Asynchronous Consistency Restoration under Epsilon Serializability

Pamela Drew[1]
Dept. of Computer Science
Hong Kong University of
Science and Technology
Clear Water Bay, Kowloon, Hong Kong
email: pam@cs.ust.hk

Calton Pu
Dept. of Computer Science and Engineering
Oregon Graduate Institute
19600 N.W. von Neumann Dr.
Beaverton, OR 97006-1999
email: calton@cse.ogi.edu

Abstract

Many database applications tolerate a certain amount of data inconsistency to gain increased concurrent processing and to accommodate real-world constraints. This paper describes how inconsistency can be modeled in a database and managed with consistency restoration methods. The correctness criterion for the maintenance of inconsistency is epsilon serializability (ESR). An informal notation to characterize inconsistency and several consistency restoration techniques are described.

Index terms: epsilon-serializability, divergence control, serializability, crash recovery, consistency restoration, asynchronous transaction processing.

[1] This research was supported by the Sino Software Research Center, contract number SSRC92/93.006.

Contents

1 Introduction

Asynchronous transaction processing [20] alleviates the limitations of serializability [3, 16] in terms of performance (increased level of concurrency) and availability. A convenient foundation for asynchronous transaction processing is *epsilon serializability* (ESR) [21, 18]. ESR allows for limited inconsistency in transaction processing (TP), specified by application designers using epsilon-transactions (ETs).

Prior work on ESR [25, 17] focussed on divergence control (DC) methods to manage inconsistency between ETs that are either read-only (Q^{ET}) or serializable updates (U^{ET}). In this case, Q^{ET}s can execute concurrently, within certain limits, with U^{ET}s. Inconsistency can be imported into the Q^{ET}s or exported from U^{ET}s. The divergence control methods ensure that inconsistency is not committed into the permanent database state. Other ESR papers [21, 18, 19, 20] also make the same assumption.

This paper addresses the considerably different case of general ETs (G^{ET}) that both import and export inconsistency. These transactions can, to a certain extent, read uncommitted changes of other transactions and make their changes visible to other transactions before they commit. Therefore, G^{ET}s may leave inconsistent data in the database. For this scenario, we need *consistency restoration* (CR) methods to repair the inconsistency introduced by G^{ET}. In contrast to divergence control methods that allow only temporary inconsistency and prevent permanent inconsistency by eliminating G^{ET}, CR methods allow bounded and repairable inconsistency to occur.

It is important to understand that this case is a significant departure from a classic TP framework and our previous work. Traditionally, a transaction is defined as a sequence of operations that transform a consistent database state into another consistent state. Inconsistent database states are outside the classic transaction model. Our contribution is using ESR to introduce the management of bounded inconsistency into databases, including an informal notation and some consistency restoration algorithms.

The practical importance of our approach is underscored by the many applications that handle inconsistency themselves. For example, scientific databases may have an error bar associated with a particular value. Banks maintain escrow accounts of deposits and transfers done by mistake. The association between the value and its error bar, however, is known only implicitly to the application. In proposing to manage the inconsistency by DBMS, we are trying to take some of this burden away from the applications, making the database inconsistency management more uniform and explicit.

In Section 2 we define inconsistency in databases in the context of ESR. Section 3, then, summarizes how inconsistency can be propagated between applications in an ESR environment.

We outline some tradeoffs in the invocation of consistency restoration in Section 4. We then, in Section 5, describe concrete, representative instances of consistency restoration methods based on compensations and independent updates. Before the conclusions in Section 7, we discuss related work in Section 6.

2 Consistency in Databases

Traditionally, atomic transactions are programs that preserve database consistency by transforming a consistent database state into another consistent state. Serializability, or other consistency criteria, is used by database management systems (DBMS) to run concurrent transactions in a correct way. In the database, data value consistency can be enforced with mechanisms such as integrity constraints. Of course, a primary objective of transaction management is to control the execution of transactions to ensure consistent updates to the permanent database state.

When inconsistency is allowed in databases, both transaction and database state inconsistency must be considered. In this section, we define consistency criteria for each case. We also introduce notation for representing inconsistency so that we can manage its propagation and reduction.

2.1 Transaction Consistency under Serializability

Transaction consistency is defined as serializability. A database is a set of data items, which support Read and Write operations. Read operations do not change data and Write operations do. (Section 5.2 introduces additional semantics to database operations.) A transaction is a sequence of operations that take a database from a consistent state to another. Transactions may be *updates* that contain at least one Write or *queries* that are read-only.

Our terminology follows the standard model of conflict-based serializability [3]. Two operations are said to conflict if at least one of them is a Write, so we have read-write (R/W) and write-write (W/W) conflicts. Each pair of conflicting operations establishes a *dependency*. A *history*, or *log*, is a sequence of operations such as Reads and Writes. A *serial* log is a sequence of operations composed of consecutive transactions. A log of transaction operations is said to be serializable (SRlog) if it produces results equivalent to some serial log, in which the same transactions execute sequentially, one at a time. *Concurrency control* methods that preserve SR (e.g., two-phase locking) are algorithms that restrict the interleaving of operations in such a way that only SRlogs are allowed.

Intuitively, in the standard model a log is shown to be an SRlog by rearranging its operations according to certain constraints imposed by R/W and W/W dependencies. The rules of rearrangement are given by concrete concurrency control methods. A more formal way to

	ImpLimit = 0	ImpLimit > 0
ExpLimit = 0	Transaction	Q^{ET}
ExpLimit > 0	U^{ET}	G^{ET}

Table 1: Four Kinds of ETs

specify concurrency control uses serialization graph (SG), where each arc represents the *precede* relation [3]. Transaction T_1 precedes T_2 when one of T_1's operations precedes and conflicts (R/W or W/W) with T_2's operations. Since the Serializability Theorem [3] says that a log H is SR if and only if its serialization graph SG(H) is acyclic, an acyclic SG implies an SRlog.

2.2 Inconsistency between Transactions

Informally, inconsistency between transactions is created by some execution order or history that cannot be shown to be equivalent to a serial execution of transactions.

We quantify the inconsistency shared between non-SR transactions by using the concept of an epsilon-transaction, denoted ET. An ET sequence of operations maintain database state consistency when executed atomically. However, an ET also includes a specification of the amount of inconsistency permitted when executed concurrently with other ETs. This per-ET limit of allowed inconsistency is called ϵ-specification, or ϵ-spec for short.

Abstractly, ϵ-spec is divided into two parts, *imported* inconsistency bound and *exported* inconsistency bound. There are many kinds of inconsistency and consequently ϵ-spec may take several forms, as shown in [25] and [2]. In this paper, we use concrete examples for illustration. The methods can be generalized to other inconsistency specifications using techniques described in [25]. For example, in an airline reservation system, the number of seats reserved could be used as a unit of measure. Each ET has two parameters, *ImpLimit* that denotes the maximum number of seats in non-SR conflicts that the ET can import from other ETs and *ExpLimit* that denotes the maximum number seats in non-SR conflicts that the ET can export to other ETs. For simplicity of presentation, our ImpLimit/ExpLimit ϵ-spec combines all sources of inconsistency.

The way in which ETs share inconsistency define which types of transaction processing methods, i.e. divergence control and consistency restoration, need to be employed. ETs are categorized into four types by the way they share inconsistency as shown in Table 1. Transactions are the traditional serializable transactions which share no inconsistent information between them, i.e. when $ImpLimit = 0$ and $ExpLimit = 0$. Query ETs denoted by Q^{ET} are read-only ETs that can import or read inconsistent data, but that do not export or share their result with other transactions. Consistent update ETs denoted by U^{ET} are transactions that can read consistent data (e.g. their ImpLimit = 0) and can export some inconsistency, up to ExpLimit. For ImpLimit > 0 and ExpLimit > 0, a General ET, denoted by G^{ET}, may import and export inconsistency at the same time. In this case the inconsistency in the database may grow unboundedly. The focus of this paper is the management of inconsistency propagation and consistency restoration techniques when G^{ET} is introduced.

A formal characterization of ESR in [21] specifies the following intuitive properties:

- When all *import-limit* and *export-limit* are zero, ESR histories are serializable histories.
- A set of transactions may not have a serializable history, but may satisfy ESR.
- That is, ESR may allow more operation orderings than serializability.

2.3 Database State Consistency

A database is a set of data items. Each data item contains a value. A database state is the set of all data values. A database state space is the set of all possible database states. Database state consistency is defined as the adherence to the database state properties. For purposes of this paper, we will limit our definition of a consistent database state space as one which adheres to a cartesian space such as integers. (The algorithms work for a more general definition of metric spaces [21], but we omit the definition here due to the lack of space.)

Many database state spaces are cartesian spaces, for example, dollar amounts in banking databases and airplane seats in airline reservation systems. The important properties of cartesian spaces (and metric spaces) are three:

- They define a distance function between all pairs of states. For instance, the distance between \$50 and \$120 is \$70.
- The distance function is symmetric. E.g., the distance between \$50 and \$120 is \$70 whether it is a debit or credit.
- The distance function follows triangle inequality. For example, suppose the current account balance is \$50 and \$70 is credited. The distance between the new state and the old state, as we saw before is \$70. Suppose \$40 is now debited. The distance between the state after the credit and the state after the debit is \$40. The distance between the initial state of the account (\$50) and the one after both updates (\$80) is \$30. Since $70+40 \geq 30$, triangle inequality is satisfied.

Usually the term "database state space" refers to the state on disk (implicitly, only the committed values). We are not restricted to the database state on disk, however, since we also consider the intermediate states of the database, including the contents in the main memory. We will use the shorter term "data state" to include the intermediate states. Note that the magnitude of an update can be measured by the distance between the old data item state and the new data item state.

2.4 Data Epsilon Specifications: Inconsistency in the Database

For a DBMS to manage the inconsistency in the database, we need to introduce a notation that quantifies the amount of inconsistency allowed for a particular data value and methods to store this information on disk. In fact, depending on the semantics of inconsistency, different notations may be used. In this paper, we extend and refine the same semantics of previous ESR papers [25, 18, 17].

We assume each value in the database state may include an inconsistency bounded by a fuzziness upper bound, an absolute limit that contains the "correct" value. The upper bound is similar to an error bar, except that the upper bound does not imply any statistical significance either within the bound or outside of bounds. In other words, we only know that the correct value (or several possible correct values) are within the interval delimited by the current value plus and minus the fuzziness upper bound.

Intuitively, database values that do not have any fuzziness continue to be represented by single values. When a value acquires some fuzziness due to external directive (user definition) or the result of a general ET (importing and exporting inconsistency), the DBMS supporting ESR will maintain an upper bound on the amount of fuzziness introduced for each data item. The fuzziness upper bound is then recorded in the database as part of the value.

Once we have defined the notation for capturing the data fuzziness in the database, we can specify the amount of inconsistency allowed on each data object. This data epsilon specification (denoted data-ϵ-spec) of the amount of fuzziness allowed per each data object is analogous to the transaction epsilon specification (denoted trans-ϵ-spec, but originally called simply ϵ-spec) in the ETs. The data-ϵ-spec is defined by the application designer or DBA, telling the DBMS that a particular data object should not have its fuzziness exceed the amount specified. When the actual fuzziness of a data object exceeds its data-ϵ-spec, consistency restoration algorithms (described in Section 5) are activated to reduce the value fuzziness. Or alternatively, the ET causing the additional fuzziness may be aborted.

In the following sections, we discuss the various ways inconsistency is propagated by ETs (the growth of the fuzziness bound) and algorithms for consistency restoration (the reduction of the fuzziness bound).

3 Inconsistency Quantification and Propagation

In this section, we quantify inconsistency between G^{ET}s and estimate the amount of inconsistency passed between them.

3.1 Terminology

Inconsistency in a data item being shared by Q^{ET}s and U^{ET}s can be quantified using the notion of write and read lock intervals [21]. Consider transactions $t_1 \ldots t_n$ where each of the t_i's updates x. A transaction's write lock interval is defined to be the time between when t_i acquires a write lock on x and when t_i releases the write lock on x. A read lock interval is defined similarly. A previous paper [21] shows how the intersection of these intervals from operations of different transactions can be used to put bounds on the amount of inconsistency shared between queries and update transactions.

With the introduction of G^{ET}, the interaction between transactions which execute write operations on the same uncommitted data needs to be quantified. To do this, we use a concrete example based in a multiple reader/single writer lock-based concurrency scheme; note, however, that a similar analysis can be made for other synchronization methods, such as timestamped schemes.

In the general model including G^{ET}, an ET t_i can read a non-permanent, and potentially inconsistent, change to data item x of another ET t_j. This can be implemented in a locking scheme as follows. We assume that no G^{ET} will execute an update on a database value directly that also has a non-permanent representation in memory. In other words, though G^{ET}s can execute concurrently in a possibly inconsistent manner, they update only one copy of the database value. This assumption ensures that we can compare the inconsistency shared between transactions. However, we do not require 2-phase locking, nor, do we require that locks be held until the transaction commits. Further, locks may be acquired and released by different transactions in arbitrary order. For purposes of this discussion, we assume that transactions acquire locks once in their lifetime, however, the following definitions could be extended to account for the release and reacquisition of new locks within transaction boundaries. In this model, transactions still have a write lock interval on x as defined in [21]; previous results regarding the bounds on inconsistency between queries and update transactions remain valid.

Consider G^{ET}s named $t_1 \ldots t_n$ where each of the t_i's can read and write to data item x. In this model, t_i can write to x, release its lock, and t_j can read t_i's result before t_i has committed. In turn, t_j can perform its own computation and write data item x. Let us define an ET's *update* interval with respect to x to be from the time it first acquires a lock on x to the time it makes its changes to x permanent in the database, i.e. commit time. Note that an ET's *update* interval is different from its *write* interval. A *write* interval defines the time the ET holds a lock on x whereas an *update* interval defines the time when an ET has made a non-permanent change to x.

Every G^{ET}, g, has a set of *Concurrent General Transactions* (denoted by $\textsc{cgt}(g)$). For this discussion, we consider U^{ET} to be a particular case of G^{ET}, since a G^{ET} becomes an U^{ET} when its $ImpLimit = 0$. A $t_i \in \textsc{cgt}(g)$ if its write interval intersects with g's update interval. Note that strict 2-phase lock-based realizations of serializability ensure that $\textsc{cgt}(g) = \emptyset$.

3.2 Quantifying Inconsistency in the Database

Inconsistency in a data item x updated by a G^{ET}, g, can be defined informally to be the distance between the value of x after g executes when $\textsc{cgt}(g) = \emptyset$ and the value of x after g executes when $\textsc{cgt}(g)$ is not empty. This distance is bounded by the data item's data-ϵ-spec, which defines the amount of inconsistency tolerable in the data value. Notice that there is no restriction which requires all ETs in $\textsc{cgt}(g)$to have committed; there may be residual inconsistency in a data item value that was written by an ET which later aborted.

Example 1. Consider a banking account aggregate example; an account summary, x, is stored in million-dollar increments. The data-ϵ-spec for a particular account is plus or minus 5 million. Transaction T_1 increments the value of the account by 1 million; Transaction T_2 decrements the account by 4 million. In the G^{ET} model, these transactions can be interleaved as follows:

$$T_1.\text{begin}; T_2.\text{begin}; T_1.\text{updatelock}(x); x := x + 1; T_1.\text{unlock}(x); \tag{1}$$

$$T_2.\text{updatelock}(x); x := x - 4; T_2.\text{unlock}(x); T_1.\text{abort}; T_2.\text{commit}. \tag{2}$$

Given a consistent initial value of $x = 5$, then the resulting value of x is 2. Had T_2 executed in isolation and not read the value written by T_1, the resulting value of x would be 1. Hence, the residual inconsistency in x is 1 million dollars. Since this is within the data item's data-ϵ-spec, the execution is ESR.

Example 2. If, on the other hand, T_1 from Example 1 updates the account summary by 10, the result of x is 11. This value contains more inconsistency than the tolerable level will allow; the distance between the result of T_1 and T_2's concurrent execution and the result of T_2's execution in isolation is 10 (from $11 - 1 = 10$). According to our definition above, the current value must be within data-ϵ-spec of the result of the transaction's execution in a consistent history, e.g., $x = 1$. Since 10 does not fall with the range of 1 ± 5 (the data-ϵ-spec) the execution is not ESR.

One way to implement this model is to store data items as a tuple $\{C, O, E, I\}$ where C is the current value or values of the data item, O is the original or most recently stored consistent value for the item, E is the data-ϵ-spec, and I is current amount of inconsistency associated with the data item. Note that I cannot be computed by taking the difference between the last consistent value and the current value because updates to the current value, C, contain both inconsistency and the actual update values. To pinpoint the inconsistency, it must be calculated during the evaluation of whether the ϵ-specs are being met and then stored separately. The last known consistent value, O, may have been computed by some serializable transaction history or, perhaps, it was the result of an independent update. Notice that whether or not an ET satisfies its trans-ϵ-spec requirement is based on a comparison between the ET and it's $\textsc{cgt}(g)$. The result of the consistent execution of every ET is not stored. O is most likely the result of a ET from some previous time.

3.3 Controlling Inconsistency between ETs

The inconsistency tolerated by a G^{ET} (trans-ϵ-spec) is defined by two components: $ImpLimit$ and $ExpLimit$. These limits are compared to two accumulators ($import - inconsistency$ and $export - inconsistency$) for the ET as it executes. Each time the ET imports some inconsistency, $import - inconsistency$ is incremented. Similar action is taken when inconsistency is exported. When an ET attempts to read and write data item x, the fuzziness of x is accumulated in the ET's inconsistency counters.

Without loss of generality, we use for illustration the case where accumulators are zero so we can compare data inconsistency with trans-ϵ-spec directly. In case the accumulators are not zero, the comparison is done with the remaining tolerance (the difference between the accumulator and the trans-ϵ-spec). In our running example, if T_2 from Example 1 has an $ImpLimit = 3$, then interleaving with T_1 is ESR, because T_1 exports an inconsistency of 1 which is within T_2's $ImpLimit$ of 3. However, in Example 2, T_1 exports an inconsistency of 10 which exceeds T_2's $ImpLimit$ and hence is not ESR.

Similarly, the $ExpLimit$ of a G^{ET} must be checked before the release of its writelocks and its changes are made visible to other transactions. Suppose that T_2 from Example 1 has an $ExpLimit$ of 2. After T_1 concurrently updates the account by 1, T_2's execution exports a total inconsistency of 1 and hence satisfies its $ExpLimit$. If, on the other hand, T_1 updated the account by 3, T_2's $ImpLimit$ is met, but its $ExpLimit$ is exceeded and becomes non-ESR. T_2's $ExpLimit$ restriction is more stringent than the data-ϵ-spec of 5 for the account summary.

3.4 Design of Data and Transaction Epsilon Specifications

A transaction's trans-ϵ-spec and data item's data-ϵ-spec are analogous but independent concepts. On the one hand, each ET may be able to tolerate more or less inconsistency for different applications. ET trans-ϵ-specs are used in the determination of ESR schedules, e.g., in the granting and releasing of locks between G^{ET}. Also, an G^{ET}'s *ExpLimit* determines how much inconsistency it can release into the current ET pool. This propagation between ETs can continue until some ET commits its inconsistency to the database.

The data-ϵ-specs, on the other hand, apply to the data item only and can be designed in an even more independent fashion from application semantics than the ET bounds. The constraints represented by data-ϵ-spec are enforced when a transaction's results are to be written permanently to the database. The data-ϵ-specs will then keep the inconsistency introduced into the permanent database under control.

The design differences between trans-ϵ-spec and data-ϵ-spec is largely determined by the application environment. For ETs that execute simple increment/decrement operations, the *ExpLimit* specification may be only slightly more constraining than, or perhaps even derived from, the data-ϵ-spec. As a plausible scenario, the ET *ExpLimit* can be omitted completely and the transaction can rely completely on data-ϵ-specs for inconsistency control. Other types of transaction with more sophisticated operations such as branching, on the other hand, may have an *ExpLimit* designed very differently from data-ϵ-specs.

4 Invoking Consistency Restoration

Once a constraint is defined, an immediate question is what happens when the constraint is violated. Divergence control algorithms [25, 17] either block or abort an ET when its trans-ϵ-spec is violated. In a DBMS supporting ESR, when data-ϵ-spec is violated, we invoke a consistency restoration algorithm to reduce the data item's fuzziness. In general, we can adopt either an eager policy where data fuzziness is reduced by preventive consistency restoration or a lazy policy where consistency restoration is invoked only when some data-ϵ-spec is violated. The analysis and evaluation of the trade-offs in the different policies is beyond the scope of this paper. We only sketch some possibilities to illustrate the policy choices.

4.1 Violation Prevention

One approach to controlling the growth of data inconsistency is to prevent intolerable fuzziness from occurring. For example, if an ET's *ImpLimit* and *ExpLimit* are about to be exceeded, a preventive method would force the ET to wait until the level of inconsistency is reduced through the execution of a consistency restoration method, or abort. This preventive control on ETs is a divergence control method for G^{ET}s. If implemented in a lock manager, then a check for *ImpLimit* violation before an ET acquired a read lock and a check for *ExpLimit* before the release of writelocks would have the desired effect. If either limit were exceeded, such an ET would be forced to abort or wait until consistency had been restored.

Preventive measures can also be designed to keep the data-ϵ-spec from being violated. In this case, when a G^{ET} tries to commit, we check to see if a data-ϵ-spec will be exceeded by this update. If a G^{ET}, t_i's, update request violates a data item's data-ϵ-spec, then t_i must either abort or be delayed until the data item's inconsistency has been reduced to a level that could incur t_i's additional error. This reduction process could be invoked immediately when a G^{ET} t_i is denied update permission and could be tailored specifically to the amount needed to satisfy t_i's request. Note that a sophisticated divergence control algorithm would recalculate the amount of fuzziness caused by t_i due to the reduced level of inconsistency in the data item.

Note that when a data item's accumulated fuzziness is approaching its data-ϵ-spec, all t_js in CGT(g) may also be denied permission to commit unless their operations reduced the inconsistency in the database. Invoking a consistency restoration method every time a data value approaches its data-ϵ-spec can be expensive since t_i's pending write request may bring the data's inconsistency level close to its data-ϵ-spec again.

Alternatively, a consistency restoration process could be triggered only after a number of G^{ET}s have been denied. This phenomenon would indicate that the data item itself is carrying an amount of inconsistency which is particularly close to its data-ϵ-spec and G^{ET} are being denied update permission even if they would only introduce a small amount of inconsistency in isolation. When a consistency restoration method is invoked on behalf of several ETs, some analysis can be performed to determine how much reduction is needed to execute the pending requests; we leave the design of such optimizations as a topic for future work.

4.2 Detection and Recovery

If trans-ϵ-specs are not enforced before G^{ET}s commit, there is potential for unbounded inconsistency to be introduced into the permanent database. ETs or queries that read these values are said to be unsafe [21]. If this is allowed, then the DBMS must detect the event by checking the final data values against their data-ϵ-specs. If a violation has occurred, the DBMS should invoke a synchronous consistency restoration immediately. This will prevent other ETs or queries from becoming unsafe by reading the violating ET's results. Once an unsafe ET has executed, all of its effects must be undone. This can be carried out by a classical crash recovery method. For data integrity, the locks of the unsafe ET should not be released until the recovery is completed.

5 Consistency Restoration Methods

To restore consistency into the database we use Asynchronous Consistency Restoration (ACR) methods. Just as DC methods reduce to classic concurrency control when trans-ϵ-spec $\rightarrow 0$, ACR methods reduce to classic crash recovery under certain conditions. Consistency restoration methods are invoked to return the permanent database state to one in which the data fuzziness is under their data-ϵ-specs. It is not a requirement of ACR methods to return a database to a completely consistent (e.g. serializable) state and the database need only be within data-ϵ-spec of being consistent. This is in accordance with real world applications where the database is always somewhat inconsistent with the world because of operator errors or data obsolescence. Any DBMS that does not guarantee this level of consistency allows a potentially uncontrollable degeneration of data integrity.

ACR methods can take several forms. Generally speaking, they can be designed as an independent process (from the basic transaction management) in a system. However, in some cases, we can take advantage of the transaction management system to simplify its design. In the following subsections, we give examples of (1) a synchronous consistency restoration based on a classical recovery scheme, (2) some alternative asynchronous schemes based on a model of compensating transactions, and, (3) a consistency restoration method based on independent updates.

5.1 Consistency Restoration Based on Classic Recovery

A model of inconsistency repair based on Read/Write compensations (REDO and UNDO) [10] consists of three steps. First, a specific operation or event is determined to have introduced some inconsistency. In a classic TP environment, these events include site failures and erroneous TP. In a G^{ET} environment, this set of events is extended to include the invocation criteria described in the previous section. Second, the offending operation is undone. Third, operations which had to be undone as a side-effect during the second step must be redone. In a traditional environment, all of the dependent transactions would have to be redone to bring the database back into a consistent state. However, when G^{ET}s are allowed, all dependent transactions may not have to be redone since the database state does not have to be restored to a completely consistent state.

Example 3. To illustrate this method, consider an inventory system in which there are three ETs updating data item y; each of these ETs allow their trans-ϵ-specs to be defined by the data-ϵ-spec. The first, T_1, decrements the value of y by 200. The second, T_2, multiplies the value of y by 10. And the third, T_3, decrements y by 2500. In the G^{ET} model, these ETs can be interleaved as follows:

$$T_1.\text{begin}; T_2.\text{begin}; T_3.\text{begin}; T_1.\text{writelock}(x); T_1 : y = y - 200; \tag{3}$$

$$T_1.\text{unlock}(y); T_2.\text{begin}; T_2.\text{writelock}(y); T_2 : y = y \times 10; \tag{4}$$

$$T_2.unlock(y); T_3.\text{writelock}(y); T_3 : y = y - 2500; T_3.\text{unlock}(y); \tag{5}$$

$$T_1.\text{commit}; T_2.\text{commit}; T_3.\text{commit}. \tag{6}$$

If y has an initial consistent value of 1000 and a data-ϵ-spec of 2500, the accumulated inconsistency would become too high with T_3's update operation. The inconsistency is calculated by the ACR as follows. First, the inconsistency from the concurrent execution of T_1 and T_2 is calculated to be 2000: the distance between the database state if T_2 had executed in isolation ($y = 10,000$) and the database state with its concurrent G^{ET}, T_1 ($y = 8000$). This value is stored by the ACR for future reference. Similarly, the inconsistency of T_3 is determined to be 7000: the absolute value of the distance between the database state if T_3 had executed in isolation ($y = -1500$) and the value of the database with both transactions, T_1 and T_2, in its *CGT* ($y = 5500$).

Suppose that a detect-and-recover strategy has been adopted. Then, some committed transaction must be undone to bring the inconsistency of the system within data-ϵ-specs. The most straightforward selection for a victim is the violating transaction, T_3. In this case, an operation level compensation for T_3 which increments the value of y by 2500 is applied. Assuming the compensation is executed synchronously, the database state is returned to the values produced by T_2.

The selection of T_2 is a more wise choice from a consistency restoration perspective, however, since undoing its effects leaves the database with an inconsistency level markedly lower than when undoing the execution of T_3. If T_2's effects are undone, then the inconsistency for the system is recalculated based on the execution of T_1 and T_3. Such a history leaves the value of ($y = -1700$) and the level of inconsistency at 200: the distance between the absolute value of the database if T_3 had executed in isolation ($y = -1500$) and the value of the database state with the execution of it's *CGT*, T_1 ($y = -1700$). There is, however, more cost associated with the the undoing of T_2 than of T_3 since T_2's operation compensation does not commute [12] with the operations of T_3. To undo the effects of T_2 completely, we must first undo T_2's dependent transaction, T_3, by incrementing the value of y by 2500, dividing the value of y by 10 (the compensation for T_2) and lastly redoing T_3. In the next section, we outline how ESR can be used to reduce this cost further. The tradeoff between the cost of such UNDO/REDO operations and the increased concurrency gained by optimizing the reduction of inconsistency is a topic for future work.

As a last brief point, consider this example under a preventive approach. In this scenario, the appropriate data-ϵ-specs would be checked before updates were committed to the database. The same inconsistency levels would be calculated, however. T_3 would be aborted or delayed until the ACR had reduced the inconsistency by undoing the effects of T_2.

5.2 ESR and Semantics-Based Compensation

Semantics-based compensation transaction management has been proposed as a way to reduce the rollback overhead of sophisticated update operations. Sagas [8] and Compensating transactions [12] are good examples. There are at least two ways that semantic-based compensation strategies can be extended effectively with ESR to allow more concurrency and reduce rollback costs. In one approach, ESR limits can be used to determine whether a compensation is legal. In the second approach, semantics-based compensating transactions can be created specifically to reduce inconsistency in the data state. Before we describe each of these methods in turn, we review the basic concepts of semantics-based compensation.

5.2.1 Definitions

To simplify the presentation we use the notation of Korth et al. [12] to describe Compensating Transactions. This model is developed to allow transactions to share uncommitted updates and and to allow the effects of a committed transaction to be undone without causing cascading aborts or redoing of entire dependent transaction histories. When the updates of transaction T_1 are read by some other transaction T_2, T_1 is said to have been *externalized*. If we want to undo the effects of T_1, a Compensating Transaction CT_1 is run. T_1 is called a *compensated-for* transaction and T_2 a *dependent transaction* with respect to T_1. Dependent transactions, denoted $dep(T)$, define a similar concept to the CGT's in the G^{ET} model.

The goal of this recovery paradigm is to undo the compensated-for transaction, T, by executing a compensating transaction, CT, but leave the effects of the dependent transactions $dep(T)$ intact. An important definition is that of soundness. (As usual [3], a *history* is a sequence of database operations.) If X is the history of transactions T, CT, and their set of dependent transactions $dep(T)$, and Y is some history of only the dependent transactions $dep(T)$, then X is said to be *sound* if for the same initial state S, $X(S) = Y(S)$. In other words, in a sound history, CT compensates for T cleanly, leaving the effects of $dep(T)$ intact. Furthermore, it should be noted that all that this means is that a consistent state is established based on semantic information. This state may not be identical to the state the would have been reached had the compensated-for transaction, T_1, had never taken place. It can be shown that if CT commutes with every transaction in $dep(T)$ then the history is sound.

This definition can be further generalized to include weak forms of compensation soundness. The history X is sound with respect to a reflexive relation R (in short R-sound), if there exists a history Y of $dep(T)$ such that $Y(S)\ R\ X(S)$. For the case of R being equality, the general definition reduces to the "regular" soundness.

5.2.2 Consistency Restoration for Compensating Transactions

A natural extension to the model of compensating transactions is to use ESR specifications as a criteria for soundness. In ESR-based TP systems, the definition of inconsistency specification implies a monotonic distance metric underlying an ϵ-spec. So, our focus is more narrow than the predicate-based generality of Korth's reflexive relations. We are interested in a relation called "Within Bound", denoted by $W(B)$, such that $Y(S)\ W(B)\ X(S)$ if the database state $Y(S)$ is within the distance bound B of state $X(S)$ in the distance metric. If the distance metric is isotropic (as in airline and bank examples and all real-world applications that have cartesian or metric database state spaces due to the symmetry property) then the relation $W(B)$ is reflexive. The result is that $W(B)$-sound histories are ESR.

In this method, the basic transaction management system is compensating transactions [12]; an application programmer is expected to write compensating transactions for any transaction in the system. The role of the ACR method is to restore consistency within data-ϵ-specs if a compensating transaction, CT, does not commute within trans-ϵ-spec boundaries with the dependent transactions of T (i.e., the relation W(B)). The event that invokes the ACR is the violation of trans-ϵ-spec by a compensating transaction, not the original transaction T for which CT was created. This ACR method is designed to restore consistency for compensating transactions, but compensating transactions are not the implementation strategy of the consistency restoration process itself.

An informal description of this ACR method is that during the processing of a compensation ET, it goes through the history checking for violations of the commutativity property. Whenever a violation is spotted, the method accumulates the update amount for all the involved ETs. If the trans-ϵ-specs are not exceeded, then the compensation remains sound and the algorithm continues.

Example 4. Consider a variation of Example 3 in which each of the transactions T_1, T_2, and T_3 has committed after each of their respective operations. The resulting history is:

$$T_1.\text{begin}; T_2.\text{begin}; T_3.\text{begin}; T_1.\text{writelock}(y); T_1 : y = y - 200; T_1.\text{commit}(y); \tag{7}$$

$$T_2.\text{writelock}(y); T_2 : y = y \times 10; T_2.\text{commit}(y); \tag{8}$$

$$T_3.\text{writelock}(y); T_3 : y = y - 2500; T_3.\text{commit}(y). \tag{9}$$

This history is clearly serializable. Now, suppose that T_2 is determined to be erroneous and must be compensated for. According to [12], T_2 can be compensated for after its changes have been made permanent to the database. T_2's compensation, CT_2, divides the value of y by 10. Let us further suppose that there is a Q^{ET}, T_4, to be executed with a $ImpLimit \leq 2500$.

We can see that though CT_2 does not commute with $dep(T)$, this history is ESR for the boundary conditions placed by Q^{ET}, T_4. In this case, the ACR must calculate the distance between the absolute value of the data state given the history $X(S) : T$, $dep(T)$, and CT $(y = 550)$ and the absolute value of the distance function for the history $Y(S) = dep(T)(y = -1700)$ for a total inconsistency of 2250. Since $2250 \leq 2500$, the $ImpLimit$ of T_4, the compensation is ESR.

When the $W(B)$ relation is violated, there are at least four different actions that an ACR method can take to reduce the inconsistency in a compensating transaction system. The first solution is to abort the violating compensating ET. Though Compensating Transactions are generally defined to be non-abortable [12], we think this is sometimes a viable choice. Consistency for Compensating Transactions is defined by the soundness of the relation between $X(S)$ and $Y(S)$. Since the definition of $X(S)$ requires committed T/CT pairs, compensating transactions must always commit. However, this model can be relaxed in light of ESR since residual inconsistency is allowed and controlled.

Note that the residual inconsistency from an aborted CT can be calculated separately from the inconsistency managed by the $W(B)$ relation. The inconsistency managed by the $W(B)$ relation represents the side-effects of a non-commutative CT on the dependent transactions of T whereas the inconsistency created by the abort of a compensating transaction (and not the original transaction, T) is captured by the CGT definition. This is because the transaction for which a CT was created is considered erroneous and must be compensated for. If this compensation is aborted, the original transaction is still erroneous and its effects on the data state must be counted towards the overall inconsistency in the data state (as defined by our G^{ET} model). The W(B) relation really only applies to CT's, not Ts. Although allowing CTs to abort may seem intractable in classic transaction theory, it is consistent with real-world databases where some amount of residual permanent inconsistency (e.g., due to data entry errors) is inevitable.

The second solution is to abort the compensating transaction and retroactively UNDO the original committed ET. Here we depart from the compensating transaction model as defined by Korth et. al. [12]. In their model of compensation, one of the basic purposes of CT is to provide a way to undo the effects of a committed transaction, particularly for those cases where a transaction cannot be "physically" undone. Hence, undoing of the original committed ET is not formally part of their model. In our metric space model, however, original committed transactions can be undone. Our model also supports the relaxation of the classical ACID properties by allowing committed transactions to be undone. This UNDO operation on the original transaction could result in cascading UNDO/REDOs for committed dependent transactions and cascading aborts for dependent transactions in progress, depending on the soundness of the resulting history. Although the worst case of redoing the entire history is costly, the number of transactions affected can be smaller in a G^{ET} environment, since only the ETs with exceeded bounds must be be undone. Furthermore, as Example 3 shows, some optimizations may be possible to prevent the UNDO/REDO of an entire history.

The third solution is to abort transactions in $dep(T)$ that do not commute with the compensating transaction, CT until the level of inconsistency can be brought within the $W(B)$ relation. Similar to solution 2, this approach can be costly if the number of aborts is high; in the worst case, cascading aborts of an entire history may occur. However, again, because some residual inconsistency is tolerable in a G^{ET} model, fewer aborts may be required if optimizations are performed and data-ϵ-specs are met.

In the fourth solution, the ACR method processes the entire history to the end, finding all transactions which have bounds violations as a result of the CT and reports the result to the compensation transaction. Based on this information, the compensation transaction can determine the magnitude of the total conflicts with it before choosing one of the first three solutions. For example, cascaded aborts may be preferable for a small number of aborts, but if $dep(T)$ is large then a large number of aborts may force the compensation to stop. On the other hand, if the abort of the compensating transaction still leaves more residual inconsistency in the data state than the data-ϵ-spec allows, undoing of the original transaction or aborting transactions in $dep(T)$ may be the only alternative.

Example 5. Reconsider Example 4 if Q's $ImpLimit = 2000$ for a concrete example of how an $W(B)$ violations can be processed. In this case, CT_2 violates the $W(B)$ relation, $X(S)$ is unsound, and the $ImpLimit$ of $Q^{ET}_{,}$ T_4, is exceeded. We reconsider each of the restoration options just described.

Alternative 1 is not a viable solution in this particular instance, since the residual inconsistency of T_2 exceeds the $ImpLimit$ of Q^{ET}. From our discussion above, if CT_2 is aborted, then T_2's updates are considered as exported inconsistency. The value of the database state without T_2's execution, e.g. the history $\{T_1, T_3, T_4\}$ leaves y = -1700. The absolute value of the database state from the history $\{T_1, T_2, T_3, T_4\}$ (and not CT_2) is 5500. The absolute value of the distance between these two states is $7200 \geq ImpLimit$ of Q^{ET}, T_4.

Alternative 2 does provide a sound solution. The ACR aborts CT_2 and undoes the original

T_2. This also requires the REDO of T_3. This solution reduces to a classic recovery mechanism.

Alternative 3 also provides a sound solution. The ACR undoes the dependent transaction T_3 with which CT_2 does not commute and then resumes the execution of CT_2. In this case, the value of y is incremented by 2500 and then CT_2 can be executed. The resulting inconsistency is 0. A solution which allows other transactions which otherwise have normal execution to be undone may, of course, strike terror in the hearts of classical TP researchers, but it in practical processing environments higher priority transactions are not uncommon.

Finally, in the fourth alternative, the ACR reports to CT_2 that Q^{ET}, T_4, has had a bounds violation. Had there been other transactions with bounds violations in the system, they would be reported as well. The programmer of CT_2 can determine which of the viable alternatives, i.e. the second or third solutions, is most appropriate for the particular environment and proceed accordingly.

Note that our model explicitly defines inconsistency and checks for places that it can arise. This type of analysis is facilitated by ESR because it is semantics-independent. This does not prevent ESR from incorporating the explicit specification of semantics-dependent inconsistency. In contrast, sagas [8] as proposed are implicitly dependent on application semantics for the maintenance of database consistency.

5.3 A Hybrid Approach

In a third ACR method, we can create a hybrid of the method based on classic recovery from Section 5.1 and the method for compensating transactions described in Section 5.2. In this hybrid model, we use compensating transactions as the basic TP paradigm. However, rather than simply checking for consistency violations from the execution of CTs, we also allow original transaction's to execute as G^{ET}s. Now, there are two classes of transactions that can introduce inconsistency into the database: Ts and CTs (where Ts and CTs are not necessarily matched pairs.)

We can take advantage of the basic compensating transaction model to create ACR method for the original transactions, T, in the system. In this scenario, the ACR generates compensating transactions solely for the purpose of consistency restoration when a G^{ET} exceeds some boundary. These compensating transactions must also satisfy $W(B)$ as defined above. Note that in this scenario, the CTs generated by the ACR are just one special type of compensating transactions in the system; they execute concurrently with other transactions and CTs coded by the application's programmer. Note that we can treat all compensating transactions uniformly without giving special care to those created to reduce inconsistency by the ACR persay. At first this may not appear to be the case since in we do allow CT's to be aborted. But, since we only allow this step when ESR bounds are maintained, we are guaranteed that either the consistency restoring CT will execute, or some other method will be implemented to reduce the inconsistency. In the worst case, cascading aborts or UNDO/REDO operations would be required.

Example 6. Building on Examples 3 and 4, we can illustrate how this method operates. Suppose we have the same three transactions which share uncommitted data as in Example 3. In addition, we extend the history with the Q^{ET} with $ImpLimit \leq 2500$ from Example 4 to execute after T_3. Recall from Example 3 that with the execution of T_3, the inconsistency in the database exceeds the data-ϵ-spec and it was shown that removing the effects of T_2 was a more effective consistency restoration choice.

In this hybrid model, the ACR method spawns, into the regular transaction stream, a compensating transaction, CT_2, that contains the compensation operation for T_2, i.e., the division of y by 10. Given the same initial value of y, the execution of $\{T_1, T_2, T_3, CT_2\}$ results in $y =$ 550. The execution of $\{T_1, T_3\}$ leaves $y =$ -1700. The absolute value of the distance between the two states is 2250 which is within the $ImpLimit$ of $Q^{ET}_{,}$ T_4. Hence, in this example, the ACR is able to reduce the inconsistency in the data state to within a tolerable range by piggybacking on the transaction model itself.

5.4 Independent Updates

Besides compensations, another method of consistency restoration is *independent updates*. In these cases, an independent source of consistent data is available. From time to time the consistent data is used to overwrite potentially inconsistent data. The first important example of this method is the propagation of replica updates in primary copy methods, such as Grapevine [5]. Since all the updates are performed first in the primary copy, the secondary copies may be allowed to diverge (within bounds specified by each distributed ET). A similar situation occurs with bank accounts. The bank database is processed in batch mode at night, at which time the updates are made. Although, each branch may log some local operations into the local replica (usually on paper), the official copy is the central database. Specific examples of bounded inconsistency in replicated systems are described in a previous paper applying ESR to asynchronous replication [18].

Another class of applications that use independent updates are the signal acquisition systems that receive fresh data every so often, such as radars or satellite photos. Even if the current data is inconsistent, a consistent version is expected to arrive at certain time intervals to restore consistency. In these embedded systems, inconsistent data (e.g., from different versions) may be used for obtaining preliminary results while the system waits for the next fresh signal. This is particularly useful for imprecise computations where partial results or order-of-magnitude results are potentially useful.

One way to use independent updates is to emulate the bank practice. An update is made locally and immediately, but the update is sent to the central site in a reliable message [4]. The update in the central site satisfies the necessary rigor in consistency constraints, for example, serializability. Periodically the central site propagates the official updates known to be consistent to the local sites. This is the way banks clear checks and maintain account databases. We can use ESR to implement an ATM's daily limit, for example. This way, any losses due to inconsistency will be limited.

6 Related Work

Besides ESR, notions of correctness weaker than SR have been proposed. Gray's different degrees of consistency [9], offers an example of a coarse spectrum of consistency. Degree 3 consistency is equivalent to SR, but degree 2 consistency offers higher concurrency for queries— at the cost of reduced consistency— since updates are allowed to "dirty" data already read by queries. Degree 2 is reportedly used at many DB2 installations, underscoring the importance of integrating inconsistency specifications. However, there are two limitations in this approach. First, degree 2 is peculiar to a particular concurrency control algorithm, namely two-phase locking. Second, because no bounds are set on the total amount of inconsistency, degree 2 queries will become less accurate as a system grows larger. Finally, ESR offers a much finer granularity control than the degrees of consistency.

Another example of optimization to alleviate data contention is the Escrow Method, introduced by Reuter [22] and further described by O'Neil [15]. Using commutative, associative, and invertible operators, Escrow Method bounds numerical data uncertainty in an interval, allowing multiple concurrent accesses to the same data item. In the restricted model (i.e. excluding G^{ET}), ESR may be added to Escrow to allow additional concurrency since there are no restrictions on the properties of operators. Issues discussed in the general model of ESR (i.e. the introduction of G^{ET}) have no direct counterpart in the Escrow Method. A fundamental difference between the Escrow Method and the G^{ET} model is that G^{ET}s allow permanent inconsistency into the database and asynchronous methods are used to remove it when boundaries are exceeded. The Escrow method, on the other hand, does not allow unserializable histories and hence no permanent inconsistency is introduced into the database. As an enchancement to ESR algorithms, Escrow methods may be used to sharpen the estimate in the inconsistency counters when appropriate operators are known.

Quasi-serializability (QSR) has been proposed [6] as an abstract correctness criterion for a multidatabase environment. QSR specifies that local databases and global schedulers should maintain SR, but isolates a global scheduler from the local schedulers. QSR is well-defined and easy to implement. However, its applicability is limited in the trade-off between consistency and performance its global serializability requirement. At the same time, unbounded inconsistency may be found when we consider the global history and the local histories together.

Garcia-Molina et al. [8] proposed *sagas* that use semantic atomicity [7] which rely on transaction semantics to define correctness. Sagas differ from ESR because an unlimited amount of inconsistency (revealed before a compensation) may propagate and persist in the database. Levy et al [14] defined *relaxed atomicity* to model non-atomic transactions similar to sagas. Non-atomic transactions are composed of steps, which may be a forward step or a recovery step. They also describe the Polarized Protocol to implement Relaxed Atomicity. The main difference between ESR and these notions of correctness is that ESR is independent of application semantics. ESR also allows a larger number of execution histories. The Polarized Protocol, for example, does not allow global state from an incomplete transaction to be seen by other transactions.

An implementation issue in asynchronous TP is to guarantee uniform outcome of distributed transactions running asynchronously. Unilateral Commit [11] is a protocol that uses reliable message transmission to guarantee that a uniform decision is correctly carried out. Optimistic Commit [13] is a protocol that uses Compensating Transactions [12] to undo the effects of partial results to reach a uniform decision. This is but one aspect of the autonomous TP problem.

Sheth et al [23] use the notion of *eventual consistency* to define *current copy serializability* (CPSR) for replicated data. Each update is done on a current copy and asynchronously propagated to the other replicas. Users have control over when the updates are propagated, and the scheme reduces to synchronous replication when the propagation delay is set to zero. Eventual consistency and ESR both provide asynchronous processing with an adjustable inconsistency tolerance. The difference is that ESR is defined for general asynchronous TP with families of ADC and ACR methods.

An example of asynchronous replication methods is Quasi-Copies [1]. Different inconsistency constraints such as time delay can be specified by the user and the system will propagate updates to maintain copy consistency accordingly. ESR can be used to model ETs reading quasi-copies, since the inconsistency specifications are similar. Beyond ESR's usefulness in asynchronous replication [18], we can ESR in asynchronous TP.

Data-value Partitioning [24] has been proposed as a method for increasing distributed TP system availability and autonomy by explicitly separating parts of the value of a data item into different sites. Since the different parts may operate asynchronously even during network partitions, Data-value Partitioning increases autonomy because of its non-blocking character. The basic idea is to allow more parallel processing by dividing the data item value. However, this makes reading a data value non-trivial. ESR can be used in the modeling and management of partitioned data-values.

7 Conclusions

Classic transaction models do not include inconsistency, since a transaction is defined as a program that transforms a consistent database state into another consistent state. In this paper, we have used ESR to extend the transaction model to include inconsistency. To achieve this, we defined data-ϵ-spec for data items in analogy to the trans-ϵ-spec in epsilon transactions (ETs). Each data item may contain some inconsistency (stored with the value and managed by the DBMS), limited by its data-ϵ-spec.

When ETs execute in an unserializable order, the data inconsistency is accumulated by the ET. If the inconsistency is tolerable (compared to trans-ϵ-spec) then the ET commits. Otherwise, the ET may wait, abort, or trigger a consistency restoration (CR) method. We described several CR methods in the paper, with different trade-offs in terms of the amount of information that needs to be stored and amount of processing needed to bring the data fuzziness to below its data-ϵ-spec levels.

By no means have we exhausted the important topic of inconsistency management in databases. For major application areas such as scientific data management, a more refined modeling of inconsistency is needed to make this service attractive to users. Much work remains to be done. Starting from the definition of inconsistency (in terms of database state space geometric properties), through the design of divergence control and consistency restoration, ending with the policies to invoke consistency restoration, we need to formalize the notation and describe the algorithms in more detail. But we believe that we have introduced the basis for this line of work in this paper.

References

[1] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval systems. *ACM Transactions on Database Systems*, 15(3):359–384, September 1990.

[2] R. Alonso and C. Pu. Inconsistency specifications for epsilon-serializability. Technical Report CUCS-0xx-91, Department of Computer Science, Columbia University, May 1991.

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

[4] P.A. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. In *Proceedings of 1990 SIGMOD International Conference on Management of Data*, pages 112–122, May 1990.

[5] A.D. Birrell, R. Levin, R.M. Needham, and M.D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of ACM*, 25(4):260–274, April 1982.

[6] W. Du and A. Elmagarmid. Quasi serializability: a correctness criterion for global concurrency control in InterBase. In *Proceedings of the International Conference on Very Large Data Bases*, pages 347–355, Amsterdam, The Netherlands, August 1989.

[7] H. Garcia-Molina. Using semantic knowledge for transactions processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[8] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 249–259, May 1987.

[9] J.N. Gray, R.A. Lorie, G.R. Putzolu, and I.L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *Proceedings of the IFIP Working Conference on Modeling of Data Base Management Systems*, pages 1–29, 1979.

[10] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[11] M. Hsu and A. Silberschatz. Unilateral commit: A new paradigm for reliable distributed transaction processing. In *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan, February 1990.

[12] H. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[13] E. Levy, H. Korth, and A. Silberschatz. An optimistic commit protocol for distributed transaction management. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, Denver, Colorado, May 1991.

[14] E. Levy, H. Korth, and A. Silberschatz. A theory of relaxed atomicity. In *Proceedings of the 1991 ACM Symposium on Principles of Distributed Computing*, August 1991.

[15] P.E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11(4):405-430, December 1986.

[16] C.H. Papadimitriou. Serializability of concurrent updates. *Journal of ACM*, 26(4):631–653, October 1979.

[17] C. Pu, W.W. Hseush, G.E. Kaiser, P. S. Yu, and K.L. Wu. Distributed divergence control algorithms for epsilon serializability. In *Proceedings of the 1993 International Conference on Distributed Computing Systems*, To Appear 1993.

[18] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 377–386, Denver, May 1991.

[19] C. Pu and A. Leff. Autonomous transaction execution with epsilon-serializability. In *Proceedings of 1992 RIDE Workshop on Transaction and Query Processing*, Phoenix, February 1992. IEEE/Computer Society.

[20] C. Pu, A. Leff, and S.W. Chen. Heterogeneous and autonomous transaction processing. *IEEE Computer*, 24(12):64–72, December 1991. Special issue on heterogeneous databases.

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

[22] A. Reuter. Concurrency on High-Traffic Data Elements. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 83–92, 1982.

[23] A. Sheth, Yungho Leu, and Ahmed Elmagarmid. Maintaining consistency of interdependent data in multidatabase systems. Technical Report CSD-TR-91-016, Computer Science Department, Purdue University, March 1991.

[24] N. Soparkar and A. Silberschatz. Data-value partitioning and virtual messages. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, Nashville, Tennessee, April 1990.

[25] K.L. Wu, P. S. Yu, and C. Pu. Divergence control for epsilon-serializability. In *Proceedings of Eighth International Conference on Data Engineering*, pages 506–515, Phoenix, February 1992. IEEE/Computer Society.