. 1, pp. 51-70, March 1988.

[7] J.R. Haritsa, M.J. Carey, and M. Livny, " Earliest Deadline Scheduling for Real-Time Database Systems," *Proceedings of the Real-Time Systems Symposium*, pp. 232-242, December 1991.

[8] J. Huang, J.A. Stankovic, K. Ramamritham, and D. Towsley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes," *Proceedings of the 17th Conference on Very Large Databases*, pp. 35-46, September 1991.

[9] Tei-Wei Kuo and Aloysius K. Mok, "Real-Time Data Semantics and Similarity-Based Concurrency Control," submitted to *IEEE Transactions on Knowledge and Data Engineering*, 1995.

[10] Y. Lin and S.H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *Proceedings of the Real-Time Systems Symposium*, pp. 104-112, December 1990.

[11] B. Purimetla, R. M. Sivasankaran, J.Stankovic and K. Ramamritham, "Network Services Databases - A Distributed Active Real-Time Database (DARTDB) Applications," *IEEE Workshop on Parallel and Distributed Real-time Systems*, April 1993.

[12] B. Purimetla, R. Sivasankaran, J.A. Stankovic, K. Ramamritham and D. Towsley, "Priority Assignment in Real-Time Active Databases," *Conference on Parallel and Distributed Information Systems*, October 1994.

[13] K. Ramamritham, "Real-Time Databases", *Distributed and Parallel Databases* 1(1993), pp. 199-226, 1993.

[14] K. Ramamritham, "Where Do Deadlines Come from and Where Do They Go?" *Journal of Database Management*, (to appear) 1996.

[15] R. M. Sivasankaran, K. Ramamritham, J. A. Stankovic, and D. Towsley, "Data Placement, Logging and Recovery in Real-Time Active Databases," *Workshop on Active Real-Time Database Systems,* Sweden, June 1995.

[16] X. Song and J. W. S. Liu, "How Well Can Data Temporal Consistency be Maintained?" *IEEE Symposium on Computer-Aided Control Systems Design*, 1992.

[17] X. Song, "Data Temporal Consistency in Hard Real-Time Systems," Technical Report No. UIUCDCS-R-92-1753, 1992.

[18] X. Song and J. W. S. Liu, "Maintaining Temporal Consistency: Pessimistic vs. Optimistic Concurrency Control," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 5, pp. 786-796, October 1995.