A Formal Characterization of Epsilon Serializability

Krithi Ramamritham[1]
Dept. of Computer Science
University of Massachusetts
Amherst MA 01003

Calton Pu[2]
Dept. of Computer Science
Columbia University
New York, NY 10027

Technical Report No. CUCS-044-91

Abstract

Epsilon Serializability (ESR) is a generalization of classic serializability (SR). ESR allows some limited amount of inconsistency in transaction processing (TP), through an interface called epsilon-transactions (ETs). For example, some query ETs may view inconsistent data due to non-SR interleaving with concurrent updates. In this paper, we restrict our attention to the situation where query-only ETs run concurrently with *consistent* update transactions that are SR without the ETs.

This paper presents a *formal characterization* of ESR and ETs. Using the ACTA framework, the first part of this characterization formally expresses the inter-transaction conflicts that are recognized by ESR and, through that, defines ESR, analogous to the manner in which conflict-based serializability is defined. The second part of the paper is devoted to deriving expressions for: (1) the *inconsistency* in the values of data – arising from ongoing updates, (2) the inconsistency of the results of a query – arising from the inconsistency of the data read in order to process the query, and (3) the inconsistency exported by an update ET – arising from ongoing queries reading uncommitted data produced by the update ET. These expressions are used to determine the preconditions that ET operations have to satisfy in order to maintain the limits on the inconsistency in the data read by query ETs, the inconsistency exported by update ETs, and the inconsistency in the results of queries. This determination suggests possible mechanisms that can be used to realize ESR.

---

[1]partially supported by the National Science Foundation under grant IRI-9109210.
[2]partially supported by NSF, IBM, DEC, AT&T, Oki Electric Ind. and Texas Instruments.

Contents

1 Introduction

Epsilon Serializability (ESR) [19, 26], a generalization of classic serializability (SR), explicitly allows some limited amount of inconsistency in transaction processing (TP). ESR enhances concurrency since some non-SR execution schedules are permitted. For example, epsilon-transactions (ETs) that just perform queries may execute in spite of ongoing concurrent updates to the database. Thus, the query ETs may view uncommitted, i.e., possibly inconsistent, data. Concretely, an update transaction may *export* some inconsistency when it updates a data item while query ETs are in progress. Conversely, a query ET may *import* some inconsistency when it reads a data item while uncommitted updates on that data item exist. The correctness notion in ESR is based on bounding the amount of imported and exported inconsistency for each ET.

In its full generality, update ETs may view inconsistent data the same way query ETs may. However, in this paper we restrict our attention to the situation where query-only ETs run concurrently with *consistent* update transactions. That is, the update transactions are *not* allowed to view uncommitted data and hence *will* produce consistent database states. In distributed TP systems, ESR may increase system availability and autonomy [20] since asynchronous execution is allowed. But in this paper we restrict our attention to ESR in a centralized TP system.

This paper is organized as follows. Section 2.1 gives an informal introduction to ESR and to ETs. Also, it specifies the bound on the imported inconsistency in terms of invariant properties. Section 2.2 formally expresses the inter-transaction conflicts that are recognized by ESR and through that defines ESR – analogous to the manner in which conflict-based serializability is defined. Concepts from the ACTA transaction framework [5, 6, 4] are used for this definition.

Sections 3 and 4 deal, respectively, with the inconsistency imported by a query ET when it reads an inconsistent data item and with the inconsistency exported by an update ET when it writes to a data item read by an ongoing query. The inconsistency of the value of the data is quantified in Section 3.1 and, based on this, preconditions on ET operations are derived in Section 3.2 to maintain the limits on imported inconsistency as specified in Section 2.1. These preconditions point to possible mechanisms that can be used to realize ESR. Section 3.3 considers transaction aborts and Section 3.4 handles update locks, as opposed to write locks. Section 4 provides a similar treatment for ensuring the limits on inconsistency exported by update ETs.

Section 5 derives the effect of the inconsistency of the data read by a query on the *results* produced by the query. This section shows some of the restrictions that need to be imposed on the queries and updates so as to be able to bound the inconsistency in the result of the query to lie within reasonable limits. This helps characterize the situations in which ESR is applicable.

Related work is discussed in Section 6 while section 7 concludes the paper and offers suggestions for further work.

2 ETs and ESR

2.1 Basic Terminology

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. A database state space S_{DB} is a *metric space* if it has the following properties:

- A distance function $distance(u, v)$ is defined over every pair of states $u, v \in S_{DB}$ on real numbers.

  The distance function can be defined as the absolute value of the difference between two states of an account data item. For instance, the distance between \$50 and \$120 is \$70. Thus, if the current account balance is \$50 and \$70 is credited, the distance between the new state and the old state is \$70.

- Symmetry. For every $u, v \in S_{DB}$, $distance(u, v) = distance(v, u)$.

  Continuing with the example, suppose, the current account balance is \$120 and \$70 is debited. The distance between the new state and the old state is still \$70.

- Triangle inequality. For every $u, v, w \in S_{DB}$, $distance(u, v) + distance(v, w) \geq distance(u, w)$.

  The account data clearly satisfies 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.

Many database state spaces have such a regular geometry. As we just saw, in banking databases, dollar amounts possess these properties. Similarly, airplane seats in airline reservation systems also form a metric space.

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.

ESR defines correctness for both consistent states and inconsistent states. In the case of consistent states, ESR reduces to classic serializability. In addition, ESR associates an *amount* of inconsistency with each inconsistent state, defined by its distance from a consistent state. Therefore, ESR has meaning for any state space that possesses a distance function. In general, serializable executions produce answers that have zero inconsistency, but if a (non-serializable) query returns an answer that differs from a serializable result by at most \$10,000 we say that the amount of inconsistency produced by the query is \$10,000. In addition, the triangle inequality and symmetry properties help us design efficient algorithms. In this paper, we will confine our attention to state spaces that are metric spaces.

To an application designer and transaction programmer, an ET is a classic transaction with the addition of inconsistency limits. A query ET has an *import-limit*, which specifies the maximum amount of inconsistency that can be imported by it. Similarly, an update ET has an *export-limit* that specifies the maximum amount of inconsistency that can be exported by it. For simplicity of presentation, we examine in detail ETs when *import-limit*s are placed on individual data items (a single attribute in the relational model). The algorithms can be extended to handle an *import-limit* that spans several attributes (e.g., checking accounts and savings accounts).

An application designer specifies the limit for each ET and the TP system ensures that these limits are not exceeded during the execution of the ET. For example, a bank may wish to know how many millions of dollars there are in the checking accounts. If this query were executed directly on the checking accounts during the banking hours, serious interference would arise because of updates. Most of the interference is irrelevant, however, since typical updates refer to small amounts compared to the query output unit, which is in millions of dollars. Hence we must be able to execute the query during banking hours. Specifically, under ESR, if we specify an *import-limit* for the query ET, for example, of \$100,000, for this query, the result also would be guaranteed to be within \$100,000 of a consistent value (produced by a serial execution of the same transactions). For example, if the ET returns the value \$357,215,000 (before round-off) then at least one of the serial transaction executions would have yielded a serializable query result in the \$325,215,000±\$100,000 interval.

During the execution of each ET, the system needs to maintain the amount of inconsistency the ET has imported so far. Note that the amount of inconsistency is given by the distance function and the incremental accumulation of inconsistency depends on the triangle inequality property of metric spaces. Without triangle inequality, we would have to recompute the distance function for the entire history each time a change occurs. In Section 3.1 we derive the algorithms necessary to maintain the specified limit on the imported inconsistency.

Let $import\_inconsistency_{t,x}$ stand for the amount of inconsistency that has already been imported by ET t on data item x. Let $import\_limit_{t,x}$ stand for the import-limit that has been set for ET t with respect to data x. Similarly, $export\_limit_{t,x}$ and $export\_inconsistency_{t,x}$ stand for the corresponding export-limit and the exported inconsistency. Thus, the system that supports ESR must maintain the following invariant for every ET t (on data item x):

$$\begin{aligned} import\_inconsistency_{t,x} &\leq import\_limit_{t,x} && (1) \\ export\_inconsistency_{t,x} &\leq export\_limit_{t,x}. && (2) \end{aligned}$$

We call the invariants (1) and (2) Safe(t,x) for brevity. For query ET q reading x, Safe(q,x) reduces to:

$$\begin{aligned} import\_inconsistency_{q,x} &\leq import\_limit_{q,x} && (3) \\ export\_inconsistency_{q,x} &= 0. && (4) \end{aligned}$$

For a consistent update ET t, Safe(t,x) reduces to:

$$\begin{aligned} export\_inconsistency_{t,x} &\leq export\_limit_{t,x} && (5) \\ import\_inconsistency_{t,x} &= 0. && (6) \end{aligned}$$

Safe(q,x) states that a query ET cannot exceed its *import-limit* and that a query cannot export inconsistency. Similarly, Safe(t,x) states that an update ET cannot exceed its *export-limit* nor can it import any inconsistency.

2.2 Formal Definition of ESR

In order to define ESR formally, we use the ACTA framework [5, 4, 6] to introduce the notion of conflicts between operations and discuss the dependencies induced between transactions when they invoke conflicting transactions.

For a given state s of a data item, we use $return(s, a)$ to denote the output produced by operation a, and $state(s, a)$ to denote the state produced after the execution of a. $value(s, P)$ denotes the value of predicate P in state s.

A history $H^{(x)}$ of operation invocations on a data item x, $H^{(x)} = a_1 \circ a_2 \circ \ldots \circ a_n$, indicates both the order of execution of the operations, (a_i precedes a_{i+1}), as well as the functional composition of operations. Thus, a state s of a data item produced by a sequence of operations equals the state produced by applying the history $H^{(x)}$ corresponding to the sequence of operations on the data item's initial state s_0 ($s = state(s_0, H^{(x)})$). For brevity, we will use $H^{(x)}$ to denote the state of a data item produced by $H^{(x)}$, implicitly assuming initial state s_0.

Definition 1 *Two operations a and b* conflict *in a state produced by $H^{(x)}$, denoted by conflict($H^{(x)}, a, b$), iff*

$$\begin{aligned}(state(H^{(x)} \circ a,\ b) &\neq state(H^{(x)} \circ b,\ a)) \quad \vee \\ (return(H^{(x)},\ b) &\neq return(H^{(x)} \circ a,\ b)) \vee \\ (return(H^{(x)},\ a) &\neq return(H^{(x)} \circ b,\ a)).\end{aligned}$$

Thus, two operations conflict if their effects on the state of a data item or their return values are not independent of their execution order.

Let $a_{t_i}[x]$ denote operation a invoked by t_i on data item x. ($a_{t_i}[x] \rightarrow b_{t_j}[x]$) implies that $a_{t_i}[x]$ appears *before* $b_{t_j}[x]$ in H.

Let us first define the classic serializability correctness criterion.

Definition 2 *Let $\mathcal{C}$ be a binary relation on transactions, and t_i and t_j be transactions.*
$(t_i\ \mathcal{C}\ t_j), t_i \neq t_j$ *iff*
$\exists x\ \exists a, b\ (conflict(H^{(x)}, a_{t_i}[x], b_{t_j}[x]) \wedge (a_{t_i}[x] \rightarrow b_{t_j}[x]))$.
Let $\mathcal{C}^*$ be the transitive-closure of $\mathcal{C}$; i.e.,
$(t_i\ \mathcal{C}^*\ t_j)$ *if* $[(t_i\ \mathcal{C}\ t_j) \vee \exists t_k\ (t_i\ \mathcal{C}\ t_k \wedge t_k\ \mathcal{C}^*\ t_j)]$.
A set of transactions T is (conflict preserving) serializable *iff*
$\forall t \in T\ \neg(t\ \mathcal{C}^*\ t)$.

To illustrate the practical implications of these definitions, let us consider the case where all operations perform in-place updates. In this case, if transactions t_i and t_j have a $\mathcal{C}$ relationship, i.e., they have invoked conflicting operations, a commit dependency [4] forms between t_i and t_j[3]. The commit order induced by the $\mathcal{C}$ relation corresponds to the serialization order. By requiring that there be no cycles in the $\mathcal{C}$ relation, the above definition states that the commit order, and hence the serialization order, must be acyclic.

The differences between SR and ESR are stated via the following definitions.

[3]Conflicting operations may also produce abort dependencies between the invoking transactions; but an abort dependency implies a commit dependency.

Definition 3 *Let* $\mathcal{C}_{ESR}$ *be a binary relation on transactions, and* t_i *and* t_j *be transactions.*

$$(t_i\ \mathcal{C}_{ESR}\ t_j), t_i \neq t_j \ \mathit{iff}$$
$$\exists x\ \exists a, b\ (\mathit{conflict}(H^{(x)}, a_{t_i}[x], b_{t_j}[x]) \wedge (a_{t_i}[x] \rightarrow b_{t_j}[x])$$
$$\wedge\ value(state(H^{(x)} \circ a,\ b), \neg(Safe(t_i, x) \wedge (Safe(t_j, x)))).$$

In other words, t_i and t_j are related by $\mathcal{C}_{ESR}$ if and only if they are related by $\mathcal{C}$ *and* they violate one of the invariants.

Definition 4 *A set of transactions* T *is* (conflict preserving) epsilon serializable *iff*
$\forall t \in T\ (\neg(t\ \mathcal{C}_{ESR}{}^{*}\ t) \wedge safe(t, x))$.

Note that the last term in the definition of $\mathcal{C}_{ESR}$ makes $\mathcal{C}_{ESR}$ strictly *weaker* than $\mathcal{C}$. Since $\mathcal{C}_{ESR}$ is a subset of the $\mathcal{C}$ relationship, a smaller number of dependencies are formed under ESR than under classic serializability. From the above definitions, we can summarize the properties of ESR as compared to serializability:

- When all *import-limit* and *export-limit* are zero, ESR reduces to serializability.
- A set of transactions may not satisfy serializability because of cycles in the $\mathcal{C}$ relation, but may satisfy ESR.
- When some *import-limit*s and *export-limit*s are greater than zero, $\mathcal{C}_{ESR} \subseteq \mathcal{C}$ (given the additional term in definition 3). That is, ESR may allow more operation orderings than serializability.

3 Inconsistency Imported by a Query ET

In this section, we study the amount of *inconsistency* in data values caused by update ETs executing concurrently with query ETs. A query ET *imports* inconsistency when it reads an inconsistent data item. We examine how *import-limits* imposed on queries is maintained. In Section 5, the effect of this imported inconsistency on the results of the query is studied.

3.1 Quantifying Inconsistency

We focus on the inconsistency of a single data item x read by a query q. Informally, *inconsistency* in x with respect to a query q is defined as the difference between the current value of x and the value of x if no updates on x were allowed to execute concurrently with q.

Consider update transactions $t_1 \ldots t_n$ where each of the t_i's updates x. Assume that a locking-based concurrency control scheme is used. Let us define a transaction t_i's *write lock interval* with respect to x to be the interval of time between when t_i acquires a write lock on x and when t_i releases the lock. A *read lock interval* is defined similarly. We allow q to read x multiple times and each of the updating t_i's to write x multiple times when holding the appropriate lock.

Every query q has a set of *Concurrent Update Transactions* (denoted by $\textsc{cut}(q)$). Update ET $t_i \in \textsc{cut}(q)$ if its write lock interval intersects with q's read lock interval. Note that in serializable executions, since write locks are incompatible with read locks, $\textsc{cut}(q) = \emptyset$.

The question we are attempting to answer here is the following: What can one say about the value of x read by q given the $\textsc{cut}(q)$? Our main objective is to bound the inconsistency in the value of x read by q. But first we establish that the transactions in $\textsc{cut}(q)$ are totally ordered, since consistent update ETs are serializable with respect to each other.

Theorem 1 *The serialization order of the transactions $t_i \in \textsc{cut}(q)$, w.r.t. x, is the same as the order in which each t_i obtains the write locks on x which in turn is the same as the order in which they commit.*

Proof: Follows directly from the exclusivity of write locks and from the fact that holders of write locks are required to commit in the order in which they acquire the locks to ensure correctness of the database states when transactions abort.

Now we name the values of x at different points in time:

- $x_{current}$ is the current value of x.
- $x^{t_i}_{final}$ is the value of x committed by transaction t_i.
- $x^{t_i}_{initial}$ is the value of x when transaction t_i in $\textsc{cut}(q)$ begins, i.e., $x^{t_i}_{initial} = x^{t_{i-1}}_{final}$.
- $x^{q}_{initial}$ is defined to be the value of x before any of the transactions in $\textsc{cut}(q)$ begin execution. That is, if $\textsc{cut}(q) \neq \emptyset, x^{q}_{initial} = x^{t_1}_{initial}$, else, $x^{q}_{initial} = x_{current}$.

From these values of x we can derive:

$$current\_change_{t_i,x} = distance(x_{current}, x^{t_i}_{initial})$$

$$max\_change_{t_i,x} = \max_{during\ t_i} \{current\_change_{t_i,x}\}$$

$$final\_change_{t_i,x} = distance(x^{t_i}_{initial}, x^{t_i}_{final})$$

Clearly, $final\_change_{t_i,x} \leq max\_change_{t_i,x}$ and $current\_change_{t_i,x} \leq max\_change_{t_i,x}$.

We are in a position to define inconsistency formally.

$$(x^{q}_{initial} - inconsistency_{q,x}) \leq x_{current} \leq (x^{q}_{initial} + inconsistency_{q,x})$$

That is, $inconsistency_{q,x}$ denotes the $distance$ between $x^{q}_{initial}$ and $x_{current}$. So, inconsistency in the value of x for a query q while t_i is in progress and update ETs $t_1 \ldots t_{i-1}$ have already committed is given by

$$inconsistency_{q,x} = distance(x_{current}, x^{q}_{initial}) = distance(x_{current}, x^{t_1}_{initial})$$

$$\leq distance(x_{current}, x^{t_i}_{initial}) + distance(x^{t_i}_{initial}, x^{t_1}_{initial})$$

$$\leq distance(x_{current}, x^{t_i}_{initial}) + \sum_{j=1}^{i-1} distance(x^{t_j}_{final}, x^{t_j}_{initial})$$

$$= current\_change_{t_i,x} + \sum_{j=1}^{i-1} final\_change_{t_j,x}.$$

Let $committed\_\textsc{cut}(q)$ denote the subset of $\textsc{cut}(q)$ containing the ETs that have committed. Let $t_{current} \in \textsc{cut}(q)$ denote the update transaction whose write lock interval has begun but has not ended yet. If no such $t_{current}$ exists, it has a "null" value and $current\_change_{null,x}$ is defined to be 0.

¿From these discussions we can state the following theorem which expresses (bounds on) the inconsistency of a data item read by a query q when its read lock interval intersects with the write lock intervals of ETs in $\textsc{cut}(q)$.

Theorem 2

$$inconsistency_{q,x} = distance(x_{current}, x^q_{initial}) \tag{7}$$

$$\leq \sum_{t_j \in committed\_\textsc{cut}(q)} final\_change_{t_j,x} + current\_change_{t_{current},x} \tag{8}$$

$$\leq \sum_{t_j \in committed\_\textsc{cut}(q)} final\_change_{t_j,x} + max\_change_{t_{current},x} \tag{9}$$

$$\leq \sum_{t_j \in committed\_\textsc{cut}(q)} max\_change_{t_j,x} + max\_change_{t_{current},x} \tag{10}$$

Whereas expression (7) is an exact expression of the inconsistency, expressions (8) through (10) can be viewed as different *bounds* on $inconsistency_{q,x}$.

We are now in a position to relate the inconsistency bound with the conflict-based definition of ESR given in Section 2.2. Recall the definitions of $\mathcal{C}$ and $\mathcal{C}_{ESR}$:

> A pair of transactions have a $\mathcal{C}$ relationship but not a $\mathcal{C}_{ESR}$ relationship iff one of them is a query and the other is an update *and* the import and export limits are not violated. Let us focus on $\mathcal{C}$ relationships induced by operations on x. Given (10), each of the update transactions t_i that appears in the pairs that belongs to $\mathcal{C}$ but not to $\mathcal{C}_{ESR}$ contributes an inconsistency of at most $max\_change_{t_i,x}$ to the value of x read by q.

3.2 Maintaining Import Limits: Pre-Conditions for ET Operations

To ensure that ($import\_inconsistency_{q,x} \leq import\_limit_{q,x}$) is an invariant, this inequality must be maintained by every update and query transaction. Specifically, it must hold (before and) after every read and write operation as well as every lock and unlock operation invoked by any transaction. In what follows, we will consider the case when all transactions commit and assume that a transaction's locks are removed only when it commits. Under these assumptions, we derive the necessary preconditions for performing the read/write and lock/unlock operations such that import and export limits of transactions are not exceeded. These will in turn be used to show how the lock managers should be constructed.

Let $wlock_{t,x}$ denote the attempt by ET t to obtain a *write* lock on x. $rlock_{t,x}$ is invoked by t to place a read lock on x. Let $unlock_{t,x}$ denote that t removes its current lock on x. We will now consider the semantics of *rlock*, *wlock*, *unlock*, *read* and *write*. There are two situations to consider. The first is if a query ET q is already in progress (initially with $committed\_{\rm CUT}(q) = \emptyset$) when an update transaction begins. This may be followed by other update ETs before q commits. The second is if an update ET is in progress when the query begins.
Recall that our attention is confined to a centralized database with a single lock manager.
Let q be a query and t be an update ET. $\leftarrow$ stands for assignment.
If query q is in progress,

$$\begin{aligned} wlock_{t,x} &\equiv (t_{current} \leftarrow t \wedge {\rm CUT}(q) \leftarrow {\rm CUT}(q) \cup t) \\ unlock_{t,x} &\equiv (t_{current} \leftarrow null \wedge committed\_{\rm CUT}(q) \leftarrow committed\_{\rm CUT}(q) \cup t) \end{aligned}$$

Otherwise, $wlock_{t,x} \equiv ()$ and $unlock_{t,x} \equiv ()$.
If an update transaction t is in progress, $rlock_{q,x} \equiv (t_{current} = t \wedge {\rm CUT}(q) \leftarrow t)$.
Otherwise, $rlock_{q,x} \equiv (t_{current} = null)$.
Here are the semantics of the other operations.

$$\begin{aligned} unlock_{q,x} &\equiv (q \leftarrow null) \\ read_{t,x} &\equiv () \\ read_{q,x} &\equiv (import\_inconsistency_{q,x} \leftarrow inconsistency_{q,x}) \\ write_{t,x}(\Delta) &\equiv (x_{current} \leftarrow x_{current} + \Delta) \end{aligned}$$

Δ is a parameter to the *write* operation that denotes the amount by which x is modified when the write occurs.

It is important to note from the above semantics that a query imports inconsistency only if it performs a read operation. That is, the inconsistency in the value of x due to updates translates to imported inconsistency only when read operations occur.

We will now establish the preconditions necessary to maintain (3), i.e.,

$$(import\_inconsistency_{q,x} \leq import\_limit_{q,x}) \tag{11}$$

Case 1: Preconditions only on $read_{q,x}$ Operations.

Given that inconsistency is imported by q only when it performs a *read*, the following precondition is all we need to maintain (11):

$$inconsistency_{q,x} \leq import\_limit_{q,x}.$$

¿From (7), this implies the precondition

$$distance(x_{current}, x^q_{initial}) \leq import\_limit_{q,x}.$$

Every *read* operations must be intercepted by the transaction management mechanism to ensure that the above precondition holds. If the predicate does not hold, the query will have to be aborted or delayed. If q a long query, this has performance implications. This is the motivation for examining other possible ways to maintain (11).

Case 2: Preconditions on *write* Operations and $rlock_{q,x}$ Operations

Suppose we satisfy the following invariant:

$$inconsistency_{q,x} \leq import\_limit_{q,x},$$

i.e.,

$$distance(x_{current}, x^q_{initial}) \leq import\_limit_{q,x}$$

Note that this is a stronger invariant than (11), i.e, if this is maintained, then (11) will be maintained. (This has a negative side-effect: If the query does not read x at all, then the allowable inconsistency on x has been restricted unnecessarily.) Given the semantics of the various operations, and the expression (7) for inconsistency, the following precondition on *write* results.

$$distance(x_{current} + \Delta, x^q_{initial}) \leq import\_limit_{q,x}$$

and given that x is in metric space, this implies the precondition

$$|\Delta| + distance(x_{current}, x^q_{initial}) \leq import\_limit_{q,x}$$

where $|\Delta|$ denotes the *absolute value* of Δ. (We also use $\mid S \mid$ to denote the cardinality of set S. The meaning should be obvious from the context.) This says that a write should be allowed only if the increase in inconsistency caused by the intended increment will not violate the limit imposed on the inconsistency imported by q.

Even though no precondition is necessary for a *read*, the following precondition is required for $rlock_{q,x}$ when it is invoked while an update transaction t is already in progress:

$$distance(x_{current}, x^q_{initial}) \leq import\_limit_{q,x}.$$

This says that if the changes that have already been done by the update transaction exceed the import limit imposed on q then the query must not be allowed to place a read lock on x.

The above preconditions imply that with each query q, we should maintain $x^q_{initial}$. This can be avoided by maintaining an even stronger invariant, corresponding to the inconsistency bound (8), i.e., by maintaining

$$\sum_{t_j \in committed\_CUT(q)} final\_change_{t_j,x} + current\_change_{t_{current},x} \leq import\_limit_{q,x}.$$

This imposes the following precondition on $write_{t,x}$:

$$\sum_{t_j \in committed\_CUT(q)} final\_change_{t_j,x} + current\_change_{t_{current},x} + |\Delta| \leq import\_limit_{q,x}$$

and the following precondition on $rlock_{q,x}$:

$$current\_change_{t_{current},x} \leq import\_limit_{q,x}.$$

This implies that write operations by update ETs and read lock requests by query ETs have to be monitored to ensure that they are allowed only when the above preconditions hold.

Both these invariants require maintenance of the most recent committed state of x. This is available anyway. However, the need to check every *write* by an update ET implies increased overheads and may also result in aborts or delays of update ETs in progress. Both can be avoided as shown below if an even stronger invariant is maintained.

Case 3: Preconditions on $rlock_{q,x}$ and $wlock_{t,x}$

Consider the following invariant corresponding to inconsistency bound (9):

$$\sum_{t_j \in committed\_CUT(q)} final\_change_{t_j,x} + max\_change_{t_{current},x} \leq import\_limit_{q,x}.$$

This inequality turns out to be the precondition for $wlock_{t_{current},x}$. $rlock_{q,x}$ has the following precondition:

$$max\_change_{t,x} \leq import\_limit_{q,x}. \tag{12}$$

This implies that unlike the previous case, no preconditions are associated with *individual* writes by update transactions. While this reduces transaction management overheads, it does introduce some pessimism into the decision making since worst case changes to x by t are assumed.

The precondition for $wlock_{t_{current},x}$ requires knowledge about $final\_change_{t_j,x}$'s. This can be avoided if the following invariant, corresponding to inconsistency bound (10), is maintained:

$$\sum_{t_j \in committed\_CUT(q)} max\_change_{t_j,x} + max\_change_{t_{current},x} \leq import\_limit_{q,x} \tag{13}$$

(13) is also the precondition for $wlock_{t_{current},x}$. (12) stays as the precondition for $rlock_{q,x}$.

Suppose $max\_change_{t_i,x}$ is the same for all update ETs t_i. Then, a given $import\_limit_{q,x}$ for a query q translates into a limit on the *cardinality* of $CUT(q)$. This is the basis for the implementation of the locks in [26] wherein precondition (13) for *write*s corresponds to LOK-2 and precondition (12) for *read*s corresponds to LOK-1.

3.3 Imported Inconsistency if Update Transactions Abort

So far we have considered the case when all transactions commit. Abortion of update transactions has the effect of increasing the inconsistency imported by a query without changing the value of x. We will prove the following theorem:

Theorem 3 *The maximum* increase *in imported inconsistency caused by aborted transactions is given by*

$$\max_{t \in CUT(q)\ aborted} \{max\_change_{t,x}\}.$$

Proof: Suppose transactions t_1 to t_{i-1} have committed and then t_i begins but subsequently aborts. In addition to the inconsistency due to t_1 to t_{i-1}, derived earlier, if q reads x any time during t_i's execution, it will experience an additional inconsistency of $max\_change_{t_i,x}$. Assume t_i aborts whereby changes made by t_i are obliterated and thus subsequent updates will increase the value of x only with respect to that resulting from t_1 to t_{i-1}.

Suppose all the transactions in $\textsc{cut}(q)$ that follow t_i commit. Then $max\_change_{t_i,x}$ is the only increase to the inconsistency due to aborted transactions and hence the theorem holds.

Suppose instead that t_{i+1} to t_{j-1} commit and t_j aborts. When q reads x after t_j begins, x will only reflect the changes done by (1) transactions t_1 to t_{i-1}, (2) transactions t_{i+1} to t_{j-1}, and (3) transaction t_j. (3) is bounded by $max\_change_{t_j,x}$. If this is larger than $max\_change_{t_i,x}$, then $max\_change_{t_j,x}$ is the increase in inconsistency due to the aborted transactions t_i and t_j and hence the theorem follows for two transaction aborts. If this is smaller, $max\_change_{t_i,x}$ remains the upper bound on the increase. That is, the maximum of the two is the effective increase in inconsistency due to two transaction aborts. This proof extends easily if further transactions abort.

3.4 Inconsistency when Transactions Use Update Locks

Thus far, we have assumed that transactions lock data items with read and write locks. However, notice that the changes they do to x may *commute* since we are considering only incremental changes done by each write. Update locks are more suitable for this scenario than write locks since the former allow more concurrency by permitting multiple concurrent update locks to be applied to a data item. In what follows, we consider the case where update transactions use update locks – as opposed to write locks. Suppose $current\_\textsc{cut}(q)$ denotes the update transactions that are in progress and concurrently executing with q. We state the following variant of theorem 2 without proof:

Theorem 4

$$inconsistency_{q,x} = distance(x_{current}, x^q_{initial})$$

$$\leq \sum_{t \in committed\_\textsc{cut}(q)} final\_change_{t,x} + \sum_{t \in current\_\textsc{cut}(q)} current\_change_{t,x}.$$

$$\leq \sum_{t \in committed\_\textsc{cut}(q)} final\_change_{t,x} + \sum_{t \in current\_\textsc{cut}(q)} max\_change_{t,x}.$$

$$\leq \sum_{t \in committed\_\textsc{cut}(q)} max\_change_{t,x} + \sum_{t \in current\_\textsc{cut}(q)} max\_change_{t,x}.$$

Given these bounds, in a fashion very similar to that of Section 3.2, we can derive the preconditions for the operations invoked by update ETs.

4 Inconsistency Exported by an Update ET

Assume query ETs $q_1 \ldots q_n$ and an update ET t where t's updates x, and q_i's read x. q_i read locks x before reading it. t write locks x before updating it. Because of the exclusivity of write locks, invariant (4) is vacuously true. So we will focus on maintaining (3), i.e., $export\_inconsistency_{t,x} \leq export\_limit_{t,x}$.

Assume that each q_i's read lock interval intersects with t's write lock interval. In this case, we call the q_i's the set of *Concurrent Query Transactions*, $\textsc{cqt}(t)$. Let $committed\_\textsc{cqt}(t)$ denote the ETs in $\textsc{cqt}(t)$ that have committed and $current\_\textsc{cqt}(t)$ denote the ETs in $\textsc{cqt}(t)$ whose read lock intervals have begun but yet to end. Also, $change\_at\_commit_{q_j,x}$ denotes the value of $distance(x_{current}, x^q_{initial})$ at the time q commits.

With this notation, the following expressions can be seen to be the counterparts of expressions (8) through (10).

$$\sum_{q_j \in committed\_\textsc{cqt}(t)} change\_at\_commit_{q_j,x} + (current\_change_{t,x} \times \mid current\_\textsc{cqt}(t) \mid) \quad (14)$$

$$\sum_{q_j \in committed\_\textsc{cqt}(t)} change\_at\_commit_{q_j,x} + (max\_change_{t,x} \times \mid current\_\textsc{cqt}(t) \mid) \quad (15)$$

$$\mid \textsc{cqt}(t) \mid \times max\_change_{t,x}. \quad (16)$$

In a manner similar to the derivation of preconditions to maintain $import\_limit_{q,x}$ in Section 3.2, we can derive the preconditions necessary to ensure that the value of the above inconsistency expressions does not exceed $export\_limit_{q,x}$. We just present the resulting preconditions below.

Consider (14). Following is the precondition for $rlock_{q,x}$.

$$\sum_{q_j \in committed\_\textsc{cqt}(t)} change\_at\_commit_{q_j,x} + (current\_change_{t,x} \times (\mid current\_\textsc{cqt}(t) \mid +1)) \leq export\_limit_{t,x}.$$

Following is the precondition for $write_{t,x}(\Delta)$.

$$\sum_{q_j \in committed\_\textsc{cqt}(t)} change\_at\_commit_{q_j,x} + ((current\_change_{t,x} + \Delta) \times \mid current\_\textsc{cqt}(t) \mid) \leq export\_limit_{t,x}.$$

Consider (15). Following is the precondition for $rlock_{q,x}$.

$$\sum_{q_j \in committed\_\textsc{cqt}(t)} change\_at\_commit_{q_j,x} + (max\_change_{t,x} \times (\mid current\_\textsc{cqt}(t) \mid +1)) \leq export\_limit_{t,x}.$$

Following is the precondition for $wlock_{t,x}$.

$$(max\_change_{t,x} \times \mid current\_{\text{CQT}}(t) \mid) \leq export\_limit_{t,x}.$$

Consider (16). Following is the precondition for $rlock_{q,x}$.

$$(max\_change_{t,x} \times (\mid \text{CQT}(t) \mid +1)) \leq export\_limit_{t,x}.$$

Following is the precondition for $wlock_{t,x}$.

$$(max\_change_{t,x} \times \mid \text{CQT}(t) \mid) \leq export\_limit_{t,x}.$$

Finally, let us consider the case where update ETs use update locks instead of write locks. In this case, because of invariant (4), if an update ET attempts to read while another update ET is in progress, it will be blocked since in this case the update ET imports some inconsistency. Consider *pure* update ETs working on x, that is, they do not read x. In this case, each query imports inconsistency from multiple update ETs. This is the case discussed in Section 3.4. Also, each update ET exports inconsistency to multiple query ETs. This is the case we discussed earlier in this section.

5 Inconsistency in the Results of a Query

Since a query, by definition, does not update data (on disk), it does not affect the permanent state of the database. Furthermore, we have assumed that updates do not import inconsistency, i.e., they operate on consistent database states. Thus, assuming that each update ET maintains database consistency, updates also do not affect the consistency of the database. The only effect of the updates is on the inconsistency of the data read by queries. In Section 3 we derived expressions for the amount of inconsistency imported by a query. Given this inconsistency, the only *observable* effect of a query ET is on the results produced by a query. In other words, the inconsistency imported by a query can *percolate* to the results of a query, in ways that are obviously dependent on the manner in which the query utilizes the values read.

This section is devoted to determining the effect of the inconsistency of data read by a query on its results. In general, a small input inconsistency can translate into an arbitrarily large result inconsistency. Therefore, we study the properties of a query that make the result inconsistency more predictable.

First we establish some terminology. Consider the situation where a query q reads data items $x_1, x_2, \ldots, x_n$ and produces a result based on the values read. In general, the results of such a query can be stated as a function of the form:

$$g(f_1(x_1), f_2(x_2), \ldots, f_n(x_n)) \tag{17}$$

where g denotes a query ET and f_i's are functions such that $f_i : S_{DB} \rightarrow R_f$, where R_f is the range of f_i. We assume that R_f is also a metric space. In practice, typically R_f is a subset of S_{DB}. For example, aggregate functions and queries on the database usually return a value in S_{DB}.

Focusing on *monotonic* queries, in Section 5.1 we derive the inconsistency in the result of a query and show that even though the inconsistency can be bound, the bound may not be tight. Suppose, similar to *import_limit* and *export_limit*, a limit is placed on the inconsistency in the result of a query. In Section 5.2, we derive the preconditions on ET operations imposed by such a limit. In Section 5.3 a class of queries called *bounded* queries is considered. Section 5.4 examines *steady* queries and discusses how queries can be designed to have tighter inconsistency bounds thereby requiring less restrictive preconditions.

5.1 Monotonic Queries

The first important class of queries consists of *monotonic* functions. A function f is *monotonically increasing* if $x \leq y \Rightarrow f(x) \leq f(y)$. A function g is *monotonically decreasing* if $x \leq y \Rightarrow f(x) \geq f(y)$. A function is called *monotonic* if it is either monotonically increasing or decreasing. Without loss of generality in the rest of this section we describe only monotonically increasing functions.

The result returned by a monotonic ET q assuming that the value of x_i read by q is given by $x_{i,read}$ is

$$g(f_1(x_{1,read}), f_2(x_{2,read}), \ldots, f_n(x_{n,read}))$$

where, if $max\_inconsistency_{x_i}$ is the maximum inconsistency in the value of x_i read by q (given by Theorem 2 of Section 3.1), $x_{i,initial}$ is the value of x_i when the first update ET in $\textsc{cut}(q)$ begins, and $x_{min} = x_{i,initial} - max\_inconsistency_{x_i}$ and $x_{max} = x_{i,initial} + max\_inconsistency_{x_i}$, then

$$x_{i,min} \leq x_{i,read} \leq x_{i,max}. \tag{18}$$

Thus, since g and the f_i's are monotonic[4], the result of the query can lie between

$$min\_result_q = g(f_1(x_{1,min}), \ldots, f_n(x_{n,min})) \tag{19}$$

and

$$max\_result_q = g(f_1(x_{1,max}), \ldots, f_n(x_{n,max})) \tag{20}$$

Thus, by our definition of inconsistency,

$$result\_inconsistency_q = \frac{(max\_result_q - min\_result_q)}{2}. \tag{21}$$

Let us look at some examples:

Example 1: n=1; $g = f_i$ = the identity function. This corresponds to the single data element case and hence the inconsistency in the result of q can be seen to be given by (18).

Example 2: n=20; $g = \sum_{i=0}^{20}$; f_i = the identity function. In this case, as one would expect, the result of the query, according to (19) and (20), will lie between $\sum_{i=0}^{20}(x_{i,initial} - max\_inconsistency_{x_i})$ and $\sum_{i=0}^{20}(x_{i,initial} + max\_inconsistency_{x_i})$.

[4]If f_i is not monotonic, the smallest (largest) value of f_i need not correspond to the smallest (largest) value of x_i.

Example 3: n=20; $g = \sum_{i=0}^{20}$; $f_i = ((x_i > 5000) \times x_i)$. (A predicate has a value 1 if it is true, otherwise 0.) In this case, the result of the query, according to (19) and (20), will lie between

$$\sum_{i=0}^{20}(((x_{i,initial} - max\_inconsistency_{x_i}) > 5000) \times (x_{i,initial} - max\_inconsistency_{x_i}))$$

and

$$\sum_{i=0}^{20}(((x_{i,initial} + max\_inconsistency_{x_i}) > 5000) \times (x_{i,initial} + max\_inconsistency_{x_i})).$$

Example 4: This is a concrete case of Example 3. Consider a bank database with 20 accounts, numbered 1-20. Each account with an odd number happens to have \$5,001 and even-numbered accounts have \$4,999. The only update transaction in the system is: Transfer(Acc_i, Acc_j, 2), which transfers \$2 from Acc_i into Acc_j. The query ET sums up all the deposits that are greater than \$5,000. Suppose that the first set of transactions executed by the system are: Transfer(Acc_{2i-1}, Acc_{2i}, 2), for i=1, ... , 10. When these finish, the following are executed: Transfer(Acc_{2i}, Acc_{2i-1}, 2), for i=1, ... , 10.

These update transactions maintain the total of money in the database, and it is easy to see that a serializable execution of the query ET should return \$50,010, since at any given time, exactly 10 accounts have more than \$5,000.

This query will produce a result between \$0 and \$100,080 since it is exactly Example 3, where,

$$\begin{aligned} \forall i = 1, \ldots, 10, x_{(i*2)-1,initial} &= \$5,001. \\ \forall i = 1, \ldots, 10, x_{(i*2),initial} &= \$4,999. \\ \forall i = 1, \ldots, 20, max\_inconsistency_{x_i} &= 4. \end{aligned}$$

The range of the result does include the serializable result of \$50,010. However, given that the range is not very "tight", it is too pessimistic. This occurs because the inconsistency caused by the updates percolate, in a rather drastic manner, to the results of the query. In Section 5.4, we identify a class of queries for which tight bounds on the results of a query exist.

One other point to note here is that even this bound requires knowledge of $x_{i,initial}$, the value of x_i when the first ET in $\textsc{cut}(q)$ begins. This has practical implications. Specifically, before an update is begun, the data values may have to be logged in order to derive the inconsistency for the queries that may subsequently begin. This is the case of systems that require UNDO capability (using the STEAL buffering policy [12]).

Given that the lower bound on the result of the above query is 0, one may be tempted to take the following solution: Assume that $x_{i,initial}$ is the smallest value x_i can take, i.e., 0. It is not too difficult to see why this will not produce the correct range for the above query's result.

5.2 Pre-Conditions for Monotonic Queries

Suppose $result\_inconsistency\_limit_q$ denotes the maximum inconsistency that an application can withstand in the result of a query q. Then

$$result\_inconsistency_q \leq result\_inconsistency\_limit_q$$

is an invariant. Just as we derived preconditions to maintain $import\_limit_{q,x}$ and $export\_limit_{q,x}$, we can derive preconditions to maintain the above invariant.

For instance, consider the expression (10) for $max\_inconsistency_x$. From this, given (21) and the semantics of ET operations (see Section 3.1), we have the following precondition for $wlock_{t,x_i}$:

$$\frac{1}{2}\left(g(\ldots, f_i(x_{i,initial} + (\sum_{t_j \in committed\_CUT(q)} max\_change_{t_j,x_i} + max\_change_{t,x_i})), \ldots)\right) -$$
$$\frac{1}{2}\left(g(\ldots, f_i(x_{i,initial} - (\sum_{t_j \in committed\_CUT(q)} max\_change_{t_j,x_i} + max\_change_{t,x_i})), \ldots)\right) \leq$$
$$result\_inconsistency\_limit_q$$

and the following precondition for $rlock_{q,x_i}$:

$$\frac{1}{2}\left(g(\ldots, f_i(x_{i,initial} + max\_change_{t,x_i}), \ldots) - g(\ldots, f_i(x_{i,initial} - max\_change_{t,x_i}), \ldots)\right) \leq$$
$$result\_inconsistency\_limit_q$$

In a similar manner, preconditions can be derived in case the other expressions for inconsistency are used.

5.3 Bounded Queries

We say that a function f is *bounded* if there is a maximum bound in the result of f. It is easy to see that we can calculate bounds on the inconsistency in the results of a query composed from bounded functions.

Example 5: Consider the following variation of Example 4. The query ET sums up all the deposits that are *not* greater than \$5,000. For this query, n=20; $g = \sum_{i=0}^{20}$; $f_i = ((x_i \leq 5000) \times x_i)$. The f_i's are not monotonic because when x_i increases from \$4999 to \$5001, f_i decreases from \$4999 to \$0. So the expressions derived for $result\_inconsistency$ in Section 5.2 do not apply.

It is easy to see that a serializable execution of the query ET should return \$49,990, since at any given time, exactly 10 accounts have balance $\leq$ \$5,000. It is also not difficult to see that for the above ET query, the smallest possible result is \$0 and the largest possible result is \$99,980.

Even though the the f_i's are not monotonic, we now show that it is possible to obtain bounds on the query results. Let min_f_i denote the smallest value of f_i for any value of

x_i in $(x_{i,min},\ x_{i,max})$ and let max_f_i denote the largest value of f_i for any value of x_i in $(x_{i,min},\ x_{i,max})$. Then as long as g is monotonic, the result of the query can lie between $g(min\_f_1, \ldots, min\_f_n)$ and $g(max\_f_1, \ldots, max\_f_n)$.

Let us return to Example 5. In this case,

$$\begin{aligned} \forall i = 1, \ldots, 10, x_{(i*2)-1,min} &= \$4,997. \\ \forall i = 1, \ldots, 10, x_{(i*2)-1,max} &= \$5,005. \\ \forall i = 1, \ldots, 10, x_{(i*2),min} &= \$4,995. \\ \forall i = 1, \ldots, 10, x_{(i*2),max} &= \$5,003. \end{aligned}$$

$min\_f_i = 0$ and max_f_i = \$5,000 and hence, the result of the query can lie between \$0 and \$100,000. Since the actual result of the query lies between \$0 and \$99,980, using the maximum and minimum posible f_i values leads to an overestimate of the inconsistency in the query results.

A generalization of bounded functions and monotonic functions is the class of functions *of bounded variation.* To avoid confusion for readers familiar with mathematical analysis, we follow closely the usual definition of these functions in compact metric spaces.

Definition 5 *If $[a, b]$ is a finite interval in a metric space, then a set of points*

$$P = \{x_0, x_1, \ldots, x_n\}$$

satisfying the inequalities $a = x_0 < x_1 < \ldots < x_{n-1} < x_n = b$ is called a partition *of $[a, b]$. The interval $[x_{k-1}, x_k]$ is called the k^{th} subinterval of P and we write $\Delta x_k = x_k - x_{k-1}$, so that $\sum_{k=1}^{n} \Delta x_k = b - a$.*

Definition 6 *Let f be defined on $[a, b]$. If $P = \{x_0, x_1, \ldots, x_n\}$ is a partition of $[a, b]$, write $\Delta f_k = f(x_k) - f(x_{k-1}), k = 1, 2, \ldots, n$. If there exists a positive number M such that*

$$\sum_{k=1}^{n} |\Delta f_k| \leq M$$

for all partitions of $[a, b]$, then f is said to be of bounded variation *on $[a, b]$.*

It is clear that all bounded functions are of bounded variation. In Example 5, $M = 5000$. Furthermore, all monotonic functions are also of bounded variation. This happens because for a monotonically increasing function f we have $\Delta f_k \geq 0$ and therefore:

$$\sum_{k=1}^{n} |\Delta f_k| = \sum_{k=1}^{n} \Delta f_k = \sum_{k=1}^{n} [f(x_k) - f(x_{k-1})] = f(b) - f(a) = M.$$

In general, for a function of bounded variation, the M bound can be used as an (over)estimate of result inconsistency given the interval $[a, b]$ caused by input inconsistency. However, the examples above show that what we need is to restrict the forms of ET queries such that tighter bounds on result inconsistency can be found without overly restricting the type of queries allowed.

5.4 Steady Queries

Let DS denote the set of distances defined by S_{DB} and DR the set of distances defined by R_f. We say that f is *steady* if for every $\epsilon \in DR, \epsilon > \epsilon_0 \geq 0$ we can find a $\delta \in DS, \delta > 0$ such that $|f(x) - f(x + \delta)| \leq \epsilon$. Steady functions on discrete metric spaces are analogous to continuous functions on compact sets. The definition is similar, except that we exclude a fixed number of small ϵ due to the discrete nature of S_{DB}. Informally, if $\epsilon < \epsilon_0$ we allow δ to be zero.

The importance of steady functions is that the application designer may specify a limit on the result inconsistency, *result_inconsistency_limit* (ϵ), and the TP system can calculate the limit on the imported inconsistency, *max_inconsistency* (δ), that guarantees the specified limit on the result inconsistency. Section 5.2 shows how this calculation can be done for monotonic functions. Note that every monotonic function can be steady with a convenient choice of ϵ_0. However, the smaller is the ϵ_0 the tighter is the bound on δ. In the following example, the bound is tight because $\epsilon_0 = 0$.

Example 6: Consider a query ET that returns the balance of a bank account. If an update is executing, say transferring some money into the account, then the query result inconsistency is equal to imported inconsistency and $\delta = \epsilon$.

For an example where ϵ_0 is large, consider Example 4. When an account balance is actually 5000, an input inconsistency of 1 may change the result by 5000. Therefore we have $\epsilon_0 = 5000$, since a smaller ϵ requires $\delta = 0$.

One way to handle such a situation is to reduce or eliminate the imported inconsistency in the data item that causes a large ϵ_0. For instance, suppose that $q = g(f_1(x_1), f_2(x_2))$ and that a large ϵ_0 is due to x_1. We should tighten the *import_limit* for x_1 and allow inconsistency only for x_2. Consider the following example which is a simple variation of Example 4.

Example 7: The query ET returns the checking account balance of customers that have savings accounts with balance greater than \$5,000. Note that in this example, x_1 refers to the savings account and x_2 to the checking account. In this case, we may specify *import_limit* = 0 for the savings account balance and *import_limit* = \$100 for the checking account balance. This way, we avoid the large ϵ_0 with respect to x_1 but maintain the tight control over result inconsistency since the function that returns the checking account balance is a steady function with $\epsilon_0 = 0$ (from Example 6).

Being able to calculate ϵ from δ and vice-versa are properties of ET queries that allow the system to maintain tight bounds on result inconsistency. Functions of bounded variation and steady functions are abstract classes of functions that have these properties. More work is needed to characterize these functions defined on discrete metric spaces to facilitate the appropriate classification of queries.

6 Related Work

6.1 General Weak Consistency Criteria

Several notions of correctness weaker than SR have been proposed previously. Gray's different degrees of consistency [11] is an example of a coarse spectrum of consistency. Degree

3 consistency is equivalent to SR, but degree 2 consistency trades off reduced consistency for higher concurrency for queries. Since degree 2 allows unbounded inconsistency, degree 2 queries become less accurate as a system grows larger and faster. In general, ESR offers a much finer granularity control than the degrees of consistency.

Garcia-Molina and Wiederhold [10] have introduced the *weak consistency* class of read-only transactions. In contrast to their WLCA algorithm, ESR is supported by many divergence control methods [26]. Similarly, Du and Elmagarmid [7] proposed quasi-serializability (QSR). QSR has limited applicability because of the local SR requirements despite unbounded inconsistency. Korth and Speegle [15] introduced a formal model that include transaction pre-conditions and post-conditions. In contrast, ESR refers specifically to the amount of inconsistency in state space.

Sheth and Rusinkiewicz [23] have proposed *eventual consistency*, similar to identity connections introduced by Wiederhold and Qian [25], and *lagging consistency*, similar to asynchronously updated copies like quasi-copies [1]. They discuss implementation issues in [21, 22]. In comparison, ESR achieves similar goals but has a general approach based on state space properties and functional properties. Barbara and Garcia-Molina [2] proposed *controlled inconsistency*, which extends their work on quasi-copies [1]. Their demarcation protocol [3] can be used for implementing ESR in distributed TP systems. ESR is applicable to arithmetic and other kinds of consistency constraints.

6.2 Asynchronous Transaction Processing

Garcia-Molina et al. [9] proposed *sagas* that use semantic atomicity [8] defined on transaction semantics. Sagas differ from ESR because an unlimited amount of inconsistency (revealed before a compensation) may propagate and persist in the database. Levy et al [17] defined *relaxed atomicity* and its implementation by the Polarized Protocol. ESR is defined over state space properties and less dependent on application semantics.

An important problem in asynchronous TP is to guarantee uniform outcome of distributed transactions in the absence of a commit protocol. Unilateral Commit [13] is a protocol that uses reliable message transmission to ensure that a uniform decision is carried out asynchronously. Optimistic Commit [16] is a protocol that uses Compensating Transactions [14] to compensate for the effects of inconsistent partial results, ensuring a uniform decision. Unilateral Commit and Optimistic Commit can be seen as implementation techniques for ESR-based systems.

Another way to increase TP concurrency is Escrow Method [18]. Like the escrow method, ESR also uses properties of data state space, but ESR does not rely on operation semantics to preserve consistency. Similarly, *data-value partitioning* [24] increases distributed TP system availability and autonomy. ESR can be used in the modeling and management of escrow and partitioned data-values.

7 Conclusions

Previous ESR papers have focused on the practical aspects of ESR: its applications (e.g., asynchronous replication [19] and autonomous execution of distributed transactions [20]) and algorithms (e.g., divergence control [26] that guarantee ESR in transaction processing systems).

In this paper, we have attempted to examine epsilon serializability (ESR) from a formal perspective. We showed precisely how ESR is related to SR, for example, which conflicts considered by SR are ignored by ESR. A conflict based specification of ESR using the ACTA formalism was employed to bring out the differences between SR and ESR. One of the main results of the paper concerns the formulae that express the inconsistency in the data values read by a query. From these we derived the preconditions, that depend on the data values and the import limits, for read, write, lock, and unlock operations to be performed. In other words, from a precise definition of ETs and ESR, we have been able to derive the behavioral specifications for the necessary transaction management mechanisms.

One other way in which this paper extends the previous work on ESR is related to the derivation of expressions for the inconsistency of the *results of queries*. We showed that since arbitrary queries may produce results with large inconsistency, it is important to restrict ET queries to have certain properties that permit tight inconsistency bounds. Clearly, more work is needed in this area since generality of the queries has to be traded off against the tightness of the result inconsistency. Among the other active topics of research is the treatment of general ETs that both import and export inconsistency.

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] D. Barbara and H. Garcia-Molina. The case for controlled inconsistency in replicated data. In *Proceedings of the Workshop on Management of Replicated Data*, pages 35–42, Houston, November 1990.

[3] D. Barbara and H. Garcia-Molina. The demarcation protocol: A technique for maintaining arithmetic constraints in distributed database systems. Technical Report CS-TR-320-91, Computer Science Department, Princeton University, April 1991.

[4] P. Chrysanthis and K. Ramamritham. A formalism for extended transaction models. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, September 1991.

[5] P.K. Chrysanthis and K. Ramamritham. ACTA: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of SIGMOD Conference on Management of Data*, pages 194–203, June 1990.

[6] P.K. Chrysanthis and K. Ramamritham. ACTA: The Saga continues. In Ahmed Elmagarmid, editor, *Transaction Models for Advanced Applications*. Morgan Kaufmann, 1991.

[7] 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.

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

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

[10] H. Garcia-Molina and G. Wiederhold. Read-only transactions in a distributed database. *ACM Transactions on Database Systems*, 7(2):209–234, June 1982.

[11] 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.

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

[13] 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.

[14] 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.

[15] H.F. Korth and G.D. Speegle. Formal model of correctness without serializability. In *Proceedings of 1988 ACM SIGMOD Conference on Management of Data*, pages 379–386, May 1988.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] A. Sheth and P. Krishnamurthy. Redundant data management in Bellcore and BCC databases. Technical Report TM-STS-015011/1, Bell Communications Research, December 1989.

[22] 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.

[23] A. Sheth and M. Rusinkiewicz. Management of interdependent data: Specifying dependency and consistency requirements. In *Proceedings of the Workshop on Management of Replicated Data*, pages 133–136, Houston, November 1990.

[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] G. Wiederhold and X. Qian. Modeling asynchrony in distributed databases. In *Proceedings of the Third International Conference on Data Engineering*, pages 246–250, February 1987.

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