Multiversion Divergence Control of Time Fuzziness

Calton Pu*

Department of Computer Science and Engineering
Oregon Graduate Institute
Portland, OR 97291-1000

Miu K. Tsang, Kun-Lung Wu and Philip S. Yu

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Abstract

Epsilon Serializability (ESR) has been proposed to manage and control inconsistency in extending the classic transaction processing. ESR increases system concurrency by tolerating a bounded amount of inconsistency. In this paper, we present multiversion divergence control (mvDC) algorithms that support ESR with not only *value* but also *time* fuzziness in multiversion databases. Unlike value fuzziness, accumulating time fuzziness is semantically different. A simple summation of the length of two time intervals may either underestimate the total time fuzziness, resulting in incorrect execution, or overestimate the total time fuzziness, unnecessarily degrading the effectiveness of mvESR. We present a new operation, called TimeUnion, to accurately accumulate the total time fuzziness. Because of the accurate control of time and value fuzziness by the mvDC algorithm, mvESR is very suitable for the use of multiversion databases for real-time applications that may tolerate a limited degree of data inconsistency but prefer more data recency.

1 Introduction

In conventional transaction processing (TP) systems, the correctness criterion has been serializability (SR) [3]. *Epsilon Serializability* (ESR) [11, 17, 10] is a compatible extension that relaxes SR's consistency constraints. In ESR, each transaction, called *epsilon transaction* (ET), has a specification of the inconsistency (fuzziness) allowed in its execution. ESR increases TP system concurrency by tolerating a bounded amount of inconsistency. For example, in a bank database the inconsistency bound can be defined by dollars.

In this paper, we apply ESR to a database system that maintains multiple versions of data, called *multiversion epsilon serializability* (mvESR), and present multiversion divergence control algorithms (mvDC) that guarantee mvESR. Note that multiversion databases and multiversion concurrency control (mvCC) algorithms have been proposed to efficiently support concurrent processing of update transactions and read-only queries by allowing update transactions and queries to access different versions of data. In a way similar to mvCC algorithms improving concurrency of single version concurrency control (1vCC), mvDC algorithms presented in this paper allow more concurrency, compared with single version divergence control (1vDC). One particularly important contribution of this paper is the addition of the time dimension to mvDC algorithms to bound not only the value fuzziness of the result of a query ET but also the timeliness, or recency, of the data that a query ET accesses. In fact, mvDC algorithms allow the results of query ETs to be less obsolete than mvCC algorithms since non-serializable executions are possible and query ETs can access more recent versions. As a result, version management can be greatly simplified by only maintaining the most recent few versions. More importantly, mvESR becomes attractive for the use of multiversion databases for real-time applications that may want to trade off a bounded degree of data inconsistency for more data recency.

In this paper, we describe mvDC algorithms that support ESR along two dimensions: time fuzziness and value fuzziness. Unlike value fuzziness, accumulating time fuzziness is semantically different. A simple summation of the length of two time intervals may underestimate the total time fuzziness, resulting in incorrect execution, or significantly overestimate the total time fuzziness, unnecessarily degrading the effectiveness of mvESR. We present a new operation, called TimeUnion, to correctly accumulate the total time fuzziness imported by a query ET or exported by an update ET. Two mvDC algorithms are presented: one based on a timestamp ordering mvCC algorithm and the other based on a 2-phase locking mvCC algorithm. The serialization order of the timestamp ordering mvCC algorithm is determined by the initiation timestamps of transactions or queries, while that of the 2-phase locking mvCC algorithm is determined by the commit timestamps. We demonstrate first the bounding of time fuzziness in the timestamp mvDC algorithm and then the bounding of both time and value fuzziness in the 2-phase locking mvDC algorithm.

The rest of the paper is organized as follows. Section 2 summarizes the background and motivation for this work. Section 3 introduces the concepts related to time fuzziness. Section 4 briefly discusses a few version selection functions for picking a version. Section 5 describes the timestamp ordering multiversion divergence control for time fuzziness only. Section 6 discusses the bounding of both time and value fuzziness. Section 7 then describes a 2 phase locking mvDC algorithm that bounds both value and time fuzziness. Section 8 concludes the paper.

*This work was partially supported by National Science Foundation and Oki Electric.

CIKM '94- 11/94 Gaitherburg MD USA
© 1994 ACM 0-89791-674-3/94/0011..$3.50

2 Background and motivation

2.1 Multiversion databases

Multiversion concurrency control algorithms have been proposed to decrease the number of conflicts between reads and writes by separating writes (creation of new versions) from reads (on immutable versions). Most of the proposed mvCC algorithms allow a multiversion history that is equivalent to a standard serial multiversion schedule. For example, a transaction T_i reads the most recent version of data items according to the serialization order of T_i. These algorithms are particularly effective for long queries, which otherwise cannot finish without stopping the update activity.

However, most of the classic mvCC algorithms implicitly assume the database accommodates an infinite number of versions for each data item [1, 2, 3, 4, 5, 6, 13, 15]. If not, a long-running query may have to be aborted because some of the versions that it needs have been garbage collected prematurely. Thus, both storage overhead and version management complexity, such as version selection and garbage collection, can be a severe problem. Recently, a new class of mvCC algorithms, such as dynamic finite versioning (DFV) [16, 8] and transient versioning [9], have been proposed to allow a choice in the trade-off between the number of versions available (i.e., the storage space occupied) and the obsolescence of data being accessed. Queries are guaranteed to read from a consistent database state, but that consistent state may be out-of-date [8]. Usually, a smaller number of versions available means less up-to-date versions being read. In contrast to these versioning algorithms that maintain a finite number of versions and preserve serializability, multiversion DC algorithms offer a different kind of trade-off. Instead of obsolescence, ESR allows query ETs to access more recent, but maybe non-SR versions. This is orthogonal to these finite versioning schemes and in principle the two approaches can be combined.

Compared to single version DC algorithms, multiversion DC can provide results with smaller fuzziness, simply because more versions are available and query ETs can carefully choose the "right" version. Furthermore, many long queries are generally simple aggregate functions (or composed of these functions) over large portions or the entire database such as summations. Usually, these aggregate functions can tolerate a limited amount of fuzziness. For example, the total amount in the checking accounts of a bank is reported in millions of dollars; consequently an inconsistency of a hundred thousand would be washed out by the rounding error. In these applications, ESR offers a higher degree of concurrency because limited non-serializable conflicts are allowed, and a higher degree of recency for the same reason.

2.2 Real-Time database applications

One important application of mvESR is the use of time-stamped multiversion databases for real-time applications that also require timely results. Real-time jobs have deadlines, beyond which their results have little value. Furthermore, many real-time jobs may have to produce timely results as well, where more recent data are preferred over older data. Classic TP based on SR does not include time or timeliness in the database model. Consequently, 1vCC algorithms may postpone a job too much into the *future* (blocked because of access conflicts), causing it to miss the deadline. On the other hand, mvCC algorithms may push a job too much into the *past* (in an effort to avoid conflicts), producing obsolete results. We say that real-time jobs have *urgency* requirements to finish before the deadline and *recency* requirements to avoid old data. Although algorithms controlling only value fuzziness can help increase concurrency of real-time databases [14], most real-time applications may also require a tight control over the time fuzziness in addition to value fuzziness. Intuitively, time fuzziness denotes how wide-spread in time the accessed data objects are, from the viewpoint of a serializable schedule.

One common problem of DFV or transient versioning algorithms is the lack of programmer control over the degree of obsolescence of data being read by queries. Consequently, most of the multiversion CC algorithms have serious drawbacks in the context of real-time database applications. These drawbacks are particularly severe for the real-time applications that need to access recent data, not just an arbitrarily old version, in addition to completion before the deadlines. Usually, real-time transactions may tolerate some limited amount of inconsistency (SR not required), but they prefer data timeliness (versions not too old). In these situations, classic multiversion CC algorithms become inadequate, since they do not support the notion of data timeliness. The goal of mvDC algorithms described in this paper is to use the tolerance for time fuzziness to increase the number of versions selectable for each query ET, thereby guaranteeing data timeliness.

One way to deal with the trade-off between data consistency and timeliness is the conservative scheduling of data access. Data consistency is maintained by scheduling transactions under SR, and data timeliness is maintained by the application programmers. In addition, the system must schedule the jobs carefully to guarantee that real-time transactions will meet their deadlines. But this conservative approach is too restrictive, since SR may not be required in many real-time applications, where sufficiently "close" versions may be more important than strict SR. Thus, mvESR offers an attractive solution in this case.

More recently, Mok [7] has introduced the notion of *similarities* to allow some fuzziness in the trade-off of data consistency and timeliness. Our work has the same goal as Mok's, but we build on the existing ESR framework, using ESR concepts, notation, and algorithms. In this paper, we apply ESR to transaction processing in multiversion databases, aiming to support real-time applications that have both urgency and recency requirements. Our algorithms allow application designers a fine-grained control by explicitly quantifying imprecision of data (in time and in value) due to bounded fuzziness in non-serializable execution. This way, queries may read inconsistent data as long as the data versions being read are "close enough" to serialization in data value and timeliness. Therefore, we can have many queries running without keeping all the versions that would have been otherwise needed for a serializable execution.

3 Time fuzziness in mvESR

3.1 Specifying time fuzziness

A database is a set of data items. The set of all possible values over the entire database is the database state space, denoted by S_{DB}. In this paper, we consider only cartesian database state spaces (e.g., integers, real numbers, sets, and strings). In practice, many important data fall into this category (or the application semantics allow for the definition of a cartesian distance function over the state space). For every pair of states $u, v \in S_{DB}$ (a cartesian database state

space), we define dist(u, v) as the distance between u and v. The distance between a potentially inconsistent state and a known consistent state is called *fuzziness*.

In ESR, each (epsilon) transaction has a specification of the fuzziness allowed in its execution, called *epsilon specification*, or ϵ-spec for short. When ϵ-spec $= 0$, an ET reduces to a classic transaction and ESR reduces to SR. As in previous ESR papers, we adopt a restricted model of ESR where update ETs are serializable with each other while query ETs need not be serializable with update ETs. Thus, fuzziness is allowed only between update ETs (denoted by U^{ET}) and read-only query ETs (denoted by Q^{ET}). The ϵ-spec of a U^{ET} refers to the amount of fuzziness allowed to be exported by the U^{ET}, while the ϵ-spec of a Q^{ET} refers to the amount of fuzziness allowed to be imported by the Q^{ET}.

Each Q^{ET} has an *import_time_limit* which specifies the maximum length of time fuzziness the Q^{ET} can accumulate. Similarly, each U^{ET} has an *export_time_limit* which specifies the maximum amount of time inconsistency the U^{ET} can export to other Q^{ET}s. (For value fuzziness the respective limits are *import_value_limit* and *export_value_limit*.) The fuzziness limits can be specified by the application designers for each ET, and they can also be changed during run-time.

The time interval $[ts(T_i) - \epsilon\text{-spec}, ts(T_i) + \epsilon\text{-spec}]$ defines all the legitimate versions accessible by T_i, where $ts(T_i)$ denotes the timestamp of (epsilon) transaction T_i. Namely, as long as the distance between timestamps is within T_i's ϵ-spec, a version can be accessed by T_i. (Note that, unless for the purpose of specific contrast, for the rest of the paper we may omit *epsilon* when referring to an epsilon transaction in mvESR.) Several criteria could be used for the choice of versions within the interval $[ts(T_i) - \epsilon\text{-spec}, ts(T_i) + \epsilon\text{-spec}]$, but it is not our intention to decide here the policy issues involved. We will discuss some concrete examples in the next section.

3.2 Calculating time fuzziness

In a multiversion database, we model the different versions of a data item x as separate data objects in the database. Let $ts(x_i)$ be the timestamp of data object x_i. Such a $ts(x_i)$ may determine the serialization order of transaction T_i that created the object. Each data object has a value and an index denoting the transaction number that created the object. Note that, when we discuss *time* in this paper, we refer to the timestamp of data versions. However, our results also apply to time as part of the database state space (e.g., account creation date in a bank database). In this paper, we focus on the fuzziness in metadata, i.e., the timestamp in data versions.

Informally, the time fuzziness for the non-SR execution of a query operation is defined as the distance in time between the version a query ET reads in an mvESR execution and the version that would have been read in an mvSR execution. Concretely, consider α and β as two time points (and $\alpha \leq \beta$ unless otherwise stated). Let us adopt the notation:

- $[\alpha, \beta]$, as the time interval between α and β;
- $\overline{[\alpha, \beta]}$ as the length of time between α and β; and
- $\text{dist}_t(\alpha, \beta) = \overline{[\alpha, \beta]}$, as the distance in time between α and β.

The problem with accumulating time fuzziness is that a simple summation of two time intervals may underestimate or overestimate the total time fuzziness. For example, if two time intervals $[\alpha, \beta]$ and $[\gamma, \delta]$ are completely disjoint (w.l.o.g. assuming $\alpha < \beta < \gamma < \delta$), we have

$$\overline{[\alpha, \delta]} > \overline{[\alpha, \beta]} + \overline{[\gamma, \delta]}; \text{ or}$$

$$\text{dist}_t(\alpha, \delta) > \text{dist}_t(\alpha, \beta) + \text{dist}_t(\gamma, \delta). \quad (1)$$

Namely, a simple summation of the lengths of two disjoint intervals underestimates the total time fuzziness. On the other hand, if two intervals overlap, then a summation will overestimate the total fuzziness. For instance, if $\alpha < \gamma < \delta < \beta$, then the simple summation of $\overline{[\alpha, \beta]}$ and $\overline{[\gamma, \delta]}$ would overestimate the total time fuzziness. Therefore, we need a different accumulation operator for time fuzziness management.

In this paper, we present an accurate time interval accumulation operator, called TimeUnion, which operates on time intervals instead of their lengths. For two time intervals $[\alpha, \beta]$ and $[\gamma, \delta]$ we define:

$$\text{TimeUnion}([\alpha, \beta], [\gamma, \delta]) = [min(\alpha, \gamma), max(\beta, \delta)].$$

An alternative and equivalent definition is:

$$\text{TimeUnion}([\alpha, \beta], [\gamma, \delta]) = [\mu, \nu] \text{ such that}$$

$$\overline{[\mu, \nu]} \text{ is maximal } \forall \mu \in [\alpha, \beta], \nu \in [\gamma, \delta].$$

The first definition is intuitive while the second definition makes explicit that the TimeUnion operator will not underestimate or overestimate time fuzziness when used as an accumulator. As an example, TimeUnion$([2, 5], [9, 10]) = [2, 10]$. Thus, for time intervals $[\alpha, \beta]$ and $[\gamma, \delta]$,

$$\overline{\text{TimeUnion}([\alpha, \beta], [\gamma, \delta])} = \overline{[\alpha, \delta]}, \forall \alpha \leq \beta, \gamma \leq \delta. \quad (2)$$

Thus, TimeUnion over a set of intervals returns the smallest interval that covers all of them, i.e., its lower bound is the minimum of all the lower bounds and its upper bound is the maximum of all the upper bounds. As a result, if time fuzziness between α and β is represented as $\text{dist}_t(\alpha, \beta) = \overline{[\alpha, \beta]}$, then the above TimeUnion operator can be used to accurately accumulate the total time fuzziness of different intervals by following Eq. 2.

3.3 Bounding time fuzziness

The objective of an mvDC algorithm is to keep the time inconsistency accumulated by each ET below its ϵ-spec. Let $import\_time\_fuzziness_{Q_i}$ be the accumulated amount of time fuzziness that has been imported by a query Q_i. Similarly, let $export\_time\_fuzziness_{U_j}$ be the accumulated time fuzziness exported to other query ETs by U_j. The objective of an mvDC algorithm is to maintain the following invariants for all Q_i and U_j:

- $import\_time\_fuzziness_{Q_i} \leq import\_time\_limit_{Q_i}$;
- $export\_time\_fuzziness_{U_j} \leq export\_time\_limit_{U_j}$.

We call these the $Safe_{time}$ conditions for an mvESR schedule.

When a query Q_i accepts a data item x_i as input in an mvESR execution, x_i may render the execution non-SR. However, this can be allowed if the distance between x_i and a serializable version does not make $import\_time\_fuzziness_{Q_i}$ exceed $import\_time\_limit_{Q_i}$. Conversely, time fuzziness can be exported from an update ET when other query ETs accept a non-SR version created by the update ET. Therefore, the time fuzziness also has to be accumulated for the update ET as well.

4 Version selection

In most classic multiversion CC algorithms, it is straightforward to determine which version should be accessed given a transaction T_i's timestamp, $ts(T_i)$. Usually, it is the version created before $ts(T_i)$ but with the closest timestamp to $ts(T_i)$. This choice ensures the serializability of T_i. In contrast, mvDC does not have to ensure serializability when ϵ-spec > 0. Therefore, mvDC may choose from a potentially larger number of versions for each transaction access.

The function that chooses a version given a transaction's timestamp and ϵ-spec is called *PickV*. There are several trade-offs involved in the choice of an appropriate version for a given situation. Detailed analysis and simulation may be needed for a systematic exploration of the design space, policy parameters, and application environments. However, for the description of mvDC algorithms, the specific details of the version selection function is not important. Thus, we only suggest some possibilities here and analyze their advantages and disadvantages in a qualitative way.

The first and simple example ($PickV_1$) is to assume that ϵ-spec $= 0$ and choose the version that makes T_i serializable. This example shows that the *PickV* function can be a generalization of the version selection function of a classic multiversion CC algorithm. The second example ($PickV_2$), ignores ϵ-spec and always chooses the most recently committed version of the data item being accessed. This policy emulates a single version database with private workspaces for update transactions. Note that mv2PL algorithms work this way, but timestamp based algorithms need to be combined with some other mechanism.

The third example ($PickV_3$) tries to minimize the fuzziness by approximating the classic mvCC. It chooses the newest version x_j of data item x such that $ts(x_j) \leq ts(T_i)$. If such an x_j is no longer available because it has been garbage collected, *PickV* can simply returns an available version x_k such that $ts(x_j) < ts(x_k) < (ts(T_i) + \epsilon\text{-spec})$. On the other hand, if x_j is not available for any other reason (e.g., a disk error), $PickV_3$ may return any version x_m such that $ts(x_m) < ts(x_j)$ and $(ts(T_i) - \epsilon\text{-spec}) < ts(x_m) < ts(T_i)$.

The fourth example ($PickV_4$) takes into account the recency (birthline) and urgency (deadline) requirements of a real-time query T_i. Regardless of $ts(T_i)$, $PickV_4$ could return a version that is near the midpoint between T_i's birthline and deadline. This policy effectively redefines the serialization order of T_i, moving it from its timestamp to the midpoint. This is a heuristic to approximate the likely CPU scheduling order of T_i. If there is more information about how T_i's CPU requirements are being scheduled, then $PickV_4$ could choose the versions that are close to its CPU run time. Of course, these choices all assume they are within the interval $[ts(T_i) - \epsilon\text{-spec}, ts(T_i) + \epsilon\text{-spec}]$.

5 Timestamp ordering mvDC for time fuzziness

In this section, we present a timestamp ordering mvDC algorithm with an emphasis on bounding time fuzziness. Namely, we assume that the ϵ-spec's of other dimensions are infinitely large. In Section 7, we will describe a 2PL mvDC algorithm that bounds both time fuzziness and value fuzziness.

Analogous to the divergency control algorithms which support ESR in a single version environment, we now describe the mvDC method to guarantee mvESR for a timestamp ordering multiversion concurrency control. We will extend the methodology described in [17] for 1vDC algorithms and modify a classical mvCC to produce the mvDC that supports mvESR. Our objective is to provide a systematic methodology to extend an existing mvCC protocol by relaxing the correctness criteria from mvSR to mvESR. Despite the differences in implementing various mvDC algorithms for different types of mvCC protocols, the key aspects of mvDC common to all algorithms are: (1) the database is always kept in a consistent state, (2) non-mvSR operations may be allowed for ET's, and (3) the import or export fuzziness of an ET is bounded by the mvDC algorithm according to the ET's ϵ-spec.

Typically, a classical mvCC algorithm implicitly assumes that an unrestricted number of versions can be accessed. However, in practice most of them are refined to operate on a finite number of versions. In such a case, if a data version needed for a serializable execution has already been garbage collected, the corresponding transaction will be aborted in an mvCC algorithm. In an mvDC algorithm, however, we can allow a query ET to read a non-SR data version with the amount of inconsistency bounded within its ϵ-spec. Only when the inconsistency grows beyond the ϵ-spec do we then either abort the ET (in a timestamp-based or optimistic mvDC) or block its execution (in a lock-based mvDC), just like a traditional mvCC algorithm.

5.1 Summary of a classic timestamp ordering mvCC algorithm

Classical timestamp ordering concurrency control guarantees an SR execution of transactions by choosing a version that makes the transaction serializable. Here we summarize a multiversion timestamp ordering (mvTO) algorithm in [3] as an example. In such an mvTO CC algorithm, each transaction is assigned a unique timestamp, denoted as $ts(T_i)$. Each operation carries the timestamp of its corresponding transaction. Each version is labeled by the timestamp of the transaction that updated it.

This mvTO CC algorithm maintains SR by translating a read operation $r_i(x)$ into $r_i(x_k)$, where $ts(x_k)$ is the largest timestamp less than or equal to $ts(T_i)$, and then reads x_k. It processes a write operation $w_i(x)$ by considering two cases. If the database system has already processsed a read $r_j(x_k)$ for another transaction T_j such that $ts(x_k) < ts(T_i) < ts(T_j)$, then it rejects $w_i(x)$ and aborts or restarts T_i. Otherwise, $w_i(x)$ creates a new version x_i. To ensure recoverability, any transaction T_j that reads a version created by another transaction T_i must wait and commit after T_i has been committed.

5.2 Design of a timestamp ordering mvDC algorithm

To eliminate the complexity of maintaining an unrestricted number of data versions and the expense of high storage overhead, a finite number of data versions are assumed for each data item in the description of the timestamp ordering mvDC algorithm. Moreover, we assumed that some *PickV* function is used to select a version for a query ET read operation. However, specific details about the selection function is not critical to the design of the mvDC algorithms. In fact, mvDC algorithms provide more room for the *PickV* function to choose an appropriate version.

Extension stage

The extension stage of an mvDC algorithm identifies the conditions where a non-SR operation may be allowed. For

a timestamp ordering mvDC algorithm, there are two such conditions. First, a query Q_i may access a non-SR data version by the $PickV$ function if the serializable version does not exist or if it cannot satisfy the recency requirement. For example, using $PickV_4$ described in Section 4 to select a version may allow Q_i to read a non-SR but more recent version for a real-time application. Secondly, a late arriving update ET may still create a new version that is non-SR with respect to some previously arrived query ETs.

Note that, for a query ET, a read operation $q_i(x)$ from a query Q_i can be processed by choosing a proper version x_k (e.g., by $PickV_4$). However, for an update ET, a read operation $r_j(x)$ from an update ET U_j is restricted to a serializable data version in order to maintain the overall database consistency. As a result, only for query read operations can the different version selection functions described in Section 4 be used. For the read operations of an update ET, the system must always return the serializable version.

For a late arriving write operation $w_i(x)$ from an update ET U_i, where there exists an x_k that has been read by another query Q_j such that $ts(x_k) < ts(U_i) < ts(Q_j)$, we may still allow $w_i(x)$ to proceed and create a new version x_i. However, this non-SR version x_i must be "close enough" to the serialization timestamp. In other words, the difference between the timestamp of the non-SR version and the serialization timestamp must be within the time fuzziness ϵ-spec.

Relaxation stage

In the relaxation stage, we calculate and bound the inconsistency resulted from the non-SR executions. On a query read $q_i(x)$ from query Q_i, it is translated into $q_i(x_j)$, where x_j is chosen based on a selection function (see Section 4). Suppose x_k is the serializable version. Then, the time fuzziness caused by allowing Q_i to read x_j instead of x_k for Q_i is the time distance between $ts(x_j)$ and $ts(x_k)$. Namely, $import\_time\_fuzziness_{Q_i,x} = export\_time\_fuzziness_{U_j,x} = \text{dist}_t(ts(U_k), ts(U_j))$. Here, $import\_time\_fuzziness_{Q_i,x}$ represents the imported time fuzziness of Q_i due to x and $export\_time\_fuzziness_{U_j,x}$ is the exported time fuzziness through x. This time fuzziness is accumulated using the TimeUnion operation described in Section 3.2 for Q_i and U_j. If the resulting accumulated time fuzziness does not exceed the corresponding ϵ-spec, then the non-SR operation can be allowed. Otherwise, it has to be rejected.

On the other hand, the time fuzziness incurred by a write operation $w_i(x)$ (of U_i) being processed even after a $q_j(x_k)$ (from Q_j) with $ts(x_k) < ts(U_i) < ts(Q_j)$ has been completed, is the time distance between $ts(U_k)$ and $ts(U_i)$, where U_k is the update ET that has created x_k. This time fuzziness is accumulated into U_i and Q_j's time fuzziness using the TimeUnion operation. Only if the resulting fuzziness does not exceed the corresponding ϵ-spec for both U_i and Q_j is the late arriving write operation allowed to proceed. Note that there may be multiple such query ET's that has read a version of x before the late arriving update operation $w_i(x)$ arrives. In such a case, the time fuzziness has to be accumulated for all the query ETs.

5.3 Proof of correctness

We first identify the cycle prevention properties for the corresponding multiversion serialization graph of the original multiversion timestamp ordering algorithm [3]. Following these properties, we show that the $Safe_{time}$ condition is hold for any violation of the properties. Thus, database consistency and bounded inconsistency for ETs are guaranteed.

In [3], every successful multiversion timestamp ordering execution history H has the following properties to guarantee its serializability [3]:

1. For each T_i, $ts(T_i) = ts(T_j)$ iff $i = j$, i.e., a unique timestamp for each transaction;
2. For every $r_k(x_j) \in H$, $w_j(x_j) < r_k(x_j)$, and $ts(T_j) \leq ts(T_k)$, i.e., each transaction T_k only reads data objects with timestamps smaller than $ts(T_k)$;
3. For every $r_k(x_j)$ and $w_i(x_i) \in H$, $i \neq j$, either
   a) $ts(T_i) < ts(T_j)$ or
   b) $ts(T_k) < ts(T_i)$ or
   c) $i = k$ and $r_k(x_j) < w_i(x_i)$,

   i.e., when the scheduler processes $r_k(x_j)$, x_j is the version with the largest timestamp that is less than or equal to $ts(T_k)$;
4. If $r_j(x_i) \in H$, $i \neq j$, and $c_j \in H$, then $c_i < c_j$, where c_j represents commit of transaction T_j (recoverability).

In a timestamp ordering mvDC algorithm, Rule 3 may be violated if the $PickV$ function returns a non-serializable data version for a query ET, or if a late arriving write operation $w_i(x)$ is allowed even if a $q_k(x_j)$ with $ts(x_j) < ts(U_i) < ts(Q_k)$ has been processed. However, for these cases the time inconsistency is accumulated for the involved query and update ETs when the conflict is detected. For rule 4, we do not delay query ET to commit until after the update ET is committed. In this case, the data versions that an already committed query ET has read may be discarded later on. However, the time inconsistency has already been taken into account for the query ET. As a result, the $Safe_{time}$ condition is ensured in all cases and the resulting schedule is guaranteed ESR. □

6 Bounding both time and value fuzziness

The calculation of value fuzziness in mvDC is similar to 1vDC, where the value fuzziness of each non-SR conflict is estimated by calculating the distance between the old value and the new value. While the accumulation of time fuzziness requires a special TimeUnion operator (see Section 3.2), the accumulation of value fuzziness for an ET can be accomplished simply by adding together the value fuzziness caused by each non-SR operation.

Similar to the $Safe_{time}$ conditions previously presented in Section 3.3 for time fuzziness only, to bound both value and time fuzziness, an mvDC algorithm must maintain at all time the following $Safe_{time,value}$ conditions for every Q_i and U_j:

- $import\_value\_fuzziness_{Q_i} < import\_value\_limit_{Q_i}$;
- $import\_time\_fuzziness_{Q_i} < import\_time\_limit_{Q_i}$;
- $export\_value\_fuzziness_{U_j} < export\_value\_limit_{U_j}$;
- $export\_time\_fuzziness_{U_j} < export\_time\_limit_{U_j}$.

In the above conditions, $import\_value\_fuzziness_{Q_i}$ represents the accumulated value fuzziness imported by Q_i, and $export\_value\_fuzziness_{U_j}$ represents the accumulated value fuzziness exported by U_j. For a non-SR operation to be processed, all these conditions must be satisfied. If any of them does not hold, the non-SR operation cannot be allowed.

In mvDC, in addition to time fuzziness, value fuzziness is also introduced by choosing a non-SR version (e.g., by $PickV_3$ or $PickV_4$) and it can be calculated by taking the difference between values from the non-SR version and the SR version. Note that, in a non-SR environment, value fuzziness and time fuzziness are always present. Given two versions, value fuzziness is the difference between their values and time fuzziness is the distance between their version timestamps.

Consider the following example. Let operation $op_{i,j}(x_k)$ be the jth operation of transaction i on data item x_k. Assume that the following schedule is an example of 1vDC.

$$w_{0,1}(x)w_{0,2}(y)c_0r_{1,1}(x)w_{2,1}(y)r_{1,2}(y)c_1w_{2,2}(x)c_2. \quad (3)$$

In a multiversion environment where each $w_i(x_i)$ creates a version x_i, the above 1vDC schedule may become the following mvDC schedule:

$$w_{0,1}(x_0)w_{0,2}(y_0)c_0r_{1,1}(x_0)w_{2,1}(y_2)r_{1,2}(y_2)c_1w_{2,2}(x_2)c_2. \quad (4)$$

In Eq. 4, both T_0 and T_1 are supposed to be serialized before T_2, but T_1 reads T_2's output with $r_{1,2}(y_2)$. Thus, the corresponding value fuzziness and time fuzziness due to $r_{1,2}(y_2)$ are increased by $dist_v(y_0, y_2)$ and $dist_t(ts(T_0), ts(T_2))$, respectively.

In some applications, the calculation of value inconsistency can be further simplified. If the data change of an application is generally small enough to be negligible without affecting the system, or if there is a relatively constant rate of change of data, we can calculate the value distance simply using the timestamps of data versions without actually accessing their values. For example, radar reports of civilian airplane locations usually have their value changes bounded by the elapse time, and the value differences among data objects could be estimated accordingly.

In the following section, we describe a 2PL mvDC algorithm that bounds both value fuzziness and time fuzziness.

7 Two-phase locking mvDC for both time and value fuzziness

7.1 Summary of a classic 2PL mvCC algorithm

In this 2PL mvCC [3], there are two phases of transaction execution: the locking phase and the unlocking phase. During the locking phase a transaction must obtain all the locks it requests. The moment when all the requested locks are granted, which is equivalent to the end of the locking phase and the beginning of the unlocking phase, is called the *lock point* of a transaction.

Each transaction and each data item in the database is initially uncertified. Any write operation $w(x)$ prepares a new and uncertified version of data item x. When a transaction read data item x, the scheduler attempts to set a read lock on x before it can read it. At the end of execution, the transaction and the new versions it has prepared, will be certified. Upon certification of transaction T_i and all the data versions it has created, we set certify locks on all these data items. These locks are governed by the compatibility matrix in Table 1, where Rl^T, Wl^T and Cl^T represent the read lock, write lock and certify lock, respectively. Note that, in Table 1, columns represent locks held while rows represent locks requested. Also, a write lock is compatible with any other locks for multiple versions, and attempt to set write lock on data object can be rejected only when the memory management protocol fails to create new data versions.

	Rl^T	Wl^T	Cl^T
Rl^T	Aok	Aok	Nok
Wl^T	Aok	Aok	Aok
Cl^T	Nok	Aok	Nok

Table 1: Compatibility matrix for a 2PL mvCC.

Data versions and transaction T_i are marked certified if all certify locks are successfully obtained and the following conditions are satisfied:

- At the moment of transaction T_i's certification, the versions of all data items read by T_i are certified;
- For each data item x that T_i wrote, all transactions that has read a certified versions of x are certified.

In order to satisfy the first condition, a *certify token* can be allocated to each data item so that all the read operations can only read the latest certified version of the data item. The second condition can be maintained by delaying the certification of an update transaction until after other transactions that have read the certified version of the same data that it updated become certified.

7.2 Design of a 2PL mvDC algorithm

Extension stage

To process a query read operation, we again invoke a version selection function $PickV$ (see Section 4) to choose a version of the data, certified or uncertified, to be read. However, for a read operation from an update ET, it has to read the data item that holds the *certify token* as in the classical algorithm in order to guarantee the final database consistency. In order to allow a query ET to read uncommitted data, an update ET can prepare a new version of a data item as soon as the Wl lock is acquired without waiting until its commit point. The locking compatibility matrix for the 2PL mvDC is shown in Table 2, where Ql^{ET} represents the read lock by a query ET and Rl^{ET} represents the read lock by an update ET.

When an ET successfully sets all the locks and ready to certify, the two corresponding conditions required for certification are extended as follow:

- Any data version returned by the function $PickV$ may be accepted for a query read, but the data version for an update ET is restricted to the latest certified one;
- For each data item x that U_i wrote, all the update ET's that have read a certified data version of x have to be certified before U_i can be certified.

	Ql^{ET}	Rl^{ET}	Wl^{ET}	Cl^{ET}
Ql^{ET}	Aok	Aok	Aok	Lok-1
Rl^{ET}	Aok	Aok	Aok	Nok
Wl^{ET}	Aok	Aok	Aok	Aok
Cl^{ET}	Lok-2	Nok	Aok	Nok

Table 2: Compatibility Matrix for 2PL mvDC.

Note that even though there is a limitation on storage, at least the latest certified version of the data item must be kept in order to ensure consistency. For data items which is frequently updated, we may try to obtain more memory for them by negotiating with other less frequently updated data items. If memory space is limited, write operations may be blocked until more space is available. Moreover, we do not allow an update ET to import any inconsistency from other ETs. Therefore, when the certified version that is serializable to an update ET is not available, the update ET should be aborted immediately without further considering the inconsistency it acquires. Finally in the relaxation stage, we calculate and bound the inconsistency arising from the conflicting operation for query and update ETs to ensure the $Safe_{time,value}$ conditions described in Section 6.

Relaxation stage

We relax the lock management at Lok-1 and Lok-2 for the 2PL mvDC (see Table 2) by calculating and restricting both the time and value inconsistency acquired in non-mvSR conflicting operations. We associate with each query ET an *import_time_fuzziness* and *import_value_fuzziness*, and each update ET with an *export_time_fuzziness* and *export_value_fuzziness*. Each uncertified data version, when it is created by a write operation, is associated with a *Conflict_Q* list which is initialized to nil. The *Conflict_Q* list is used to remember all queries that have read the uncertified data version, and we calculate the time and value fuzziness for these queries by the time this data version is certified. For each data item x, we define a function $certify\_token(x)$ to return the latest certified version of data item x. Now let us look at the details for each operation.

Read operations of a query ET

- If query Q_i tries to read a certified data version x_j prepared by U_j through Lok-1, the time inconsistency and value inconsistency to be accumulated for Q_i are defined as follows:

$$import\_time\_fuzziness_{Q_i,x} =$$

$$\text{dist}_t(ts(U_j), ts(certify\_token(x)));$$

$$import\_value\_fuzziness_{Q_i,x} =$$

$$\text{dist}_v(x_j, certify\_token(x)).$$

The same time fuzziness is accumulated to the existing time fuzziness of both Q_i and U_j using the TimeUnion operator described in Section 3.2. Similarly, the value inconsistency of Q_i and that of U_j are also increased by the value difference between versions x_j and *certify_token(x)*. The Lok-1 request can be allowed only if the resulting fuzziness does not exceed its corresponding ϵ-spec. Namely, the $Safe_{time,value}$ conditions must hold true. Otherwise, the Lok-1 lock request cannot be granted.

- On the other hand, if a query is allowed to read an uncertified data version x_k through Lok-1, then Q_i is first added to the $Conflict\_Q$ list of x_k, and the calculation of time and value fuzziness is delayed until the data version is certified.

Note that, for applications which cannot afford to delay the certification, one can use external information to estimate the timestamp and value of the data version that is serializable to it, and then accumulate the fuzziness to the total time and value fuzziness of the corresponding ETs. If it exceeds any of the bounds, specified in $Safe_{time,value}$, we either abort Q_i or re-invoke the $PickV$ function for another version of the data item. Otherwise, the query is allowed to be certified. Note that when an estimation is wrong and inconsistencies go beyond the bound, a query may need to be rolled back at the time when data versions are certified.

Read operations of an update ET

In order to maintain the final database consistency, the read operations of an update ET are restricted to see consistent data objects and hence the latest certified versions must be read.

Certification of ETs

To certify a query ET, we check the time and value inconsistency incurred to decide whether to certify or abort it. To certify an update ET through Lok-2, the process is a little more complicated. For an update U_i, when all the certify locks are successfully set on data items, we further check the following conditions before certifying it:

- For each data item that U_i wrote, if there is a read operation from U_j which accessed certified version of the data item, we delay the certification until after U_j is certified.
- For each data item x_i that U_i wrote, the time and value inconsistency of any query Q_j in $Conflict\_Q$ of x_i that has read the uncertified version x_i, AND any uncertified (active) Q_k that has read a certified version of x before x_i is created, are increased by the following respective amounts:

$$\text{dist}_t(ts(certify\_token(x)), ts(x_i));$$

$$\text{dist}_v(certify\_token(x), x_i).$$

If the $Safe_{time,value}$ conditions are all held, then Lok-2 can be granted and U_i can be certified.

- After Lok-2 is granted, we can discard the $Conflict\_Q$ of each data object, pass the *certify token* and certify all the versions U_i has created. Then we certify U_i.

8 Conclusions

In this paper, two multiversion divergence control algorithms were described for epsilon serializability in multiversion databases to allow for more concurrency by tolerating bounded inconsistency. The inconsistency tolerance were divided into two dimensions: value fuzziness and time fuzziness. To correctly control the total time fuzziness, a new operator called TimeUnion, was introduced. Because of the accurate control of both time and value fuzziness, multiversion ESR represents a very useful solution to real-time transactions that may tolerate some degree of data inconsistency in order to gain more freedom in version selection and scheduling. With mvESR, a real-time application designer can control the data inconsistency and data timeliness independently.

The design of multiversion divergence control algorithms were built on an existing single version ESR framework. The existing design methodology for single version divergence control was used to derive new multiversion divergence control algorithms from the corresponding multiversion concurrency control algorithms. First, a certain non-serializable operations are identified for possible relaxation. Then, the time fuzziness and/or value fuzziness due to the non-serializable operations are accurately accumulated for the corresponding query and update epsilon transactions. If the accumulated fuzziness does not exceed its specified limit, then a non-serializable operation is allowed to proceed. Otherwise, it is rejected.

Finally, the multiversion divergence control algorithms described in this paper were not combined with any CPU scheduling algorithms such as earliest-deadline-first. However, multiversion divergence control algorithms do provide more room for CPU scheduling algorithms, since real-time transactions can run unimpeded within their inconsistency tolerance. Moreover, they also allow more room for version selection functions to pick the appropriate versions because of the carefully controlled relaxation in both value and time fuzziness.

References

[1] D. Agrawal and S. Sengupta. Modular synchronization in multiversion databases: Version control and concurrency control. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 408–417, 1989.

[2] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Trans. on Database Systems*, 8(4):465–483, Dec. 1983.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[4] P. M. Bober and M. J. Carey. On mixing queries and transactions via multiversion locking. In *Proc. of Int. Conf. on Data Engineering*, pages 535–545, 1992.

[5] M. J. Carey and W. A. Muhanna. The performance of multiversion concurrency control algorithms. *ACM Trans. on Computer Systems*, 4(4):338–378, Nov. 1986.

[6] A. Chan and R. Gray. Implementing distributed read-only transactions. *IEEE Trans. on Software Engineering*, SE-11(2):205–212, Feb. 1985.

[7] T.-W. Kuo and A. K. Mok. Application semantics and concurrency control of real-time data-intensive applications. In *Proc. of Real-Time Systems Symposium*, pages 35–45, 1992.

[8] A. Merchant, K.-L. Wu, P. S. Yu, and M.-S. Chen. Performance analysis of dynamic finite versioning for concurrent transaction and query processing. In *Proc. of 1992 ACM SIGMETRICS and PERFORMANCE '92*, pages 103–114, 1992.

[9] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 124–133, 1992.

[10] C. Pu, W. Hseush, G. E. Kaiser, K.-L. Wu, and P. S. Yu. Distributed divergence control for epsilon serializability. In *Proc. of Int. Conf. on Distributed Computing Systems*, pages 449–456, 1993.

[11] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 377–386, 1991.

[12] K. Ramamrithan and C. Pu. A formal characterization of epsilon-serializability. Technical Report CUCS-044-91, Department of Computer Science, Columbia University, Dec. 1991.

[13] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Trans. on Computer Systems*, 1(1):3–23, Feb. 1983.

[14] S. H. Son and S. Kouloumbis. Replication control for distributed real-time database systems. In *Proc. of Int. Conf. on Distributed Computing Systems*, pages 144–151, 1992.

[15] W. E. Weihl. Distributed version management for read-only actions. *IEEE Trans. on Software Engineering*, SE-13(1):55–64, Jan. 1987.

[16] K.-L. Wu, P. S. Yu, and M.-S. Chen. Dynamic finite versioning: An effective versioning approach to concurrent transaction and query processing. In *Proc. of Int. Conf. on Data Engineering*, pages 577–586, 1993.

[17] K.-L. Wu, P. S. Yu, and C. Pu. Divergence control for epsilon-serializability. In *Proc. of Int. Conf. on Data Engineering*, pages 506–515, 1992.