Efficient Numerical Error Bounding for Replicated Network Services

Haifeng Yu and Amin Vahdat
Computer Science Department
Duke University, Box 90129
Durham, NC 27708
{yhf, vahdat}@cs.duke.edu

Abstract

The goal of this work is to use database techniques to support replicated network services that accept updates to numerical records from multiple locations. Given the high overhead of maintaining strong consistency, many replicated services can tolerate divergence of their shared data, as long as the numerical error is bounded. Target distributed services include replicated stock quotes services, online auctions, distributed sensor systems, wide-area resource accounting and load balancing for replicated servers. While these target systems are broader than typical database applications, this work demonstrates how variants of existing database techniques can support these applications.

We present two algorithms to efficiently bound absolute error using only local information. Split-Weight AE *separately bounds negative and positive weights, while* Compound-Weight AE *bounds them together. The two algorithms can be combined to provide good performance and low space overhead. Our* Inductive RE *bounds relative error also based on local information, taking advantage of the fact that the divergence was properly bounded prior to each invocation of the algorithm to reduce required wide-area communication. We discuss two optimizations that enable the algorithms to scale to thousands of data items and hundreds of replicas.*

1 Introduction

In our efforts to build high performance replicated network services, we repeatedly encountered scenarios where we needed to bound the relative error of numerical values stored and updated at multiple sites. Consider the following replicated services:

Stock Quotes Services Users retrieve stock quotes from replicated stock quotes servers. Each server may also accept updates to the current quotes. Users are concerned with the amount of "error" in the quotes they observe. For example, a user may prefer to see quotes within only ± 1 cent/share (absolute error) or $\pm 1\%$ (relative error) of the accurate quotes.

Online Auctions Each online auction server maintains the highest current bid for a number of items. A user accessing a replica desires guarantees regarding the maximum difference between the highest bid stored locally and the largest global bid.

Distributed Sensor Systems The sensor system takes the average temperature (pollution level, etc.) of an area. Each sensor periodically takes a sample at a fixed point in the area, and updates the average value according to the new sample value taken. User may retrieve the average value at any sensor location. It is desirable that the error in the average value is bounded.

Wide-area Resource Accounting This is an open problem in current operating system and network research[7, 21, 22]. As we move toward global distributed computing, one goal is to account for aggregate consumed resources across multiple providers on a per-user basis. Given the scale of this problem, maintaining accurate resource usage information will incur prohibitive overhead. Allowing bounded error in usage information is a promising approach to solving this problem.

Load Balancing for Replicated Servers For many replicated services, client programs do not directly choose which replica to contact. Instead, they contact a nearby font end and the front end then forwards the request to the server judged to deliver the highest quality of service for that request. A front end uses the forwarding history to estimate the load of each server. When there are multiple front ends[16, 21], each front end sees a subset of the request stream, and uses that information to update the estimated load information. Once again, the load information is updated from multiple locations, and it is beneficial to bound the maximum error on load information observed by each front end.

One approach for guaranteeing accurate numerical information is to utilize standard techniques for maintaining strong consistency across wide-area networks. However, the communication costs and latency associated with such techniques often have prohibitively high overhead. We observe that many replicated services, including the ones described above, can tolerate some level of inconsistency in exchange for improved performance, as long as they are provided guarantees regarding the maximum allowable error. In this context, the goal of this work is to build on existing database techniques to efficiently bound numerical inaccuracy by reducing the amount of required wide-area communication.

Despite the importance of bounding numerical error for replicated network services, this topic has not been well studied in the literature. In the context of data caching, [2] proposes the concept of bounded numerical error. However, the authors do not generalize the concept to replicated databases. Much work[13, 14, 15, 18, 19, 20] that exploits weak consistency concentrates on aspects other than numerical error. Integrity constrain management algorithms[3, 4, 5, 6, 10, 11] for distributed databases are related to error bounding but most of them cannot be efficiently applied to the special case of bounding numerical error. The demarcation protocol[3] allows easy maintenance of linear inequalities for distributed databases.

Error bounding is closely related to enforcing linear inequalities but has three important properties not present in general linear inequalities: i) The copies of a data item are inter-related, ii) servers have approximate information about what writes other servers have seen, iii) during write propagation, writes on all data items are propagated. Not being able to exploit these properties makes the demarcation protocol infeasible for bounding numerical error because of prohibitive communication and space overhead. See Section 7 for detailed discussion.

In this paper, we present algorithms to efficiently bound numerical error for replicated network services. They are developed in the TACT[23] project, which is a toolkit for building replicated Internet services. Two algorithms *Split-Weight AE* and *Compound-Weight AE* are proposed to bound absolute error. They all bound error by limiting the "total weighted writes" (see Section 2) accepted by one server but not seen by another. All decisions are based on local information. Split-Weight AE makes conservative decisions but easy to optimize space usage, while Compound-Weight AE makes optimal decisions but difficult to reduce space overhead. We combine the two algorithms to achieve good performance and small space overhead. Our *Inductive RE* bounds relative error by transforming the problem into absolute error and then using Split-Weight AE or Compound-Weight AE to bound the absolute error. Inductive RE takes advantage of the fact that the divergence was properly bounded prior to each invocation of the algorithm. We also study the performance of our algorithms through both analysis and simulation.

This paper makes the following contributions:

- We describe the importance of bounding numerical error to support replicated network services.
- We propose practical algorithms to bound absolute error and relative error using only local information.
- Two optimizations enable the algorithms to scale to thousands of data items and hundreds of replicas.

The next section describes our replicated database model. We present our error bounding algorithms in Sections 3 and 4. Section 5 discusses two important optimizations for our algorithms. In Section 6, we study the performance of the algorithms. Related work is described in Section 7. In Section 8, we draw the conclusions.

2 System Model

The database we consider is replicated on n *server*s, $server_1$, $server_2$, ..., $server_n$, maintaining the data shared by the network servers. The replicated database is composed of multiple *data item*s. Each data item has a numerical value for which the service desires to bound *error*. In the stock quotes example, a data item is the quote of a stock. The allowed error for a data

item is independent of other data items. We first focus on the case of a single data item and then discuss scalability issues for multiple data items in later sections.

Every server can accept reads (inquires) and writes (updates) from users. Reads return the current value of the data item on the server. A write W increases or decreases the value of a data item by some amount, which is called the *weight* ($W.weight$) of the write. $W.weight$ is positive for increases and negative for decreases. While beyond the scope of this paper, our algorithms can be extended to transactions that consist of multiple primary operations in a straightforward manner.

The server that accepts a write W from a client is the *originating server* of the write, and is denoted by $W.server$. Upon accepting a write, a server does not have to update other servers immediately and divergence among replicas is allowed. However, we still assume that *eventual consistency*[8, 15, 17] is preserved in the system. With eventual consistency, all database replicas will converge to the same "final image" within finite amount of time, if no new writes are introduced into the system. A server updates other servers by propagating writes. The database image itself is never communicated to other servers. Writes with the same originating server are always propagated according to the order they are accepted by that server. *Write propagation* can be done in the form of gossip messages[15], anti-entropy sessions[8, 17], broadcast or even unicast. To reduce communication overhead, some write propagation methods allow multiple writes to be merged into one write during propagation. Our algorithms are orthogonal to the write propagation method used by the database, although the freshness of views (defined later in this section) may be affected.

A server may propagate writes to other servers at any time, and such write propagation is called *voluntary write propagation* or *background write propagation*. The error bounding algorithms may require a server to propagate writes, which is called *compulsory write propagation*. Compulsory write propagation is necessary for the correctness of the algorithms, while voluntary write propagation only affects performance.

Each server maintains a write log, which is an ordered list of writes the server accepts from clients or sees from other servers. Write log recycling can be done using various techniques[8, 15, 17]. We define the functions $twn(i,j)$ and $twp(i,j)$ as:

$$twn(i,j) = \sum\{W.weight \mid W.weight < 0 \text{ and } W.server = server_j \text{ and } W \in \text{write log of } server_i\}$$

$$twp(i,j) = \sum\{W.weight \mid W.weight > 0 \text{ and } W.server = server_j \text{ and } W \in \text{write log of } server_i\}$$

Intuitively, $twn(i,j)$ is the total negative weights of the writes $server_i$ sees originated from $server_j$, while $twp(i,j)$ is the total positive weights. Distinguishing negative weights and positive weights is necessary because we allow weights on different data items to be added up in our optimization, which means negative weights on one data item should not offset the positive weights on another data item (see Sections 5.1 and 5.2).

We use V_i to denote the value of the data item on $server_i$, and V_{init} to denote its initial (consistent) value. We use V_{final} to denote the eventual consistent value. The following

equalities hold for V_i, V_{init} and V_{final}:

$$\begin{aligned} V_i &= V_{init} + \sum_{k=1}^{n} (twn(i,k) + twp(i,k)) \\ V_{final} &= V_{init} + \sum_{k=1}^{n} (twn(k,k) + twp(k,k)) \end{aligned}$$

For $server_i$, a data item's *absolute error*(AE) is bounded within $[\alpha_i, \beta_i]$ ($\alpha_i \leq 0 \, and \, \beta_i \geq 0$) if and only if at all times, the following inequality holds:

$$\alpha_i \leq V_{final} - V_i \leq \beta_i \tag{1}$$

For instance, in the stock quotes service, if we want to bound the error observed by users on $server_1$ within ± 1 cent/share from the accurate quote, we can set $\alpha_1 = -1$, $\beta_1 = 1$. Similarly, we say the *relative error*(RE) is bounded within $[\gamma_i, \delta_i]$ ($\gamma_i \leq 0 \, and \, 0 \leq \delta_i \leq 1$) if an only if:

$$\gamma_i \leq 1 - \frac{V_i}{V_{final}} \leq \delta_i \tag{2}$$

In the stock quotes example, we can bound the stock quotes error on $server_1$ within $\pm 1\%$ by setting $\gamma_1 = -0.01$ and $\delta_1 = 0.01$. For relative error, we assume $V_i > 0, \ 1 \leq i \leq n$.

Each server in the system has approximate knowledge of what writes other servers have seen. We say that each server has its *view* of $twn(i,j)$ and $twp(i,j)$, for $1 \leq i \leq n, 1 \leq j \leq n$. The views are updated during write propagation. The actual update fashion and view freshness depend on the write propagation method. For example, if we use unicast, then during each write propagation, the two parties can inform each other of the writes they see. For anti-entropy sessions, more efficient view update mechanism can be used and details can be found in [8]. The correctness of our algorithms does not depend on the freshness of the views, but performance is affected.

We denote $server_k$'s view of $twn(i,j)$ and $twp(i,j)$ as $twn_k(i,j)$ and $twp_k(i,j)$. Intuitively, $twn_k(i,j)$ is the total negative weights of the writes that $server_k$ believes that $server_i$ sees from $server_j$. During a *view advance*, $server_k$ updates $twn_k(i,j)$ and $twp_k(i,j)$. Views are *conservative* in that $server_k$ will never assume that $server_i$ sees a write that $server_i$ actually does not see. So we have the following properties for $1 \leq i, j, k \leq n$:

$$twn(j,j) \leq twn(i,j) \leq twn_k(i,j) \leq 0 \tag{3}$$

$$twp(j,j) \geq twp(i,j) \geq twp_k(i,j) \geq 0 \tag{4}$$

3 Bounding Absolute Error Using Local Information

This section describes two different algorithms for bounding AE. The idea is to bound the total weights of writes accepted by one server but not seen by other servers. The first algorithm,

Split-Weight AE, bounds positive weights and negative weights separately. The second algorithm, *Compound-Weight AE*, keeps track of the possible range of values on other servers and allows negative weights and positive weights to offset. Both algorithms require cooperation of all servers in the system, that is, each server must help every other server to enforce its bounds. For each algorithm, we discuss how $server_j$ acts to bound the error for a single data item on $server_i$.

3.1 Split-Weight AE

In this algorithm, each $server_j$ maintains two local variables x and y for $server_i, i \neq j$. They are used to record the total negative and positive weights of the writes accepted by $server_j$ but not seen by $server_i$. $server_j$ uses its view to compute x and y. However, since the view is conservative, x and y are also conservative.

Both variables x and y are originally zero and are updated in the following fashion:

1. When $server_j$ accepts a new write W, if $W.weight < 0$, $x = x + W.weight$, else $y = y + W.weight$.

2. When $server_j$ advances its view, if $twn_j(i,j)$ and $twp_j(i,j)$ are updated to $twn'_j(i,j)$ and $twp'_j(i,j)$ respectively, then $x = x - (twn'_j(i,j) - twn_j(i,j))$ and $y = y - (twp'_j(i,j) - twp_j(i,j))$. Intuitively, this subtracts weights of the newly propagated writes from x and y.

When $server_j$ receives a write W from a client, it checks the conditions:

$$x + W.weight \geq \alpha_i/(n-1), \ if \ W.weight < 0 \tag{5}$$

$$y + W.weight \leq \beta_i/(n-1), \ if \ W.weight > 0 \tag{6}$$

If the conditions do not hold, $server_j$ must advance its view for $server_i$ (potentially propagating writes to $server_i$) before it can accept this write. At the extreme, if the conditions still do not hold after propagating all writes to $server_i$, $server_j$ must perform two-phase commit with $server_i$ for this new write.

Theorem 1 *Split-Weight AE, if carried out by all servers in the system, bounds the absolute error on each server.*

Proof: From the way we update x, we know at any time on $server_j, 1 \leq j \leq n$, $x = twn(j,j) - twn_j(i,j)$. Checking condition (5) ensures that $x \geq \alpha_i/(n-1)$. Taking inequality (3) into account, the error for $server_i$ is:

$$\begin{aligned} V_{final} - V_i &= (V_{init} + \sum_{j=1}^{n}(twn(j,j) + twp(j,j))) - (V_{init} + \sum_{j=1}^{n}(twn(i,j) + twp(i,j))) \\ &= \sum_{j=1}^{n}(twn(j,j) - twn(i,j)) + \sum_{j=1}^{n}(twp(j,j) - twp(i,j)) \end{aligned}$$

$$\geq \sum_{j=1}^{n}(twn(j,j) - twn(i,j)) \geq \sum_{j=1}^{n}(twn(j,j) - twn_j(i,j))$$
$$\geq (n-1) \times \alpha_i/(n-1) = \alpha_i$$

In a similar fashion, it can be shown that $V_{final} - V_i \leq \beta_i$. □

Split-Weight AE is pessimistic, in the sense that $server_j$ may propagate writes to $server_i$ when it is actually not necessary to bound the error. For example, the algorithm does not consider the case where negative weights and positive weights may offset each other. In our simulation study (see section 6.2), we will quantify how pessimistic Split-Weight AE is under different workloads. However, this simple design enables several optimizations not applicable to Compound-Weight AE (see Section 3.2). For example, in order to optimize the space overhead, several data items may need to share the same x and y variables (see section 5.1).

3.2 Compound-Weight AE

In Compound-Weight AE, each $server_j$ maintains three local variables z, min and max for $server_i$, $i \neq j$. Intuitively, z is the total weights of the writes accepted by $server_j$ but not seen by $server_i$, in $server_j$'s view. However, since a view can be stale, $server_i$ may actually see more writes than $server_j$ believes. So we use min/max to record the minimum/maximum possible total weights of those writes that $server_i$ can see but is not in $server_j$'s view. Both values are conservative, however, $server_j$ can only use these conservative values without having global knowledge. We will show later that this algorithm makes optimal decisions, given only local information.

All variables z, min and max are originally zero and are updated in the following fashion:

1. When $server_j$ accepts a new write W, $z = z + W.weight$. If $z < min$, then $min = z$. If $z > max$, then $max = z$.

2. When $server_j$ advances its view for $server_i$, $server_j$ first sets all three variables to zero and then re-scans the write log corresponding to the unseen part of the write log as if the writes were newly submitted from clients. In this way, z, min and max are re-established for this new view. Rescanning the write log whenever $server_j$ advances its view appears redundant, but is actually necessary for correctness.

When $server_j$ receives a write W from a client, it checks the following conditions:

$$z + W.weight - max \geq \alpha_i/(n-1) \quad (7)$$
$$z + W.weight - min \leq \beta_i/(n-1) \quad (8)$$

If the condition does not hold, $server_j$ must advance its view for $server_i$ (potentially propagating writes to $server_i$) before it can accept this write.

Theorem 2 *Compound-Weight AE, if carried out by all servers in the system, bounds the absolute error on each server.*

Proof: At any time, $z = (twn(j,j) + twp(j,j)) - (twn_j(i,j) + twp_j(i,j))$ and $z - max \geq \alpha_i/(n-1)$, $z - min \leq \beta_i/(n-1)$. The algorithm ensures that max is the largest z $server_j$ has observed since last view advance. This is also the largest possible total weighted writes $server_i$ can see from $server_j$ after last view advance. So we have:

$$max \geq (twn(i,j) + twp(i,j)) - (twn_j(i,j) + twp_i(i,j))$$

$$\begin{aligned}
&(twn(j,j) + twp(j,j)) - (twn(i,j) + twp(i,j)) \\
&= (twn(j,j) + twp(j,j) - (twn_j(i,j) + twp_j(i.j)) - \\
&\quad (twn(i,j) + twp(i,j) - (twn_j(i,j) + twp_i(i,j)) \\
&\geq z - max \geq \alpha_i/(n-1)
\end{aligned}$$

Similarly, we can prove $(twn(j,j)+twp(j,j))-(twn(i,j)+twp(i,j)) \leq \beta_i/(n-1)$. From these two inequalities, as in the proof for Theorem 1, it can be shown that $\alpha_i \leq V_{final} - V_i \leq \beta_i$. □

As opposed to Split-Weight AE, Compound-Weight AE makes the optimal decision that can be made without global knowledge on whether to propagate writes. In other words,

$$\alpha_i/(n-1) \leq (twn(j,j) + twp(j,j)) - (twn(i,j) + twp(i,j)) \leq \beta_i/(n-1) \tag{9}$$

holds if and only if conditions (7) and (8) hold. Theorem 2 states the "if" part and we now informally prove the "only if" part by contradiction. If condition (7) or (8) does not hold, without loss of generality, suppose $z-max \leq \alpha_i/(n-1)$. It is possible that $server_i$ sees a prefix of $sever_j$'s write log that corresponds to max. In that case, we have $(twn(i,j) + twp(i,j)) - (twn_j(i,j) + twp_j(i,j)) = max$. Using the same approach in the proof of Theorem 2, we can show $(twn(j,j) + twp(j,j)) - (twn(i,j) + twp(i,j)) = z - max < \alpha_i/(n-1)$, which violates inequality (9). Thus conditions (7) and (8) are both sufficient and necessary conditions for (9).

4 Bounding Relative Error Using Local Information

The basic idea of our relative error bounding algorithm, *Inductive RE*, is to transform the relative error into absolute error. Here we should emphasize that V_{final} may not be known by any server. So one naive way is to transform definition (2) to $\gamma_i/(1-\gamma_i) \times V_i \leq V_{final} - V_i \leq \delta_i/(1-\delta_i) \times V_i$. By setting $\alpha_i = \gamma_i/(1-\gamma_i) \times V_i$ and $\beta_i = \delta_i/(1-\delta_i) \times V_i$, we can apply either of the previous algorithms to enforce the inequality. However, since V_i changes over time, $server_i$ must constantly update α_i and β_i and inform other servers in the system. This requires that a consensus algorithm be run among all servers whenever V_i decreases. As a result, the performance could degrade significantly.

Inductive RE is based on the observation that for any j, V_j was properly bounded before the invocation of the algorithm and is an approximation of V_{final}. So $server_j$ may use V_j as an approximate norm to bound γ_i and δ_i. Transforming the definition of RE, we have:

$$\gamma_i \leq 1 - V_i/V_{final} \leq \delta_i \iff \left\{ \begin{array}{lcl} V_{final} - V_i & \geq & \gamma_i \times V_{final} \\ V_{final} - V_i & \leq & \delta_i \times V_{final} \end{array} \right.$$

We know that $\gamma_j \leq 1 - V_j/V_{final}$, so $V_{final} \geq V_j/(1-\gamma_j)$. The following two inequalities are sufficient conditions for the above two inequalities:

$$\begin{aligned} V_{final} - V_i &\geq \frac{\gamma_i}{1-\gamma_j} \times V_j \\ V_{final} - V_i &\leq \frac{\delta_i}{1-\gamma_j} \times V_j \end{aligned}$$

The left-hand side expressions can be evaluated using only local information. So in order to bound relative error for $server_i$, $server_j$ only needs to apply Split-Weight AE or Compound-Weight AE and use $\gamma_i/(1-\gamma_j) \times V_j$ as α_i and $\delta_i/(1-\gamma_j) \times V_j$ as β_i. Note that since α_i and β_i change with V_j, whenever V_j changes, the limits should be recomputed and re-checked. However, no consensus algorithms are necessary because V_j is local.

Theorem 3 *Inductive RE, if carried out by all servers in the system, bounds the relative error on each server.*

Proof: Define a step to be a server accepting a write or propagating writes to another server. Use mathematical induction on the number of steps. First of all, the condition $\gamma_i \leq 1 - V_i/V_{final} \leq \delta_i$ holds for all servers when the system starts. By applying the algorithm for bounding absolute error at step $m+1$, $server_j$ ensures

$$\begin{aligned} twn(j,j) + twp(j,j) - (twn(i,j) + twp(i,j)) &\geq \frac{\gamma_i}{(1-\gamma_j) * (n-1)} \times V_j \\ twn(j,j) + twp(j,j) - (twn(i,j) + twp(i,j)) &\leq \frac{\delta_i}{(1-\gamma_j) * (n-1)} \times V_j \end{aligned}$$

From induction hypothesis, $V_{final} \geq V_j/(1-\gamma_j)$. So

$$\begin{aligned} twn(j,j) + twp(j,j) - (twn(i,j) + twp(i,j)) &\geq \gamma_i/(n-1) \times V_{final} \\ twn(j,j) + twp(j,j) - (twn(i,j) + twp(i,j)) &\leq \delta_i/(n-1) \times V_{final} \end{aligned}$$

Taking all servers into account, $V_{final} - V_i \geq \gamma_i \times V_{final}$ and $V_{final} - V_i \leq \delta_i \times V_{final}$ hold for $server_i$ $1 \leq i \leq n$. It then follows that $\gamma_i \leq 1 - V_i/V_{final} \leq \delta_i$ holds for all servers at step $m+1$. □

5 Optimizing for Scalability

5.1 Reducing Space Overhead

We have discussed how to bound AE and RE for a single data item. The algorithms incur a per data item space overhead of $O(n)$, where n is the number of servers. If we simply use multiple instances of the algorithms, the size of the data structure maintained by the algorithms can be n times the size of the database itself. In the case where a database has tens of thousands of data items, this high space overhead is prohibitive.

To reduce space overhead, we assume that for all data items, $server_i$ has the same α_i and β_i (or γ_i and δ_i), otherwise the space needed simply for storing α_i and β_i will grow linearly

with the number of data items. The application may still use several different α_is(β_is) for different data items by using multiple instances of our algorithm. We reduce space overhead by exploiting the fact that during write propagation, writes to all data items on a server are propagated to another server. So we only need to maintain information for those data items accessed between two write propagations. We also take advantage the possible locality among the writes accepted by a server. For "hot" data items, we maintain accurate information needed by the algorithms. For data items seldom accessed, we allow them to share the same data structure and maintain conservative information.

We use a hashtable to store the variables needed by our algorithms. Each server maintains one hashtable for every other server in the system. The hashtables are used to maintain the information on "hot" data items. Whenever $server_j$ receives a writes on data item D, it uses D as a key to create or update variables in the hashtables. The total space used by the hashtable is bounded. In the case where a hashtable becomes full, we created a shared entry for all other data items without a hashtable entry. On each write propagation, the hashtable and the shared entry corresponding to the receiving server are cleared and the space is freed.

Care must be taken when maintaining the shared entry. For Split-Weight AE, the shared entry simply consists of two variables x and y, which are updated in the same way as normal hashtable entries. For Compound-Weight AE, it is difficult to maintain a shared entry for multiple data items, so we use Split-Weight AE for that shared entry and Compound-Weight AE for hashtable entries. In Inductive RE, the shared entry must also record the smallest V_j of the data items using that entry, so that the computed α_i and β_i values are tight. Using shared entries may result in over-pessimistic behavior, since weights accumulated on multiple items are considered on a single item. However, a server can improve performance at the cost of larger hashtables. Thus, our design allows a server to trade space for performance.

Some simple analysis can provide us with intuition on how much space overhead the optimization can reduce. Suppose the database consists of 100,000 data items and the space needed for our algorithms is $(16 \times n)$bytes per data item, where n is the number of servers. With 100 servers, the space overhead will be around 160MBytes if we simply use 100,000 instances of our algorithms. On the other hand, with our optimization, if on average there are 50 data items accessed between two write propagations and we do not bound the size of hashtables, the overhead will be cut down to about 80KBytes without performance being affected.

5.2 Reducing Computational Overhead

In our algorithms, a server needs to update one hashtable for every other server in the system when accepting a write. The updating operations are on the critical path and if there are a large number of servers, the overhead of updating n hashtables on each write can be high. In this section, we discuss how to reduce this computational overhead.

The first possible optimization is to collapse the hashtables for multiple servers. We can

group together servers with similar bounds, and enforce the tightest bounds for a group of servers. The servers in the group can then share a single hashtable. Once again, a server can trade space for performance by using smaller groups. Note that this optimization also reduces the space overhead.

Another optimization is to use a cache, so that in most cases, we only need to update the cache rather than n hashtables. We only discuss how to use a cache for bounding error with Split-Weight AE. The data structures in Compound-Weight AE make it difficult to utilize a cache.

For bounding absolute error, each cache entry maintains the following information:

item: database item
x: total negative weights of newly accepted writes since entry creation
y: total positive weights of newly accepted writes since entry creation
limitx: the limit for x
limity: the limit for y
serverx: the server whose limit we use for this entry's *limitx*
servery: the server whose limit we use for this entry's *limity*

To create a cache entry for data item D, we scan all hashtables, and set

$$\begin{aligned} limitx &= max\{\alpha_i/(n-1) - x \mid x \; in \; the \; hashtable \; entry \; for \; D\} \\ limity &= min\{\beta_i/(n-1) - y \mid y \; in \; the \; hashtable \; entry \; for \; D\} \end{aligned}$$

The variables x and y are set to zero. On each cache hit, we check $x + W.weight \geq limitx$ (if $W.weight < 0$) or $y + W.weight \leq limity$ (if $W.weight > 0$). As long as the condition holds, we only need to update x or y in the cache entry, rather than updating all hashtables. If the condition does not hold, we writeback the cache entry to the hashtables, and potentially perform compulsory write propagation. After that, we can establish a new cache entry for the data item with new *limitx* and *limity* values. The cache must be flushed whenever $server_i$, $1 \leq i \leq n$ changes α_i or β_i. We consider this an infrequent operation, so the performance penalty will not be excessive.

A further optimization is to use a linked list for each cache entry. The first node in the list has the tightest *limitx* and *limity*, the second node has the second tightest values and so on. When x or y reaches *limitx* or *limity*, we remove the first node and update the hashtable corresponding to *serverx* and *servery*. In this way, we can avoid scanning all hashtables to find the next tightest limits. However, after updating the hashtables for *serverx* and *servery*, we still need to go through the linked list to see whether *serverx* or *servery* now has tighter limits than nodes in the list.

A cache "snapshot" must be made on write propagation. This snapshot is used in the future to create a "diff" when we write back a cache entry. The x and y in the snapshot are subtracted from the cache entry being written back, before the cache entry is added up to the hashtable entry.

Applying the cache idea to bounding relative error is subtle. Since the computed α_i and β_i changes with V_j, in order to choose safe $limitx$ and $limity$, we must decouple the limits from V_j. Recall the conditions we want to enforce are:

$$x \geq \gamma_i/(1-\gamma_j) \times V_j = s_i \times V_j \quad (10)$$
$$y \leq \delta_i/(1-\gamma_j) \times V_j = t_i \times V_j \quad (11)$$

Let the current value of x, y and V_j be x_0, y_0 and V_{j0}, respectively. We have $V_j = (V_{j0} - x_0 - y_0) + x + y$. Use this equation to substitute V_j in (10) and (11), we have:

$$x \geq s_i \times ((V_{j0} - x_0 - y_0) + x + y)$$
$$y \leq t_i \times ((V_{j0} - x_0 - y_0) + x + y)$$

Solve these two inequalities and choose a rectangular solution area, we have sufficient conditions for (10) and (11):

$$x \geq \frac{s_i}{1-s_i} \times (V_{j0} - x_0 - y_0)$$
$$y \leq \frac{t_i}{(1-t_i)(1-s_i)} \times (V_{j0} - x_0 - y_0)$$

Using these two conditions, we can now set:

$$limitx = max\{s_i/(1-s_i) \times (V_{j0} - x_0 - y_0) - x_0 \mid x_0 \textit{ in the hashtable entry for } D\}$$
$$limity = min\{t_i/((1-t_i)(1-s_i)) \times (V_{j0} - x_0 - y_0) - y_0 \mid y_0 \textit{ in the hashtable entry for } D\}$$

V_{j0} only changes when $server_j$ accepts writes from other servers. In that case, the cache should be flushed.

Once again, we use some simple analysis to intuitively understand the benefits of using a cache. Suppose there are 100 servers and it takes 0.1ms to update a hashtable or cache entry. Without optimization, the updating operation will add around 10ms to the latency of each write. If we use a cache and if on average we can use cache for 90% of the writes, the overhead is reduced to about 1.09ms per write.

6 Performance Study

6.1 Performance Analysis

In this section, we compare our approach to a standard two-phase commit protocol in terms of throughput and latency. Our algorithms and two-phase commit protocol treat reads in the same way, so we are mainly interested in the performance for writes and we only consider write workloads. We assume the database consists of a single data item. To simplify discussion, all servers are assumed to have the same bounds, i.e. α_i and β_i. We also assume that the workload is evenly distributed among the n servers. We do not consider background write propagation or indirect view advance, both of which will improve the performance of our algorithms.

notation	meaning	Case 1	Case 2
n	number of servers	10	20
t_{apply}	CPU time to apply a write to database	3ms	3ms
t_{check}	time to check limits and update n hashtables in error bounding algorithms	2ms	4ms
t_{delay}	round-trip message delay in write propagation	200ms	500ms
t_{setup}	CPU time on one replica for TCP connection setup	10ms	10ms
t_{send}	CPU time to send *one* write	1ms	1ms
t_{recv}	CPU time to receive *one* write	1ms	1ms
t_{lock}	average CPU time to acquire (potentially remote) lock in two-phase commit protocol	2ms	2ms
L_i	random variable, the length of an epoch on $server_i$ error bounding algorithms	N/A	N/A
Q_i	random variable, queuing delay on $server_i$ in error bounding algorithms and two-phase commit	100ms	100ms

Table 1: Symbols and Default Values used in the Analysis

We first describe the terms and notations used in our analysis, as summarized in Table 1. We consider two sets of parameters. Case 1 corresponds to a replicated network service distributed across the country, while Case 2 models an international replicated network service. An *epoch on* $server_i$ *for* $server_j$ is the period on $server_i$ between two write propagations to $server_j$. Note that according to this definition, on $server_i$, an epoch for $server_j$ may overlap with an epoch for $server_k$. We define the *length* of an epoch as the number of writes accepted by a server directly from clients during that epoch.

The performance of the algorithms is dependent on the characteristics of the workload, such as the weight of each write and the inter-arrival time between writes. To make our analysis generally applicable, we abstract the workload characteristics with two high-level random variables L_i and Q_i (see Table 1). Our goal is to cover the workload spectrum by choosing different distributions for L_i and Q_i. To gain understanding of what distributions we can expect in real world cases, we perform simulations for several workloads (see Section 6.2).

We now analyze the throughput of our algorithms. The analysis is applicable to all three algorithms, since the difference among them is captured in L_i and Q_i. A write that does not incur write propagation is processed locally, consuming local CPU time $t_{check} + t_{apply}$. Since we assume all servers have the same limit and no background write propagation, if a write on $server_i$ exceeds the limit for $server_j$, it must exceed the limits for all other servers. In that case, $server_i$ must perform compulsory write propagations to $(n-1)$ other servers in the system. Our assumption also means that the epochs on $server_i$ for different servers match and have the same length. So for a write that triggers write propagation, the local CPU time

consumed is:

$$t_{check} + (n-1)t_{setup} + (n-1)L_i \times t_{send} + t_{apply}$$

This write also consumes $t_{setup} + L_i \times t_{recv} + L_i \times t_{apply}$ CPU time on every other server in the system. Thus, the aggregate system CPU time consumed by this write is:

$$\begin{aligned} &t_{check} + (n-1)t_{setup} + (n-1)L_i \times t_{send} + t_{apply} + (n-1)(t_{setup} + L_i \times t_{recv} + L_i \times t_{apply}) = \\ &\quad t_{check} + t_{apply} + 2(n-1)t_{setup} + (n-1)L_i \times (t_{send} + t_{recv} + t_{apply}) \end{aligned}$$

On average, for every $E(L_i)$ writes accepted by $server_i$, where $E(L_i)$ is the expectation of L_i, $(E(L_i) - 1)$ writes can be processed locally and the last write triggers write propagation. So the average system CPU time consumed per epoch is:

$$(E(L_i)-1)(t_{check} + t_{apply}) + t_{check} + t_{apply} + 2(n-1)t_{setup} + (n-1)E(L_i)(t_{send} + t_{recv} + t_{apply})$$

Since we assume all servers have the same limit, L_i and L_j have the same distribution for all i and j. Also note that there are n servers in the system, we have the system throughput:

$$\begin{aligned} Throughput &= \frac{n \times E(L_i)}{E(L_i)(t_{check} + t_{apply}) + 2(n-1)t_{setup} + (n-1)E(L_i)(t_{send} + t_{recv} + t_{apply})} \\ &= \frac{n}{t_{check} + nt_{apply} + 2(n-1)t_{setup}/E(L_i) + (n-1)t_{send} + (n-1)t_{recv}} \end{aligned} \quad (12)$$

The above analysis does not consider the possibility of merging writes during propagation. As we mentioned in Section 2, in many cases, it is possible to merge multiple writes into one write in order to save communication overhead. In the extreme, all writes accepted by a server during an epoch can be merged together during write propagation. We present the throughput of our algorithms for this particular case:

$$Throughput = \frac{n \times E(L_i)}{E(L_i)(t_{check} + t_{apply}) + 2(n-1)t_{setup} + (n-1)(t_{send} + t_{recv} + t_{apply})} \quad (13)$$

In standard two-phase commit protocol, each write must be propagated to all other servers before it can commit, so the throughput for two-phase commit is:

$$Throughput = \frac{n}{t_{lock} + nt_{apply} + 2(n-1)t_{setup} + (n-1)t_{send} + (n-1)t_{recv}} \quad (14)$$

Figure 1 shows the throughput of our algorithms versus two phase commit as a function of $E(L_i)$. The key "No WM" stands for error bounding algorithms without write merging, "WM" stands for error bounding algorithms with write merging, and "TC" stands for two-phase commit protocol. We plot the graphs for $E(L_i) \in [1, 5]$. By definition, $E(L_i)$ is greater than 1. Since our algorithms perform better as $E(L_i)$ increases, the graphs are conservative by not considering larger $E(L_i)$, which we expect to be common for many applications. As expected, the error bounding algorithms have considerably higher throughput than a two-phase commit protocol by reducing wide-area communication. As $E(L_i)$ increases, the throughput of the error bounding algorithms also increases, and in the write merging case, the increase is almost

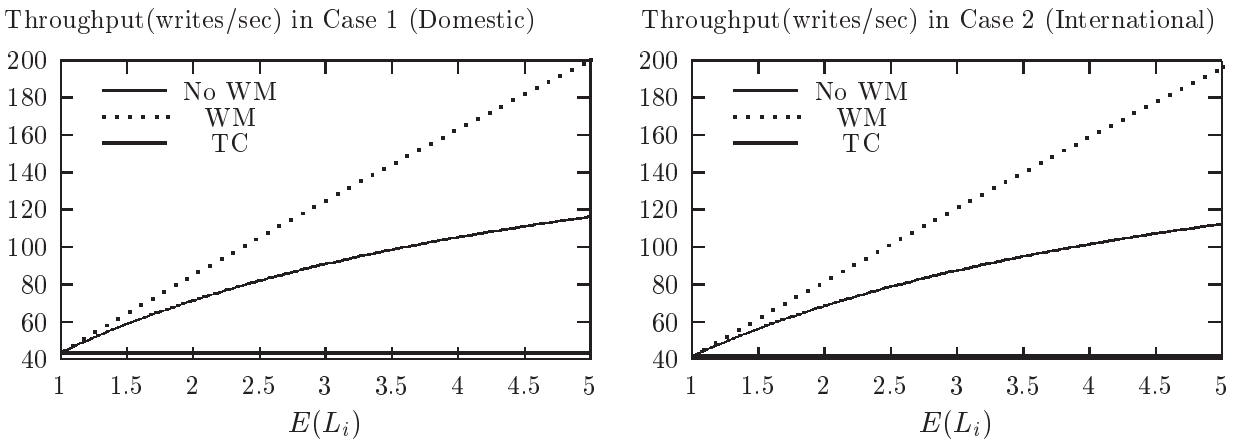

Figure 1: Throughput of Error Bounding Algorithms vs. Two-phase Commit

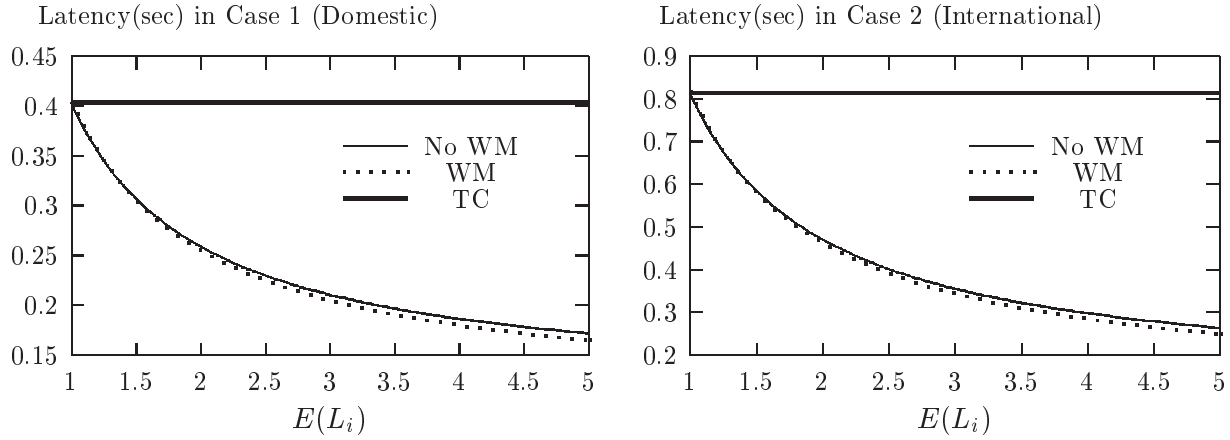

Figure 2: Latency of Error Bounding Algorithms vs. Two-phase Commit

linear. The performance improvement, however, does not come without cost. Larger $E(L_i)$ can sometimes only be gained by tolerating larger numerical error (see Section 6.2). Thus, the gains available to a network service depends upon the magnitude of the numerical error it is willing to tolerate.

Having compared the throughput, we now analyze the latency of the system. In our algorithms, the latency of a write that does not incur write propagation is $Q_i + t_{check} + t_{apply}$. If a write incurs write propagation, the latency (no write merging) is:

$$Q_i + t_{check} + (n-1)t_{setup} + (n-1)L_i \times t_{send} + t_{apply} + t_{delay}$$

Using the same approach in the throughput analysis, we have the latency in the error bounding algorithms as:

$$Latency \quad = \quad Q_i + t_{check} + t_{apply} + (n-1)t_{send} + \frac{(n-1)t_{setup} + t_{delay}}{E(L_i)} \tag{15}$$

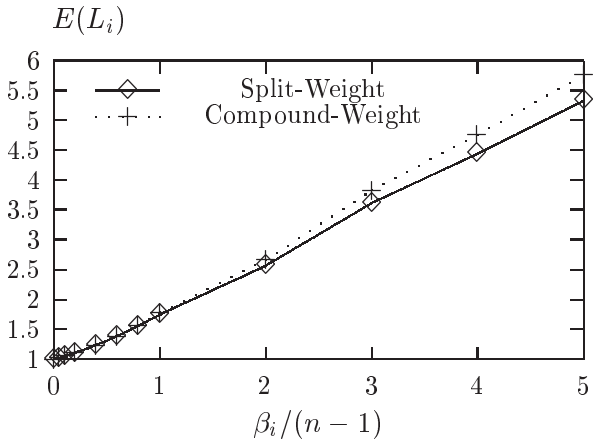

Figure 3: Split-Weight AE vs. Compound-Weight AE (Weight ∼ U(-1, 3))

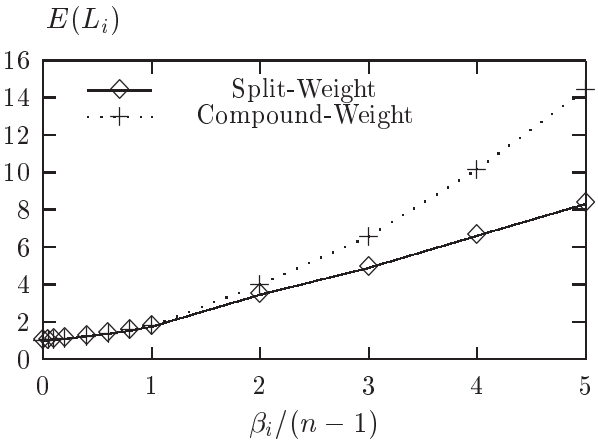

Figure 4: Split-Weight AE vs. Compound-Weight AE (Weight ∼ U(-2, 2))

Again, if we consider the possibility of writes merging and in the extreme case where all writes can be merged, the latency will be:

$$Latency = Q_i + t_{check} + t_{apply} + \frac{(n-1)t_{setup} + (n-1)t_{send} + t_{delay}}{E(L_i)} \quad (16)$$

The average latency in two-phase commit protocol is:

$$Latency = Q_i + t_{lock} + (n-1)t_{setup} + (n-1)t_{send} + t_{delay} + t_{apply} \quad (17)$$

Figure 2 shows the latency of the error bounding algorithms versus two-phase commit as a function of $E(L_i)$. The keys have the same meaning as in Figure 1. We can see that the error bounding algorithms have smaller latency than the two-phase commit protocol. Furthermore, the latency in our algorithms decreases rapidly as $E(L_i)$ increases. However, as with throughput, the performance improvement comes at the cost of data accuracy.

6.2 Simulation Results

In this section, we use simulation to determine a range of typical values for $E(L_i)$ based on the distribution of the weight of individual writes. Although numerous factors affect $E(Q_i)$, it is determined by $E(L_i)$ to a large extent. Thus we believe studying $E(L_i)$ can give us insight into $E(Q_i)$ as well. The simulation results on $E(L_i)$ also quantify the performance difference between Split-Weight AE and Compound-Weight AE. As mentioned earlier, the latter is optimal while the former is better suited for optimizations. The resulting different $E(L_i)$ and $E(Q_i)$ values in the two algorithms directly affect system performance.

$E(L_i)$ is uniquely determined by the distribution of the weights of writes. We consider two different distributions for the weights: uniform distribution and normal distribution. The

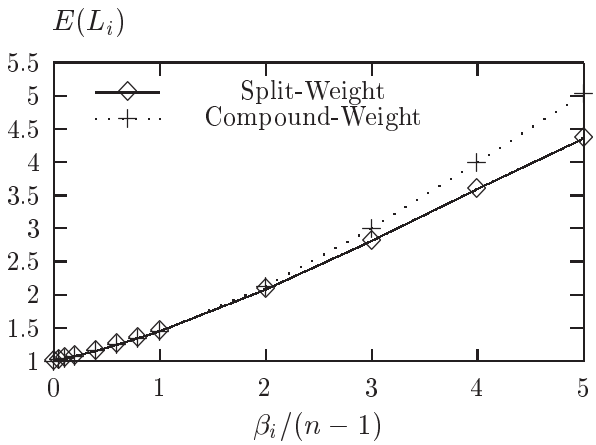

Figure 5: Split-Weight AE vs. Compound-Weight AE (Weight ∼ N(1, 2))

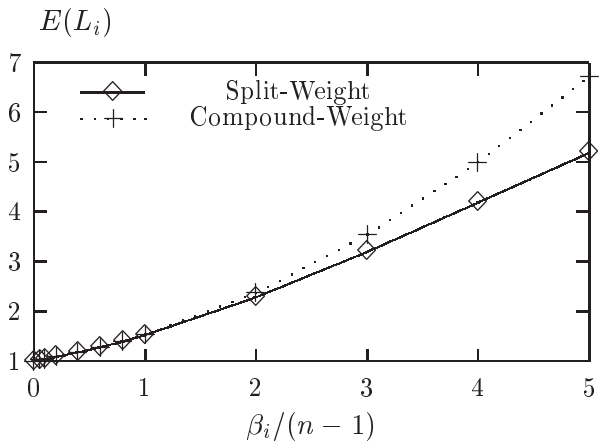

Figure 6: Split-Weight AE vs. Compound-Weight AE (Weight ∼ N(0, 2))

weights in the first workload are uniformly distributed within $[-1, 3]$, while those in the second are uniformly distributed within $[-2, 2]$. We denote the distributions by $U(-1, 3)$ and $U(-2, 2)$, respectively. For normal distribution, we consider weights conforming to $N(1, 2)$ and $N(0, 2)$. Each workload consists of one million writes and we measure the average epoch length as a function of $\beta_i/(n-1)$. The bound α_i is set to $-\beta_i$ in our simulation. Figures 3, 4, 5 and 6 summarize the simulation results.

The figures show that in most cases, $E(L_i)$ increases roughly linearly with $\beta_i/(n-1)$. For Compound-Weight AE in Figures 4 and 6, $E(L_i)$ increases faster than linearly. As we expect, $E(L_i)$ in Compound-Weight AE is always bigger than that in Split-Weight AE. The difference is not so obvious when the distribution is biased toward positive weights and becomes clearer when the distribution is symmetric.

To consider the applicability of these distributions to real-world applications, we consider the load balancing example discussed in Section 1. Suppose there are 5 front ends, updating the load information about several back end servers. Each front end records the rate at which it forwards requests to each server, and uses this information to update the estimated number of requests processed per second by the back end servers. Suppose this request rate fluctuates, but remains roughly stable over the long term. So the mean of the weights of the writes is roughly zero in this case. Suppose the probability of a fluctuation of load by a certain amount conforms to $N(0, 2)$. Then according to Figure 6, if the front ends are willing to tolerate ± 12 requests/sec on the estimated load of the back end servers, $\beta_i/(n-1) = 3$ and $E(L_i)$ is roughly 3.2 in Split-Weight AE or 3.5 in Compound-Weight AE. With more front ends, for instance, 10 front ends, we have to tolerate larger error, i.e. ± 24 requests/sec on the estimated load in order to obtain the same $E(L_i)$. However, this is inherent to the case where updates can be accepted from multiple locations.

7 Related Work

Alonso et. al.[2] propose four coherent conditions in the context of "quasi-copy" caching. One of the four conditions is "arithmetic condition," which specifies the allowed numerical error. Since in quasi-copy caching only the master database may accept updates, maintaining arithmetic condition is a trivial problem. Relative to this effort, we define numerical error for replicated databases and discuss algorithms for bounding the error if updates may be accepted by more than one replicas.

Bounding numerical error in a replicated database is closely related to maintaining integrity constraints in distributed databases. In [6], strong theoretical conclusions are made on how to decompose an arbitrary global constraint into a number of local constraints and communication constraints. The conclusions form the basis of the demarcation protocol[3], which applies a number of optimizations to the special case of linear arithmetic inequalities. Bounding numerical error is intrinsically enforcing an inequality. However, we exploit three special properties in this problem, which makes our algorithms practical and efficient for numerical error bounding. First of all, in the error bounding problem, the copies of a data item are inter-related. For example, if we want to bound the AE on $server_1$, it is not necessary to limit V_1 and our algorithms do not limit it. But the demarcation protocol will have to put a limit on every variable present in the inequality. Secondly, in our algorithms, view advance is automatically incorporated and there is no need to explicitly re-adjust limits for local values. On the other hand, the demarcation protocol does not exploit the fact that servers may have knowledge of what writes other servers have seen. In other word, limits re-adjustments are always done explicitly. Also, because the demarcation protocol cannot exploit the fact that copies are brought to eventual consistency through write propagation, it is difficult to design efficient limit re-adjustment policies for it. The third property we utilize is that during a write propagation, all write are propagated and all limits can be reset. This allows us to optimize the space overhead using hashtables. The demarcation protocol requires space for at least n limits for each data item, which limits its scalability.

Gupta et.al. [10] describe an algorithm to verify a global constraint using only local information. In the case of a tuple insertion, the algorithm uses other "covering tuples" already in the tuple space to prove that the constraint is not affected by the new tuple. The technique cannot be applied to bounding numerical error, since no "covering tuple" can be obtained when users update a numerical data item.

In Section 5.2, we discussed how to efficiently check n conditions given a new write. This is a special case of how to efficiently check local integrity constraints given an update to the database. The general problem has been very well studied[4, 5, 11]. However, most of the study[5, 11] concentrates on how to filter those local constraints that are unaffected by the update. Others[4] only consider a particular class of local constraints and updates. Thus none of the techniques is applicable to our case. The n conditions we intend to check are all linear conditions, so the techniques[1, 12, 9] developed in computation geometry are also related.

But since in our case, the linear conditions change frequently, the cost of reconstructing the data structures in [1, 9] will easily exceed the benefits.

8 Conclusion

In this paper, we argue for efficiently bounding numerical error to support replicated network services. Two algorithms, Split-Weight AE and Compound-Weight AE, are proposed to bound absolute error. They can be combined to achieve good performance and low space overhead. Our Inductive RE bounds relative error by transforming it into absolute error and then apply Split-Weight/Compound-Weight AE. Exploiting the fact that V_j is an approximation of V_{final}, we are able to perform the transformation through induction and use only local information. We propose two optimizations to scale the error bounding algorithms to thousands of data items and hundreds of replicas. Through performance analysis and simulation, we show that a replicated network service using our error bounding algorithms has superior performance in terms of latency and throughput compared to a network service using traditional two-phase commit protocol.

References

[1] Pankaj K. Agarwal, Lars Arge, Jeff Erickson Paolo G. Fanciosa, and Jeffrey Scott Vitter. Efficient searching with linear constraints. In *Proceedings of the 17th ACM Symposium on Principles of Database Systems*, 1998.

[2] Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Transactions on Database Systems*, September 1990.

[3] Daniel Barbara and Hector Garcia-Molina. The demarcation protocol: a technique for maintaining linear arithmetic constraints in distributed database systems. In *Proceedings of the International Conference on Extending Database Technology*, 1992.

[4] Philip Bernstein, Barbara Blaustein, and Edmund Clarke. Fast maintenance of semantic integrity assertions using redundant aggregate data. In *Proceedings of the 6th Conference on Very Large Data Bases*, 1980.

[5] Peter O. Buneman and Eric K. Clemons. Efficiently monitoring relational databases. *ACM Transactions on Database Systems*, September 1979.

[6] O.S.F. Carvalho and G. Roucairol. On the distribution of an assertion. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 1982.

[7] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. In *International Journal of Supercomputer Applications*, volume 11(2), pages 115–128, 1997.

[8] Richard Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, University of California, Santa Cruz, December 1992.

[9] Jonathan Goldstein, Raghu Ramakrishnan, Uri Shaft, and Jie-Bing Yu. Processing queries by linear constraints. In *Proceedings of the Sixteenth ACM Symposium on Principles of Distributed Computing*, 1997.

[10] Ashish Gupta and Jennifer Widom. Local verification of global constraints in distributed databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1993.

[11] Robert Kowalshi, Fariba Sadri, and Paul Soper. Integrity checking in deductive databases. In *Proceedings of the 13th Conference on Very Large Data Bases*, 1987.

[12] Norbert Beckmannand Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1990.

[13] Narayanan Krishnakumar and Arthur Bernstein. Bounded Ignorance in Replicated Systems. In *Proceedings of the 10th ACM Symposium on Principles of Database Systems*, May 1991.

[14] Narayanan Krishnakumar and Arthur Bernstein. Bounded Ignorance: A Technique for Increasing Concurrency in a Replicated System. *ACM Transactions on Database Systems*, 19(4), December 1994.

[15] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, November 1992.

[16] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, 1998.

[17] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997.

[18] Calton Pu and Avraham Leff. Epsilon-Serializability. Technical Report CUCS-054-90, Columbia University, 1991.

[19] Calton Pu and Avraham Leff. Replication Control in Distributed System: an Asynchronous Approach. Technical Report CUCS-053-90, Columbia University, January 1991.

[20] D. Terry, K. Petersen, M. Spreitzer, and M. Theimer. The Case for Non-transparent Replication: Examples from Bayou. In *IEEE Data Engineering*, pages 12–20, December 1998.

[21] Amin Vahdat, Thomas Anderson, Michael Dahlin, Eshwar Belani, David Culler, Paul Eastham, and Chad Yoshikawa. WebOS: Operating System Services for Wide-Area Applications. In *Proceedings of the Seventh IEEE Symposium on High Performance Distributed Systems*, Chicago, Illinois, July 1998.

[22] David Wetherall. Active network vision and reality: lessions form a capsule-based system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, 1999.

[23] Haifeng Yu and Amin Vahdat. Building replicated internet services using TACT: A toolkit for tunable availability and consistency tradeoffs. In *Second International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems*, June 2000.