Divergence Control for Distributed Database Systems

Calton Pu[1], *Wenwey Hseush* and *Gail E. Kaiser*

Department of Computer Science
Columbia University
New York, NY 10027

Kun-Lung Wu and *Philip S. Yu*

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

Abstract

This paper presents distributed divergence control algorithms for epsilon serializability for both homogeneous and heterogeneous distributed databases. Epsilon serializability allows for more concurrency by permitting non-serializable interleavings of database operations among epsilon transactions. We first present a strict 2-phase locking divergence control algorithm and an optimistic divergence control algorithm for a homogeneous distributed database system, where the local orderings of all the sub-transactions of a distributed epsilon transaction are the same. In such an environment, the total inconsistency of a distributed epsilon transaction is simply the sum of those of all its sub-transactions. We then describe a divergence control algorithm for a heterogeneous distributed database system, where the local orderings of all the sub-transactions of a distributed epsilon transaction may not be the same and the total inconsistency of a distributed epsilon transaction may be greater than the sum of those of all its sub-transactions. As a result, in addition to executing a local divergence control algorithm in each site to maintain the local inconsistency, a global mechanism is needed to take into account the additional inconsistency.

Index terms: epsilon serializability, distributed divergence control, extended transaction models, distributed databases, heterogeneous transaction processing.

[1] Calton Pu is currently with the Dept. of Computer Science and Engineering, Oregon Graduate Institute, Beaverton, OR 97006.

1 Introduction

Epsilon Serializability (ESR) [13, 19] is a generalization of classic serializability (SR). ESR allows a limited amount of inconsistency in transaction processing. An *epsilon transaction* (ET) is a sequence of operations that maintain database consistency when executed atomically. However, an ET extends the standard notion of a "transaction" in the sense that it includes a specification of the amount of permitted inconsistency. For a typical case, a query ET (read-only) is allowed to view inconsistent data due to non-serializable interleavings of operations with concurrent update ETs, where an update ET may change the database state. Such non-serializable interleavings can increase transaction processing system performance through added query ET concurrency, while update ETs still preserve database consistency. With ESR, application programmers can specify the amount of inconsistency allowed for each ET.

As an example, assume that a bank manager is running a query ET to calculate the sum of all accounts to find out the current cash position. At the same time, customers are making deposits, withdrawals and transferring money from one account to another. Customers would be extremely unhappy if they see inconsistent account balances. However, the bank manager may accept a calculated sum (given in millions of dollars) that is "close enough", say within \$10,000 of the actual amount. That is, the query ET is allowed to view inconsistency of at most \$10,000. The execution history may be non-serializable, but the result is acceptable and useful. As long as the query ET result is within \$10,000 of a serializable query, the ET history is epsilon serializable.

An important assumption in ESR is the existence of a distance function and an associated regular geometry in the database state space, similar to a metric space. Fortunately, many of the real world database state spaces fulfill this requirement. For instance, integers and real numbers in banking, airline, and scientific databases are cartesian spaces that have a natural definition of distance function and the regular geometry of a metric space. With a distance function on a metric space, inconsistency associated with a transaction or a database can be measured quantitatively. Inconsistency associated with a transaction may be transient and may not cause inconsistency in the database. But, inconsistency associated with a database may remain permanent unless further actions are taken to restore consistency. In this paper, we only address transient inconsistency associated with transactions. We focus on the environments where query ETs may access inconsistent data through non-serializable interleavings with other update ETs, but update ETs are serializable among themselves. The same assumptions are also made in [12, 13, 14, 19], where [12, 14] describes ESR for autonomous transaction executions and as a general framework for transaction processing, [13] describes the application of ESR to replica control in distributed systems and [19] describes the algorithms to maintain ESR histories for centralized database systems.

The bounded inconsistency in ESR is automatically maintained by *divergence control* (DC) algorithms similar to the way in which serializability is enforced by concurrency control algorithms in classic transaction processing systems. Various divergence control algorithms for *centralized* transaction processing systems have been described in [19]. In this paper, we introduce divergence control algorithms for *distributed* transaction processing systems. In general, a distributed ET consists of many sub-ETs, each executing in one component database, or site. We formulate and provide solutions to the unique problems that arise in a distributed environment, including the maintenance of global serializability of the distributed ETs, the distribution of allowed inconsistency among the sub-ETs executing in the component databases, and the enforcement of bounded global inconsistency of a distributed ET from the local inconsistencies of all its sub-ETs.

In this paper, we present two classes of distributed divergence control (DDC) algorithms. The first class is designed for *homogeneous* distributed databases, while the second class is designed for *heterogeneous* distributed databases. For homogeneous distributed databases, we assume that the local orderings of sub-ETs are the same among all the component databases. We describe two distributed divergence control algorithms: a strict 2-phase locking distributed divergence control algorithm (S2PLDDC) and an optimistic distributed divergence control algorithm (ODDC) using weak locks. For both algorithms, the same local divergence control algorithm is executed in each site, and the bounded global inconsistency of a distributed ET can be enforced in a relatively simple way. For the first algorithm (S2PLDDC), the global inconsistency limit is first distributed to its sub-ETs before execution, and if needed, a sub-ET may ask, during runtime, for more limit from another sub-ET. For the second algorithm (ODDC), the global inconsistency limit is not distributed to the sub-ETs at the beginning of execution. Instead, each sub-ET optimistically uses the entire global inconsistency limit during its execution.

For heterogeneous distributed databases, the local orderings of all the sub-ETs of a distributed ET may not be the same, and hence the global orderings of distributed ETs may not be serializable even though all the sub-ETs are locally serializable. In this case, merely employing a local divergence control algorithm in each site and accumulating all the local inconsistencies are not sufficient to enforce the bounded global inconsistency for a distributed ET. Thus, a global mechanism is needed to guarantee ESR. We use the Superdatabase architecture in [15] as a general model and present a corresponding distributed divergence control algorithm based upon it.

The paper is organized as follows. Section 2 summarizes a semi-formal definition of ESR, outlines the general design methodology for centralized divergence control algorithms, introduces ESR-related concepts for distributed databases, and describes several practical applications that can benefit from ESR in a distributed database environment. Section 3 introduces two distributed divergence control algorithms, S2PLDDC and ODDC, for homogeneous distributed databases. The distribution of inconsistency among the sub-ETs executing in each site is discussed in detail. Section 4 presents the Superdatabase distributed divergence control algorithm for heterogeneous distributed databases. Section 6 discusses some advanced issues in the design of a general distributed divergence control algorithm. Section 7 summarizes related work.

2 Background and Motivation

2.1 Semi-Formal Definition of ESR

2.1.1 Epsilon Serializability

A formal ESR model has been discussed by Ramamrithan and Pu [16] based on the ACTA framework [**?**]. We briefly describe the formal model.

We say that two operations, a and b, *conflict*, denoted by $conflict(a, b)$, if both operate on the same data item and one of them is a read operation. Assume that we are given a set of transactions T. For two different transactions $t_i \in T$ and $t_j \in T$, we say that t_i *conflicts with* t_j, denoted by $t_i\ \mathcal{C_{SR}}\ t_j$, if an operation a issued by t_i and an operation b issued by t_j conflict, and a precedes b in the history. For a transaction $t \in T$, we say that $(t\ \mathcal{C_{SR}}^*\ t)$ is a *conflict cycle*, where $\mathcal{C}^*_{\mathcal{SR}}$ is the transitive-closure of $\mathcal{C_{SR}}$. A history over T is *serializable*, if and only if there does not exist a conflict cycle.

A formal definition of epsilon serializability is established by assuming the existence of a safe condition for a transaction t (denoted by $Safe(t)$):

$$\begin{aligned} Fuzziness_t^{import} &\leq Limit_t^{import} \\ Fuzziness_t^{export} &\leq Limit_t^{export}, \end{aligned}$$

$Fuzziness_t^{import}$ is the import fuzziness of t, which could be an approximation of the "actual" import fuzziness by a mechanism (e.g., a divergence control method). $Fuzziness_t^{export}$ (the export fuzziness of t) is defined in the similar way. $Limit_t^{import}$ is the import fuzziness limit of t and $Limit_t^{export}$ is the export fuzziness limit of t. Please refer to Section 2.2 for import and export fuzziness.

For two transactions $t_i \in T$ and $t_j \in T$, we say that t_i *epsilon-conflict with* t_j, denoted by $t_i\ \mathcal{C_{ESR}}\ t_j$, if $t_i\ \mathcal{C_{SR}}\ t_j$ and $\neg Safe(t_j)$. That is, t_i conflicts with t_j and the safe condition of t_j is violated after t_j invokes its operation. A conflict cycle that does not contain a $\mathcal{C_{ESR}}$ edge is a *safe* conflict cycle. Otherwise, it is an *unsafe* conflict cycle. A history is *epsilon serializable*, if and only if there does not exist an unsafe conflict cycle.

2.1.2 Distance Functions and Transaction Views

The objective of epsilon serializability is to control the amount of inconsistency in applications. The amount of inconsistency is measured in terms of a distance function defined over the database state space. $W(conflict(r, w))$, the weight of a conflict between a read r and a write w is a function that maps the conflict into a distance.

$$W(conflict(r, w)) \; = \; distance(s_{before}, \; s_{after})$$

That is, the weight of $conflict(r, w)$ is the distance between the state before the write (s_{before}) and the state after the write (s_{after}). Intuitively, the weight of a conflict is the potential fuzziness that can be introduced to the transactions due to the conflict.

The distance function is often defined under the semantics of data items. For example, it is natural for an application designer to choose the numerical difference function as the distance function for numerical data items. Assume that x_1 and x_2 are two states of data item x. The distance between two states is the difference of two values.

$$distance(x_1, \; x_2) \; = \; | \; x_1 \; - \; x_2 \; |$$

Assume that a read operation on x returns 5 and later a write operation changes x from 5 to 13. The weight of the read-write conflict is 8.

$$W(conflict(r, \; w)) \; = \; 8$$

For a second example, we describe a distance function on database states, which are constituted by multiple data item states. Assume that the database has three numerical data items, x, y and z, three checking accounts owned by three different customers, for example. A state is represented by a triple (x, y, z). We can define the distance function as follows:

$$distance((x_1, y_1, z_1), \; (x_2, y_2, z_2)) = |x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2|$$

So, the distance between a state (100,200,400) and a state (300,500,400) is 500 (i.e., 200+300+0). To move (100,200,400) to (300,500,400) requires at least two write operations, which may cause two read-write conflicts for a query transaction. The weight of the conflict on x is 200 and the weight of the conflict on y is 500.

Each transaction has a *view* of the database state (referred to as a *transaction view*), which is its knowledge about the current database state, collected piece by piece via the returned values of operations issued since the transaction started. A transaction view could be skewed beyond acceptance during the collection process, if there is no proper control applied. One major task in the ESR work is to design proper mechanisms to ensure acceptable transaction views while maintaining consistent database states.

Given a history H, the transaction view of a transaction t on H (at commitment), denoted by $\mathcal{TV}_t(H)$, is the set of returned values of the read operations issued by t. The transaction starts with an empty transaction view and accumulates states of individual data items while proceeding. A transaction view is a partial database state. We can extend the distance function on transaction views by treating each absent data item state as an "don't care state" (denoted by "*") that does not add to total distance. Such an extended transaction view is a database state. The distance between a transaction view and a consistent database state (e.g., produced by a serializable history) is an alternative definition of the transaction's imported inconsistency (also called $Fuzziness_t^{import}$ for transaction t).

2.1.3 First Lemma of Epsilon Serializability

The following lemma is applied only to the transactions that are concerned with import inconsistency.

Lemma 1 *A history H over a non-empty set of transactions T is* epsilon serializable, *iff* $\exists\ H^{SR}$, *a serial history consisting of all transactions in T, such that*

$$\forall\ t\ \in\ T, distance(\mathcal{TV}_t(H), \mathcal{TV}_t(H^{SR}))\ \leq\ \epsilon\text{−spec}_t.$$

First, we prove the only–if part: Given a non-empty set of transactions T, if there is no unsafe conflict cycle in history H, then $\exists\ H^{SR}$, such that $\forall\ t\ \in\ T, distance(\mathcal{TV}_t(H), \mathcal{TV}_t(H^{SR}))\ \leq\ \epsilon\text{−spec}_t$.

The proof is by contradiction. Let us assume the negation of the thesis. Given T and H, $\forall\ H^{SR}\ \exists\ t\ \in\ T$, such that $distance(\mathcal{TV}_t(H), \mathcal{TV}_t(H^{SR}))\ >\ \epsilon\text{−spec}_t$. Since $distance(\mathcal{TV}_t(H), \mathcal{TV}_t(H^{SR}))$ is an alternative definition of $Fuzziness_t^{import}$, and $\epsilon\text{−spec}_t = Limit_t^{import}$, we have:

$$Fuzziness_t^{import} = distance(\mathcal{TV}_t(H), \mathcal{TV}_t(H^{SR}))\ >\ \epsilon\text{−spec}_t = Limit_t^{import},$$

which contradicts the safety condition, i.e., our hypothesis. $\Box$.

Second, we prove the if part. If $\exists\ H^{SR}$, such that $\forall t \in T$, $distance(\mathcal{TV}_t(H), \mathcal{TV}_t(H^{SR})) \leq \epsilon\text{−spec}_t$, then there are no unsafe conflict cycles. Since $distance(\mathcal{TV}_t(H), \mathcal{TV}_t(H^{SR})) = Fuzziness_t^{import}$ and $\epsilon\text{−spec}_t = Limit_t^{import}$, we have the safety condition by definition. $\Box$.

Intuitively, the theorem says that a serial history H^{SR} can be constructed that maintains the distance between it and each transaction view. A constructive proof depends on the mechanism that bounds the distance and is similar to traditional concurrency control algorithm proofs. For example, a constructive proof that two-phase locking divergence control produces histories with such H^{SR} is similar to the one in [5].

2.2 Centralized Divergence Control Algorithms

We briefly summarize the general design methodology of centralized divergence control algorithms in [19]. Import fuzziness (Z_{import}) is the amount of inconsistency that an ET "sees" from the database or other ETs through operations. Export fuzziness (Z_{export}) is the amount of fuzziness that an ET introduces to other ETs. When a conflict between a read operation in a query ET and a write operation in an update ET causes the execution to diverge from a serial history, we say the query ET *imports* fuzziness from the update ET and the update ET *exports* fuzziness to the query ET. By definition, the amount of imported or exported fuzziness is at most equal to the weight of the conflict. Since we focus only on environments where database consistency cannot be compromised, we do not allow an update ET to import fuzziness from another update ET. Thus, an import fuzziness limit is specified for each query ET, and an export inconsistency limit is specified for an update ET. We consider that the export limit of a query ET and the import limit of an update ET are set to zero.

By keeping track of the amount of potential inconsistency for each ET, a divergence control algorithm guarantees that the result of any ET will be within the ϵ-spec from a serializable database state. For example, if a query ET returns an answer that differs from a serializable result by \$10,000, it is said that the query has an inconsistency of \$10,000. To control the inconsistency limit during execution, an *ImpCount* is associated with a query ET, accumulating the total fuzziness imported so far to the query ET; an *ExpCount* is associated with an update ET, accumulating the total fuzziness exported so far from this update ET.

The design of centralized divergence control methods follows a two-stage methodology: extension and relaxation. In the first stage, existing concurrency control methods are extended, by identifying the places where the concurrency control methods detect non-serializable, conflicting operations. In the second stage, the extension is relaxed, by using fuzziness accumulators ($ImpCount$ and $ExpCount$) to allow more concurrency for ETs within the specified limits. The underlying idea is that (conflict-based) concurrency control methods must be able to identify the non-serializable conflicts and prevent a cycle from forming in the *serialization graph* [**?**]. The extension stage isolates the identification part of concurrency control and the relaxation stage modifies the cycle prevention part so as to permit limited inconsistencies.

The extension of classic 2-phase locking concurrency control to 2-phase locking divergence control (2PLDC) results in the lock compatibility shown in Table 1, where Q^{ET} is a query ET, U^{ET} is an update EY, *Ql* represents a read lock by a query ET, *Rl* a read lock by an update ET, and Wl a write lock by an update ET. In this table, columns represent locks held and rows locks requested. The squares marked AOK are always compatible, dashes are always incompatible. The right lower corner of the table is the same as in the classic 2PL concurrency control, so update ETs

	Q^{ET}	U^{ET}	
	Ql	Rl	Wl
Ql	AOK	AOK	LOK–1
Rl	AOK	AOK	—
Wl	LOK–2	—	—

Table 1: Lock Compatibility for 2PLDC.

are guaranteed to be consistent with respect to each other. In the case of R/W and W/R conflicts between Q^{ET} and U^{ET} (LOK–1 and LOK–2), the fuzziness incurred is accumulated in both ETs. LOK–1 allows query ETs to read uncommitted data (i.e., write–read conflicts), while LOK–2 allows update ETs to overwrite data that other query ETs are reading (i.e., read–write conflicts). When a conflict is detected in either LOK-1 or LOK-2, a query ET adjusts its $ImpCount$ by adding the weight of the conflict into its $ImpCount$.

$$ImpCount \ := \ ImpCount \ + \ W(conflict)$$

Also, an update ET adjusts its $ExpCount$ by adding the weight of the conflict to its $ExpCount$

$$ExpCount \ := \ ExpCount \ + \ W(conflict)$$

$ImpCount$ and $ExpCount$ are both potential fuzziness estimated by the divergence control method. They are greater than or equal to the actual fuzziness, since the potential fuzziness is calculated regardless of whether a conflict cycle is formed. If the potential fuzziness is below each ET's specification, the lock is granted. Otherwise, the requesting query ET or update ET is blocked until the lock is released. For example, when a query ET requests a Ql lock under LOK–1, the $ImpCount$ of the query ET and the $ExpCount$ of the conflicting update ET are both checked to see if a granting of the lock would cause either of them to exceed its limit. If yes, the requesting query ET is blocked. Otherwise, the lock is granted. In general, a centralized divergence control guarantees two following conditions:

$$Z_{import} \leq ImpCount \leq ImpLimit$$
$$Z_{export} \leq ExpCount \leq ExpLimit$$

We have used $Z_{import}$and Z_{export}as the *actual* fuzziness, and $ImpCount$ and $ExpCount$ as the *estimated* fuzziness in this section to show the difference between the theoretical view point and the system view point. One addresses the issue of correctness and the other one address the issue of efficiency. However, in the rest of the paper, we do not distinguish the actual fuzziness from the estimated fuzziness. We simply use $Z_{import}$and $Z_{export}$for both cases, assuming that the estimated fuzziness is equal to or greater than the actual fuzziness.

2.3 Applications of ESR

One class of applications where ESR seems particularly desirable involves *rapidly changing data*. This class of applications can be characterized by: (1) enormous amounts of data; (2) sharing of data among large numbers of simultaneous users (human or program); (3) frequent changes to data outside the control of the system; (4) soft real-time constraints, where updates must be accepted within a time interval or they will be missed; and (5) the acceptance by application designers that decisions may be based on obsolete or inconsistent data [9].

The purpose of distributed divergence control is to limit the obsolescence or inconsistency seen by query ETs to the degree that can be tolerated by the application. Changes that may come from outside the system may not be encapsulated in traditional transactions. In these cases, we treat each individual data item update as a complete update ET, in order to guarantee serializability among update ETs. Further, the inherent ability to degrade gracefully as time constraints are not met permits query ETs that have exceeded their limits to not always be aborted. But instead, the excess degree of fuzziness may be reported and the decision as to how to regard the query result (with some excess fuzziness) can be made by the user.

Two major application areas for rapidly changing data are on-line financial decision support systems, such as automated portfolio managers [18], and automated network management systems for high-speed networks [10]. In the former case, a large volume of updates come from outside the system itself, e.g., from the stock exchange wire. The portfolio managers make financial decisions based on monitoring of trends over time as well as recently reported prices. In the latter case, the majority of the traffic on the network is externally generated, and the job of the network management is to query sensors in order to detect faults and performance bottlenecks, and allocate resources to alleviate or repair problems. In both cases, the age of data and the deviation of the reported data values from the actual data values is of great concern.

Consider a portfolio manager involved in a relatively lengthy computation, say five minutes, to determine the current value of a large portfolio or to decide whether to make substantial changes in cash holdings versus stock/option investments. The incremental changes to prices during this computation may or may not affect the decision reached by the manager. For example, if there are repeated quarter-point increases or decreases in prices with little or no change in the overall value of the portfolio, the result would be only slightly different if recomputed based on the new prices (and the new values themselves are unlikely to be stable). The user understands this general inability to obtain exact information, and is willing to put up with a small discrepancy in order to increase performance, and to avoid perhaps numerous repeats of the query triggered by continuing changes. ESR provides a significant advantage over serializability in such cases. The allowable degree of fuzziness could be stated on a per-price basis for each ET. However it would be desirable for this application to modify the divergence control algorithms slightly to use the actual values rather than magnitudes of fuzziness, since numerous tiny changes up and down are tolerable but a single large change may not be.

As an example, assume that a query reads the stock prices from a database to calculate the value of a portfolio of 500 different stocks. During the execution, another update transaction can overwrite the stock prices that have been read by the query. However, the calculated portfolio value may be fuzzy. To control inconsistency, an ϵ-spec is specified for the query to allow a limited amount of price change. Assume *ImpLimit* for the query is $1\frac{1}{2}$, i.e., the maximum total price change allowed is $1\frac{1}{2}$. During the execution, the price of IBM drops $\frac{1}{4}$, MTEL drops $\frac{1}{4}$, THDO drops $\frac{1}{4}$, CYE drops $\frac{1}{4}$, DEC drops $\frac{1}{8}$ and ORCL rises $\frac{1}{4}$. The total price change is $1\frac{3}{8}$. The query is allowed to continue since the total price change does not exceed $1\frac{1}{2}$. Assume the next concurrent transaction is to update a price drop of $\frac{1}{4}$ for SYBS. This update, if allowed, would have caused the import fuzziness of the query to exceed $1\frac{1}{2}$. Thus, the action taken would be to abort the query and restart it, since the portfolio value is inaccurate beyond the allowable limit.

A network manager agent has similar characteristics. Since computations in this domain are unlikely to be of long duration, notifications of changed sensor readings can be considered and acted upon frequently. A primary concern here is age of data as much as change in value, since a high performance network must react extremely quickly to re-allocate resources as needed to buffer increased traffic over congested links and to reroute traffic away from failed links. Otherwise, packets will be lost, resulting in repeated traffic shortly thereafter, and user satisfaction will decrease. Queries that miss a few additional packets or a new failure point may not matter, since a new query will be issued in a few seconds and it will catch any additional problems. Serializability is unacceptable in this context, since the fuzziness permitted by ESR is required by the real-time nature of the application domain. Here, the allowable degree of fuzziness can be dynamically determined by the application for specific management agents or sensors based on recent trends, with a computed ϵ-spec provided as a parameter to each query ET as it is issued.

As an example, assume that the status of a network of 1,000 nodes and 200,000 links is maintained in a database. A query reads the status of the network from the database to calculate the best routing configuration. Network status may be changed asynchronously during the query execution. However, the calculated routing configuration may still be considered correct if there are less than 100 status changes during the query execution. In this case, the import fuzziness limit (ImpLimit) can be specified as 100. As a result, so long as the number of actual status changes is less than or equal to 100 during the query execution, the calculated routing configuration can be considered correct.

2.4 ESR In Distributed Databases

A distributed database is composed of several component databases or sites. A distributed ET consists of one or more sub-ETs, with each sub-ET on one site. For homogeneous distributed databases, our distributed transaction model is similar to the traditional model, such as the R* [11], where the local orderings of all sub-ETs are the same. However, for heterogeneous distributed databases, our model is similar to the one used in the Superdatabase [15], where the local orderings of all sub-ETs may be different. We designate a special sub-ET to be the coordinator as the collector of results and the coordinator of the commit protocol. We assume that each distributed ET has one ϵ-spec, i.e., only input one parameter for each ET. For simplicity, there is no shared data between sites (database is fully distributed), and there is no data replication.

Let local fuzziness, denoted by Z^{local}, refer to the fuzziness detected and maintained locally for a sub-ET by a centralized divergence control algorithm, and total fuzziness, denoted by Z^{total}, be the fuzziness of a distributed ET. In the simplest case, to be discussed in Section 3, the total fuzziness of a distributed ET is equal to the sum of the local fuzziness of all its sub-ETs:

$$Z^{total} = \sum_{sub-ET} Z^{local}.$$

This simple case corresponds to the *rigorous schedules* in [3], or the *commitment ordering* in [17], where the local orderings of sub-ETs are the same among all the sites. However, although all the sub-ETs may be locally serializable in each site, the global orderings of distributed ETs in general may not be serializable, especially in a heterogeneous distributed database system. This is a consequence of serializability being a global property [8]; the union of local serializability does not guarantee global serializability. Thus, the total fuzziness of a distributed ET may be greater than the sum of local fuzziness of all its sub-ETs.

Our discussions of distributed divergence control algorithms will proceed from simple cases to more complex ones. Section 3 considers only the simplest cases. Section 4 takes into consideration that the local orderings of sub-ETs in different sites may not be the same, thus making the distributed ET globally non-serializable.

3 Divergence Control Algorithms For Homogeneous Distributed Databases

In this section, we describe a strict two-phase locking divergence control algorithm (S2PLDDC) and an optimistic validation divergence control algorithm (ODDC) using weak locks. One important property for both algorithms is that the local commit orderings of sub-ETs are the same among all sites.

3.1 A Common Lemma

We first derive a common lemma which serves the common base for both algorithms (S2PLDDC and ODDC).

Lemma 2 *The total fuzziness of a distributed ET is at most equal to the sum of the local fuzziness of all its sub-ETs if the local serialization orderings of sub-ETs are the same among all sites. It is formally expressed as follows:*

$$Z^{total} \leq \sum_{sub-ET} Z^{local}$$

Proof: By contradiction. Assume that, given a global history over a set of distributed ETs, T, the total fuzziness of a distributed ET, $t \in T$, is greater than the sum of the local fuzziness of all its sub-ETs.

$$Z^{total} > \sum_{sub-ET} Z^{local} \tag{1}$$

According to the triangle inequality of the database metric space, we know that, if local divergence control algorithms have detected all conflicts, there must exist H^{SR}, a serial history over T, such that

$$\sum_{sub-ET} Z^{local} \geq distance(\mathcal{TV}_t(H), \mathcal{TV}_t(H^{SR})).$$

The distance between the transaction view of t on H and the transaction view of t on a serial history is actually the total fuzziness. This contradicts the above assumption (1). So, we know the local divergence control algorithms do not detect all conflicts. If all sub-ETs agree on one serialization order, we should be able to find H^{SR} such that the local divergence control algorithms detect all conflicts. There must exist some global conflicts due to the disagreement on the serialization order among some sub-ETs. □

Without loss of generality, we say the total fuzziness is the sum of local fuzziness (we over-estimate total fuzziness).

$$Z^{total} = \sum_{sub-ET} Z^{local}$$

3.2 Strict 2-Phase Locking Distributed Divergence Control Algorithm

The first algorithm is *strict 2-phase locking distributed divergence control* (S2PLDDC). S2PLDDC consists of two components: a centralize strict 2-phase locking divergence control algorithm running at each site and a commit protocol that guarantees the agreement of every site the same transaction outcome as well as the same transaction orderings by synchronizing the global lock point. A centralized strict 2-phase locking divergence control ensures that locks held by a sub-ET would not be released until the ET's commit time.

Let *ExpLimit* and *ImpLimt* be the global export ϵ-spec of an update distributed ET and the global import ϵ-spec of a query distributed ET, respectively. $Z^{local}_{export}$and $Z^{local}_{import}$are the corresponding local export and import fuzziness, respectively, estimated by the local 2-phase locking divergence control.

Lemma 3 *A history is epsilon serializable if (1) the local commit orderings of sub-ETs are the same among all sites, and (2) the following two conditions hold for every ET.*

$$\sum_{sub-ET} Z^{local}_{export} \leq ExpLimit \tag{2}$$

$$\sum_{sub-ET} Z^{local}_{import} \leq ImpLimit \tag{3}$$

Proof: From the definition of ESR, the history is ESR if the following conditions hold.

$$Z^{total}_{export} \leq ExpLimit$$

$$Z^{total}_{import} \leq ImpLimit$$

Thus, from Lemma 2, condition 2 and 3 are the sufficient conditions for ESR. □

To guarantee ESR, we must enforce conditions 2 and 3 for all distributed ETs. The first step of the S2PLDDC algorithm is to distribute the global ϵ-spec to the sub-ETs. For simplicity we initially divide the inconsistency limit evenly among all the sub-ETs. Section 3.3 presents approaches to dynamically distributing the inconsistency limit to sub-ETs. We refer to the ϵ-spec for a sub-ET as *LocalExpLimit* (local export fuzziness limit) and *LocalImpLimit* (local import fuzziness limit), respectively. Thus,

$$LocalExpLimit = \frac{ExpLimit}{|sub-ETs|};$$

$$LocalImpLimit = \frac{ImpLimit}{|sub-ETs|},$$

where $|sub - ETs|$ denotes the total number of sub-ETs of a distributed ET. Due to the even split of ϵ-spec to each site, conditions 2 and 3 hold, as long as the following two conditions hold for every sub-ET:

$$Z^{local}_{export} \leq LocalExpLimit; \tag{4}$$

$$Z^{local}_{import} \leq LocalImpLimit. \tag{5}$$

The second step of the S2PLDDC algorithm is to execute a local strict 2-phase locking divergence control algorithm with its assigned *LocalExpLimit* and *LocalImpLimit*. The local divergence control algorithm is similar to the 2PLDC algorithm described in [19] and in Section 2.2 of this paper with an additional assumption that locks at any site are released only at the commit time of a distributed ET.

The calculation of Z^{local}_{export} and Z^{local}_{import} for a sub-ET can be done *locally*. For a local query sub-ET, the only source that contributes to its Z^{local}_{import} is its reading of inconsistent data from other sub-ETs in the local site. By assumption, a local update sub-ET causes no permanent inconsistent data in the database, but it can export fuzziness to other sub-ETs by overwriting the data that have already been read. A local update sub-ET can also export fuzziness, if other sub-ETs read uncommitted data from it and it is aborted. To bound the exported fuzziness of a distributed update ET or the imported fuzziness of a distributed query ET, each of its local sub-ETs applies the above algorithm based on conditions 4 and 5. If every sub-ET has its Z^{local}_{import} or Z^{local}_{export} under the assigned limits at commit time, the distributed ET is allowed to commit. This is due to conditions 2 and 3 and the fact that strict 2PL guarantees global serializability if all subtransactions are locally serializable (e.g., see [15]).

However, a sub-ET may exceed its local fuzziness limit. If any local fuzziness of a sub-ET exceeds the assigned local limit, it can request or negotiate more ϵ-spec from other sub-ETs. This is the step to be described in Section 3.3. If all the sub-ETs have reached their limits, the sub-ET either blocks or initiates an early abort.

As an example, assume that a distributed query ET with an *ImpLimit* of 30 is composed of three sub-ETs, q_1, q_2 and q_3. This *ImpLimit* is first evenly distributed among the three sub-ETs; q_1 with $ImpLimit_1 = 10$, q_2 with $ImpLimit_2 = 10$ and q_3 with $ImpLimit_3 = 10$. Each sub-ET operates under a centralized divergence control method at each site. Let us consider three cases. For the first case, assume that, at the commit time of the distributed ET, q_1 has $Z^{local}_{import1} = 8$, q_2 has $Z^{local}_{import2} = 2$ and q_3 has $Z^{local}_{import3} = 6$. In this case, all three sub-ETs can be allowed to commit since $Z^{local}_{import_1} \leq ImpLimit_1$, $Z^{local}_{import_2} \leq ImpLimit_2$ and $Z^{local}_{import_3} \leq ImpLimit_3$. For the second case, assume that, at a point prior to the commit time, q_1 has $Z^{local}_{import1} = 9$, q_2 has $Z^{local}_{import2} = 2$ and q_3 has $Z^{local}_{import3} = 3$. If an update transaction in q_1's site conflicts with q_1 and would export a fuzziness of 4 to q_1, then instead of aborting the entire query ET, q_1 may request more limit from q_2, since q_2 still has 8 unused. For the third case, assume that q_1 has $Z^{local}_{import1} = 9$, q_2 has

$Z^{local}_{import2} = 9$ and q_3 has $Z^{local}_{import3} = 9$. If an update transaction in q_1's site conflicts with q_1 and would export a fuzziness of 4 to q_1, then the query ET must either abort or wait until the update transaction commits.

Theorem 1 *The distributed strict 2-phase divergence control algorithm guarantees ESR.*

Proof: The local strict 2PLDC algorithm (pessimistically) calculates the maximum amount of inconsistency potentially caused by read/write operations [19]. Since each site maintains conditions 4 and 5 locally and the local serialization orderings of all sub-ETs are the same, when we add up the Z^{local}_{export} for all sites we have the safe conditions 2 and 3. By Lemma 3, the history is ESR. □

3.3 Dynamically Distributing Inconsistency Limits

When the fuzziness of a sub-ET exceeds its initially assigned limit, we have several choices. First, we can negotiate more ϵ-spec fuzziness quota from other sub-ETs. This alternative is discussed in more detail below. Second, we can fall back to serializable execution for the remaining part of the sub-ET. This means blocking for lock-based divergence control and aborting for optimistic divergence control. Whether the ϵ-spec of a sub-ET was deduced by the system or individually specified by application designers, we simply follow the action prescribed by the centralized divergence control algorithm. The choice of whether to use a negotiation process is a policy issue, just like the choice of which negotiation algorithm. The trade-offs inherent in these choices are a subject of future research.

Nevertheless, we describe a negotiation algorithm as an existential proof that such an algorithm can be incorporated into a distributed divergence control. Our choice is the demarcation protocol described in [1]. The demarcation protocol was designed to maintain global arithmetic constraints among sites, by establishing safe limits as "lines drawn in the sand" for updates. It provides a way to change these limits dynamically and asynchronously. From the viewpoint of the demarcation protocol, conditions 2 and 3 are just two arithmetic constraints. Given the definition of ϵ-specs, we always have an arithmetic constraint of the inequality kind.

For each sub-ET, its LocalExpLimit and LocalImpLimit act as the safe limits, indicating that the sub-ET can continue to increment Z^{local}_{export} and Z^{local}_{import} as long as conditions 4 and 5 hold. When one constraint is about to be violated, a negotiation process is triggered. For each type of limit, each sub-ET maintains a table of N entries locally (N is the number of sub-ETs), indicating the allowed but unused fuzziness in other sub-ETs. For example, the unused ϵ-spec is the difference of LocalExpLimit and Z^{local}_{export}. The table is updated asynchronously.

The negotiation process starts if the local table indicates that there is unused fuzziness in other sub-ETs; otherwise the sub-ET blocks or aborts. Each round of negotiation begins with sending a request for unused fuzziness to the sub-ET that (indicated by the local table) has the most unused fuzziness. The request message also carries the information of the sending sub-ET's currently unused fuzziness. The receiving sub-ET first updates the entry of the sending sub-ET in its local table, and checks whether it has enough unused fuzziness for the request. If positive, it decides how much of unused fuzziness can be given away, and sends that information in a reply message to the requesting sub-ET. The requesting sub-ET updates its limit and the entry of the replying sub-ET. If the requested sub-ET does not have enough unused fuzziness, it sends a reject message back. The reject message also carries the information of its currently unused fuzziness. This allows the requesting sub-ET to update its local table. The process continues until either (1) one message returns with some unused fuzziness, or (2) no sub-ET, indicated in the local table, has enough unused fuzziness.

Consider an example of a distributed ET with three sub-ETs, t_1, t_2 and t_3 with a global ϵ-spec of 15. When the distributed ET starts, it partitions the ϵ-spec evenly and distributes them to each sub-ET. Consider the case when t_1's Z^{local}_{import} is 5. At the time that the fuzziness of t_1 is about to exceed its limit (from 5 to 6), t_2's fuzziness is 5 and t_3's fuzziness is 2. Suppose t_1's table shows t_2 has 5 left. t_1 sends a request (for 1 unit of fuzziness) message to t_2. After receiving the message, t_2 first updates its local table for t_1, and sends a reject message back. t_1 then updates the entry of t_2 in its local table, and sends another request message to t_3. t_3 updates the entry for t_1 and sends back a reply message. After receiving the allocated fuzziness, t_1 updates the entry for t_3 and continues to execute.

It has been shown in [1] that the loss of messages does not cause the constraints to be violated, even though it may decrease performance. Further improvements to increase performance, reliability, availability, or autonomy of demarcation protocol are possible but they are beyond the scope of this paper.

3.4 Optimistic Distributed Divergence Control Using Weak Locks

We start by summarizing the centralized optimistic validation divergence control algorithm (ODC) in [19], and then extend it to the optimistic distributed divergence control algorithm (ODDC). The centralized ODC method was implemented using *strong locks* and *weak locks*. Either one can be in a shared mode (a read-only lock) or an exclusive mode (an update lock). A weak lock is always compatible with another weak lock. However, a strong lock of exclusive mode is not compatible with any lock.

Under the centralized ODC, an ET (update or query) asynchronously requests a weak lock at the time it accesses a data item. If a strong lock of exclusive mode is held on the data item by another update ET, a query ET that is requesting a weak lock is marked as non-serializable (however, a requesting update ET will be marked for abort). In addition, the fuzziness counter of the query ET, $Z_{import}$is incremented by the magnitude of the conflicting update. Similarly, the fuzziness counter of the conflicting update ET, $Z_{export}$is incremented by the same amount. If either fuzziness counter exceeds its limit, the requesting query ET will be marked for abor. At certification time, a query ET not marked for abort completes successfully, even though it may be marked as non-serializable. If marked, an update ET is aborted. Otherwise, an update ET converts all of its weak locks of exclusive mode into strong locks and then commits. While converting a weak lock of a data item, all the query ETs holding a weak lock on the same data item are marked as non-serializable if their limits are not exceeded after the corresponding counters are incremented. Any query ET whose limit is exceeded is marked for abort. Any update ET holding a weak lock on the same data item during the lock conversion time is also marked for abort.

In the ODDC algorithm, a distributed update ET is assigned a limit for export fuzziness (ExpLimit) and a distributed query ET is assigned a limit for import fuzziness (ImpLimit). Each sub-ET executes the ODC algorithm locally. Unlike the S2PLDDC algorithm, the local ODC uses the entire distributed ET's global ϵ-spec and optimistically runs to completion, assuming that the other sub-ETs will not use their allowed fuzziness. If the ET's ϵ-spec is exceeded locally, then it is marked for abort. At certification time, all sub-ETs of the distributed ET are synchronized (by the coordinator). If any sub-ET is marked for abort, the distributed ET must be aborted. If not, it first sums the local fuzziness counters (i.e., $ImpCount$ or $ExpCount$) of its sub-ETs. If the total amount exceeds its ϵ-spec, the distributed ET (query or update) is aborted. Otherwise, all weak locks held by its sub-ETs of a distributed update ET are upgraded to strong locks, similar to the centralized ODC algorithm, and then the distributed update ET can commit. Notice that the mechanism presented in Section 3.3 for negotiating more fuzziness is not required in this case. Furthermore, the implementation of this ODDC algorithm requires no changes to the local ODC algorithm.

Theorem 2 *The distributed optimistic divergence control algorithm guarantees global ESR.*

Proof: The proof is similar to the one in S2PLDDC, since all local lock points are synchronized at the certification time when the weak locks are converted into strong locks. □

4 Divergence Control Algorithm For Heterogeneous Distributed Databases

4.1 Problem Description

We consider the case of cross-site cycles in the global serialization graph. In general, unless the local orderings of sub-ETs are the same among all the sites, the union of local ESR histories does not guarantee global ESR. To illustrate this problem, let us consider the following example. Assume that two distributed ETs, an update ET u and a query ET q, running on both site 1 and site 2. The two-phase locking divergence control algorithm is adopted at each local site. At site 1, sub-ET u_1 releases all its locks before q_1 starts acquiring any lock. The local counters for Z^{local}_{export} of u_1 and $Z^{local}_{import}$of q_1 is zero, since no conflicts are detected. At site 2, sub-ET q_2 releases all its locks before u_2 acquires any lock. The local counters is also zero. The local 2PLDC algorithm shows no fuzziness at either site, even though there exists inconsistency due to non-serializable ordering in the global history. (From site 1, u_1 precedes q_1, while from site 2, q_2 precedes u_2.) If the amount of inconsistency exceeds the ϵ-spec of any ET, the global history is not ESR.

Therefore, the total fuzziness is

$$Z^{total} = \sum_{sub-ET} Z^{local} + Z^{global}.$$

$Z^{local}$is maintained by the local divergence control algorithm, and Z^{global} is the global fuzziness calculated by the global divergence control mechanism, to be described in the next section.

4.2 Superdatabase Distributed Divergence Control

Superdatabase Concurrency Control

We use the Superdatabase architecture [15] as a general model for heterogeneous distributed databases. In the Superdatabase architecture, element databases are integrated together by a superdatabase [15]. The element databases maintain local serializability while the superdatabase controls global serializability. Each element database returns the ordering information about its subtransactions in the form of *Order-element* (*O-element*), one for each sub-ETs. The serialization order of each local sub-ET is represented by its O-element. The O-elements of all the sub-ETs of a distributed ET form the O-vector of the ET. By comparing the O-elements of two sub-ETs at the same site, the partial ordering of the two sub-ETs in an element database can be determined. Similarly, by examining the O-vectors of two ETs, the superdatabase can determine one of four relationships between the two ETs: (1) preceding, (2) succeeding, (3) independence and (4) non-serializable conflict, which indicates that the local orderings of sub-ETs are different. The first three indicate that the local orderings of sub-ETs are the same. If ET t_1 and ET t_2 have a relation of non-serializable conflict, we say that t_1 *non-serializable conflicts* with t_2,

Superdatabase Divergence Control

To implement a superdatabase distributed divergence control algorithm, each element database adopts a local divergence control algorithm. The distribution of limits using the demarcation protocol is the same as described in S2PLDDC. Since we assume a system where the union of local ESR may not guarantee global ESR, inconsistency may be introduced by the cross-site cycles in the global serialization graph (i.e., $Z^{global} > 0$). A global validation is needed to bound Z^{total} ($Z^{total} = \sum_{sub-ET} Z^{local} + Z^{global}$). At commit time, the superdatabase validates a global transaction to make sure that all its subtransactions have been "epsilon-serialized" in the same order. The validation is done by comparing the O-vector of the global transaction against those of committed transactions. In the case of serializability (i.e., ϵ-spec = 0), the validation prevents cycles in the global serializability graph. In the case where ϵ-spec is not equal to zero, the validation must verify whether the total fuzziness is bounded within ϵ-spec.

We assume the knowledge of the maximum fuzziness that an update ET can potentially export to a query ET. We refer to this as the *update magnitude* of an ET, or $MU(ET)$. For example, a back customer can withdraw at most $500.00 from a checking account per transaction. No matter how many query ETs are currently running, the maximum fuzziness the update ET can export to a query ET is $500.00. In general, the update magnitude of an ET can be estimated as the sum of the weights of potential conflicts.

When a sub-ET completes its computation, it sends its O-element and its local fuzziness (Z^{local}_{import} for a query sub-ET or Z^{local}_{export} for an update sub-ET) to the coordinator. The coordinator maintains a history of committed ETs. The entry of a committed update ET in the history contains its O-vector, Z_{export} and $ExpLimit$. Similarly, the entry of a committed query ET contains its O-vector, Z_{import}, and $ImpLimit$. The O-vector indicates the ET's global ordering and Z_{export} (or Z_{import}) indicates the amounts of fuzziness that the ET accumulates so far.

Given an ET t_i, after receiving this information from all sub-ETs of t_i, the coordinator first compares the received O-vector with a subset of committed ETs. This subset (referred to as H) contains the committed ETs that may still conflict with any active ET. Any committed ET in the history but outside of H must precede t_i or be independent (concurrent) to t_i. The details about the algorithm to derive H can be found in [15]. With respect to t_i, H can be further divided into three groups:

- Non-conflicting ETs (NC) : $t \in NC$, t_i precedes t, or t precedes t_i, or t_i is concurrent to t (i.e., the intersection between read and write sets is empty).

- Conflicting query ET (C_{query}) : $t \in C_{query}$, t is a query ET and t non-serializable conflicts with t_i.

- Conflicting update ET (C_{update}) : $t \in C_{update}$, t is an update ET and t non-serializable conflicts with t_i.

```
If t_i is a query ET do
  Begin
    For every t ∈ C_update do
      Begin
        If (Z_export(t) + MU(t) ≥ ExpLimit(t)) then Abort t_i;
      End
    If t_i is not aborted do
      Begin
        Z_import(t_i) = Σ_{sub-ET∈t_i} Z^local_import(sub − ET) + Σ_{t∈C_update} MU(t);
        If (Z_import(t_i) ≥ ImpLimit(t_i)) then Abort t_i;
        If t_i is not aborted do
          Begin
            For every t ∈ C_update do
              Begin
                Z_export(t) = Z_export(t) + MU(t);
              End
            Commit t_i;      /* commit/add into history */
          End
      End
  End
If t_i is an update ET do
  Begin
    If |C_update| > 0 then Abort t_i
    If t_i is not aborted do
      Begin
        For every t ∈ C_query do
          Begin
            If (Z_import(t) + MU(t_i) ≥ ImpLimit(t)) then Abort t_i;
          End
        If t_i is not aborted do
          Begin
            Z_export(t_i) = Σ_{sub-ET∈t_i} Z^local_export(sub − ET) + |C_query| × MU(t_i);
            If (Z_export(t_i) ≥ ExpLimit(t_i)) then Abort t_i;
            If t_i is not aborted do
              Begin
                For every t ∈ C_query do
                  Begin
                    Z_import(t) = Z_import(t) + MU(t_i);
                  End
                Commit t_i;      /* commit/add into history */
              End
          End
      End
  End
```

Figure 1: The Superdatabase Distributed Divergence Control validation

Global Validation For A Distributed Query ET

The global validation for a committing an ET t_i has two cases: one for update ETs and one for query ETs. The algorithm in pseudo-code is shown in Figure 1. In the case that the t is a query ET, two conditions are checked. First, the exported fuzziness of a committed update ET that conflicts with t_i may increase if t_i is committed. The following condition must hold.

$$Z_{export}(t) + MU(t) < ExpLimit(t), \quad \forall t \in C_{update}.$$

The maximum amount of fuzziness that t can export to t_i is $MU(t)$. Second, the import fuzziness of t_i must be less than its import limit. The maximum amount of import fuzziness can be calculated as follows.

$$Z_{import}(t_i) = \sum_{\mathrm{sub-ET} \in t_i} Z^{local}_{import}(\mathrm{sub-ET}) + \sum_{t \in C_{update}} MU(t).$$

The import fuzziness comes from two sources: (1) local import fuzziness of its sub-ETs and (2) global fuzziness due to reading inconsistent data from the conflicting update ETs. The maximum amount from the second source is $\sum_{t \in C_{update}} MU(t)$.

If both conditions are true, t_i can commit. It is then added to the history with its Z_{import}. Also, $Z_{export}(t)$ ($t \in C_{update}$) is incremented by $MU(t)$. If either condition is not true, t_i is aborted or restarted.

Global Validation For A Distributed Update ET

In the case that the committing ET t_i is an update ET, three conditions are checked. First, the number of ETs in C_{update} is zero, since we do not wish to introduce permanent inconsistency into the database. This follows from the assumptions made in divergence control work in this paper (sections 3.2, 3.4) and other previous papers [19, 16], that update ETs are serializable with respect to each other.

$$|C_{update}| = 0$$

Second, the import fuzziness of a committed query ET that conflicts with t_i may increase by $MU(t_i)$, if t_i is committed. The following condition must hold.

$$Z_{import}(t) + MU(t_i) < ImpLimit(t), \quad \forall t \in C_{query}.$$

The maximum amount of fuzziness that t can import from t_i is $MU(t_i)$. Third, the export fuzziness of t_i must be less than its export limit. The maximum amount of global export fuzziness can be calculated as follows.

$$Z_{export}(t_i) = \sum_{\mathrm{sub-ET} \in t_i} Z^{local}_{export}(\mathrm{sub-ET}) + |C_{query}| \times MU(t_i).$$

The export fuzziness comes from two sources: (1) local export fuzziness of its sub-ETs and (2) global fuzziness by conflicting with committed query ETs. The maximum amount from the second source is the amount that t_i exports to all $t \in C_{query}$, $MU(t_i)$ for each t.

If all three conditions are true, t_i can be committed. It is then added to the history with Z_{export}. Also, Z_{import} of t is incremented by $MU(t_i)$. If one of these conditions is not true, t_i is aborted. Several optimization techniques are being developed to tighten this bound for practical use.

Theorem 3 *The superdatabase distributed divergence control guarantees ESR.*

Proof: For the case that t_i is a query ET, the worst case to commit and add it into the history is (1) to introduce additional global fuzziness to a committed update ET that has cross-site read-write conflicts with t_i such that the update ET exceeds its export fuzziness limit and becomes a non-ESR ET, or (2) to import cross-site global fuzziness from these committed update ETs such that its total fuzziness (local fuzziness + global fuzziness) exceeds its limit. The maximum global fuzziness that can go to t_i is the sum of the update magnitude of all conflicting update ETs in C_{update}. In cases (1) and (2), the query ET's import fuzziness remains below its ϵ-spec. Therefore, the history is ESR.

For the second case that t_i is an update ET, we eliminate the write-write conflicts by definition. The worst case to commit and add t_i into the history is (1) to export additional global fuzziness to any committed query ET that has cross-site read-write conflicts with t_i such that the query ET exceeds its import fuzziness limit and becomes a non-ESR ET, or (2) to add cross-site global fuzziness such that its total fuzziness (local fuzziness + global fuzziness) exceeds it limit. The maximum global fuzziness that can go to t_i is the update magnitude that t_i exports to all conflicting query ETs. In cases (1) and (2), the update ET's export fuzziness remains below its ϵ-spec. Therefore, the history is ESR. □

5 Discussion

In this section, we provide a performance analysis for three distributed divergence control algorithms: S2PLDDC, ODDC and Superdatabase DDC. In general, a distributed divergence control algorithm involves two parts: local and global.

In S2PLDDC, a centralize 2-phase locking divergence control operates at each site to control local fuzziness. It detects lock conflicts based on a compatibility table in the same way which a 2-phase locking concurrency control detects lock conflicts. In addition, when a conflict is detected, $ImpCount$ and $ExpCount$ is first adjusted by adding the weight of the conflict and then verified against $ImpLimit$ or $ExpLimit$. The process requires only local computation, which is expected to be constant. However, the degree of concurrency increases due to the relaxed form of lock conflicts.

As shown previously, a bank manager runs a query ET to calculate the sum of accounts, while customers may run update ETs to transfer money from one account to another. Using strict 2-phase locking concurrency control, the manager's ET will be blocked every time a customer initiates an update ET that conflicts with the query ETs. Instead, using the divergence control, the bank manager's ET can proceed with a limited amount of lock conflicts.

S2PLDDC adopts two global algorithms: Demarcation protocol and a commit protocol. Demarcation protocol is used to dynamically redistribute ϵ-spec among sub-ETs. The protocol gives each site substantial autonomy for performing changes to its local fuzziness limit. The performance overhead is the cost of message passing needed for negotiation. For example, the bank manager's ET is composed of two sub-ETs running at two sites. Initially, \$5,000 is assigned to the import limit of each sub-ET. Assume that the first sub-ET is about to have its local fuzziness to exceed \$5,000. It sends out a message to request more limit from the second sub-ET. The second sub-ET may decide to give \$2,000 to the first sub-ET. After the first sub-ET increases its local limit by \$2,000, it can proceed with more fuzziness tolerated. The justification of Demarcation protocol design has been discussed in [1]. The second global algorithm is a commit protocol to guarantee the same transaction outputs and global orderings.

Even though S2PLDDC and ODDC have similar characteristics, but they have different implementations. ODDC needs to maintain the information of who holds a weak lock. Since each sub-ET is assigned a limit of the entire ET's ϵ-spec, it does not need to run Demarcation protocol to negotiate for more limit. It needs a certification process at the end to verify whether the sum of the fuzziness of all sub-ETs exceeds the ET's limit. This precess can be absorbed in the commit protocol and requires no additional messages.

In Superdatabase DDC, each local site may run different divergence control algorithms. The coordinator needs to maintain a history of committed ET, where the entry of a committed ET contains its O-vector, accumulated fuzziness and fuzziness limit. Finding a conflicting committed ET against a committing ET may be expensive, unless further optimization is done. However, Superdatabase DDC provides substantial autonomy in a heterogeneous environment.

6 Advanced Issues

In this paper, we have described several algorithms that control the amount of inconsistency in distributed transaction processing. These algorithms can be extended in several ways. In this section, we outline possible solutions for two advanced problems. Section 6.1 outlines the calculation of transaction output inconsistency given the bounded input inconsistency guaranteed by distributed divergence control algorithms. Section 6.2 describes how the ϵ-spec can be dynamically changed during the execution of a distributed ET.

6.1 Calculating the Output Inconsistency

In the previous sections, we have described algorithms that limit the amount of inconsistency in the input of an ET. In many applications the output inconsistency is a simple function of the input. For example, in an ET that sums up all the checking account balances, the output inconsistency equals the sum of input inconsistency. In general, however, the output of an ET is based on some computation of the data retrieved from either the database, end users or message exchange among its sub-ETs. Since inconsistency propagation depends on the semantics of each computation, it is non-trivial to estimate the output inconsistency from the input inconsistency [16]. Here, we only sketch a possible way to handle the problem.

If a data item Y is a function of a data item X, we say that Y is *consistency-dependent* on X. A consistency-dependency must be encapsulated in an ET. For a distributed ET, consistency-dependency across sites may be created by message exchange between two sub-ETs.

We only consider the simple case in which the direct graph of consistency-dependencies for sub-ETs is acyclic, where each node represents a sub-ET, and an arc represents a function dependency. There is an edge directed from sub-ET$_a$ to sub-ET$_b$, if a data item in sub-ET$_b$ is consistency-dependent on a data item in sub-ET$_a$. The graph of consistency-dependencies can either be provided by the programmer or compiler.

For each sub-ET, we maintain two counters: Z^{local}_{import} and $Z^{local}_{internal}$. Z^{local}_{import} contains the imported inconsistency that we have described in this paper so far. $Z^{local}_{internal}$ contains the inconsistency generated internally by the ET due to imported inconsistency. Each data item accessed by the sub-ET is assigned a magnifying factor M_f (determined by the application designer for the ET), which represents the multiplication factor on the inconsistency when computation is performed on that data item. Similarly, a M_f is assigned to each arc in the consistency dependency graph and the M_f is applied when data flow through that arc. For each of the distributed divergence control methods described in this paper, fuzziness is accumulated in Z^{local}_{import}. At the time of fuzziness accumulation, either a data item is being read or data is flowing through the consistency dependency graph. In either case, the appropriate M_f is multiplied to the inconsistency being propagated and accumulated in $Z^{local}_{internal}$. In the distributed divergence control methods, which bounds only input fuzziness, if Z^{local}_{import} exceeds the local limit, it can request more ϵ-spec from other sub-ETs using the demarcation protocol.

Since we do not guarantee a bound on output inconsistency, the content of $Z^{local}_{internal}$ will be only informational. It is possible, however, to use the content of $Z^{local}_{internal}$ to bound the output fuzziness the same way the content of Z^{local}_{import} is used to bound the input fuzziness.

6.2 Dynamic Change of ϵ-Spec

So far, we have assumed that the ϵ-spec does not change dynamically during ET execution. This is a limitation that can be relaxed using the demarcation protocol. In section 3.3 we have described the algorithm to negotiate for more fuzziness allowance when the current local limit has been reached. If during the execution of an ET, a sub-ET decides that its ϵ-spec needs to be changed, we have two cases. First, the local ϵ-spec may be increasing, and that introduces no problems. We can either retain the larger fuzziness locally, or use the demarcation protocol to spread it to other sites. Second, the local ϵ-spec may be decreasing. If the new ϵ-spec is still greater than the current Z^{local}_{import}, then we do not have any immediate problem. However, if the new ϵ-spec is lower than the current Z^{local}_{import}, then we need to stop the ET execution and obtain more fuzziness allowance from other sub-ETs immediately. If successful, then the sub-ET may proceed. If unsuccessful, then the ET should abort.

For ETs that change their ϵ-spec upwards through the ET execution, another plausible policy is to continue executing the sub-ETs optimistically, even if the ϵ-spec has been exceeded. There are several cases in which this policy may make sense. First, maybe the ET will eventually increase its ϵ-spec and make the current computation legal. Second, an improved distributed divergence control method may be able to make more accurate estimate of accumulated fuzziness, for example, canceling a positive error with a negative one. However, the ET programmer must take this into account, since the calculation of fuzziness propagation (as in Section 6.1) typically makes the assumption that the total fuzziness will not exceed the ET's ϵ-spec.

7 Related Work

This paper describes distributed divergence control algorithms in contrast to the centralized divergence control algorithms discussed in [19]. Unlike centralized divergence control algorithms, distributed divergence control algorithms must deal with the specific problems associated with distributed transaction processing systems, including the maintenance of global serializability of distributed ETs, the distribution of inconsistency limits among the sub-ETs in the component databases, and the enforcement of local inconsistency limits as well as global inconsistency limits.

A large body of literature exists concerning various approaches to effectively supporting concurrent transaction and query processing [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. These approaches typically employ multiple versions of data to eliminate R/W conflicts between update transactions and read-only queries. Queries access transaction-consistent, but maybe out-of-date, database states. In contrast, ESR does not require multiple versions of data, thus does not incur extra storage overhead. However, ESR allows query ETs to access inconsistent data in a controlled way.

In the area of real-time database, Kuo and Mok developed a notion called *similarity* [?], which also allows relaxed serializability. A real-time database reflects the status of an external system that changes continuously and the value of a data object cannot be updated continuously to perfectly track the status of the real-world entity. However, data values of a data object that are slightly different in age or in precision are oftenacceptable as read data for transactions. This observation underlies the concept of similarity among data values.

Another paper that allows weaker forms of consistency in a way similar to Read-Only transactions [6] is a recent algorithm by Bober and Carey [2]. However, their queries still see a transaction-consistent view of the database, possibly with some older version that does not contain all the updates.

In addition to the demarcation protocol [1] mentioned in Section 3.3, there are other papers concerning different approaches to dynamically distributing resources in a distributed system [?, ?, ?]. With appropriate modifications, these general resource allocation approaches can also be used to redistribute the ϵ-spec among the sub-ETs in the design of distributed divergence control algorithms.

Although the Superdatabase paper [15] described an optimistic validation algorithm, there are other alternative schedulers for heterogeneous concurrency control that enforce global serializability more conservatively. For example, one could distribute an ordering at transaction creation time [4]. This solution depends on the ability of element databases to enforce the ordering, which is not the case for 2PL-based element databases, to give one example. Another possibility is the use of forced local conflicts to obtain a rigorous schedule [7]. Whether these other algorithms can be easily modified to support distributed divergence control is a subject of research.

8 Summary

In this paper, we demonstrated the feasibility of epsilon serializability in both homogeneous and heterogeneous distributed databases by showing various concrete and representative distributed divergence control algorithms for ESR. Specifically, the strict 2-phase locking distributed divergence control (S2PLDDC), the optimistic distributed divergence control using weak locks (ODDC), and the Superdatabase distributed divergence control algorithms were presented. For both S2PLDDC and ODDC algorithms, we assumed that the local orderings of sub-ETs are the same among all the sites, and as a result, the total fuzziness of a distributed ET is simply the sum of the local fuzziness of all its sub-ETs. However, for the Superdatabase distributed divergence control algorithm, no assumption regarding the local orderings of sub-ETs was made, and the total fuzziness of a distributed ET may be greater than the sum of the local fuzziness of all its sub-ETs. Therefore, in addition to executing a local divergence control algorithm in each site, a global mechanism was described to take into account the additional fuzziness. We also discussed some advanced issues in distributed divergence control design, including how to calculate the output inconsistency of an ET and alternative policies to dynamically change the epsilon specification of a sub-ET.

References

[1] D. Barbara and H. Garcia-Molina. The demarcation protocol: A technique for maintaining linear arithmetic constraints in distributed database systems. In *Proceedings of the International Conference in Extending Database Technology*, Vienna, March 1991.

[2] P.M. Bober and M.J. Carey. Multiversion query locking. In *Proceedings of the Twenty-First International Conference on Very Large Data Bases*, Vancouver, August 1992.

[3] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, SE-17(9):954–960, September 1991.

[4] A. Elmagarmid and W. Du. A paradigm for concurrency control in heterogeneous distributed database systems. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 37–46, Los Angeles, February 1990.

[5] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of ACM*, 19(11):624–633, November 1976.

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

[7] D. Georgakopoulos and M. Rusinkiewicz. On serializability of multidatabase transactions through forced local conflitcs. In *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan, April 1991.

[8] P.M. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[9] Gail E. Kaiser and Brent Hailpern. An object model for shared data. In *International Conference on Computer Languages*, pages 135–144, New Orleans LA, March 1990.

[10] Iyengar Krishnan and Wolfgang Zimmer, editors. *IFIP TC6/WG6.6 2nd International Symposium on Integrated Network Management*, Washington DC, April 1991. North-Holland.

[11] B. Lindsay. A retrospective of R*: A distributed database management system. *Proceedings of the IEEE*, 75(5):668–673, May 1987.

[12] C. Pu. Generalized transaction processing with epsilon-serializability. In *Proceedings of Fourth International Workshop on High Performance Transaction Systems*, Asilomar, California, September 1991.

[13] 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.

[14] 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.

[15] Calton Pu. Superdatabases for composition of heterogeneous databases. In Amar Gupta, editor, *Integration of Information Systems: Bridging Heterogeneous Databases*, pages 150–157. IEEE Press, 1989. Also in IEEE Computer Society Tutorial, *Multidatabase Systems: An Advanced Solution for Global Information Sharing*. The paper originally appeared in the Proceedings of Fourth International Conference on Data Engineering, 1988, Los Angeles.

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

[17] Y. Raz. The principle of commitment ordering. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 292–312, Auguest 1992.

[18] *1st International Conference on Artificial Intelligence Applications on Wall Street*, New York NY, October 1991. IEEE Computer Society Press.

[19] 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.