Lock-Free Consistency Control for Web 2.0 Applications *

Jiang-Ming Yang[1,3], Hai-Xun Wang[2], Ning Gu[1], Yi-Ming Liu[1], Chun-Song Wang[1], Qi-Wei Zhang[1]

[1]Fudan University, {ninggu, yimingliu,0572202,qiweizhang}@fudan.edu.cn
[2]IBM T. J. Watson Research Center, haixun@us.ibm.com
[3]Microsoft Research Asia, jmyang@microsoft.com

ABSTRACT

Online collaboration and sharing is the central theme of many web-based services that create the so-called Web 2.0 phenomena. Using the Internet as a computing platform, many Web 2.0 applications set up mirror sites to provide large-scale availability and to achieve load balance. However, in the age of Web 2.0, where every user is also a writer and publisher, the deployment of mirror sites makes consistency maintenance a Web scale problem. Traditional concurrency control methods (e.g. two phase lock, serialization, etc.) are not up to the task for several reasons. First, large network latency between mirror sites will make two phase locking a throughput bottleneck. Second, locking will block a large portion of concurrent operations, which makes it impossible to provide large-scale availability. On the other hand, most Web 2.0 operations do not need strict serializability – it is not the intention of a user who is correcting a typo in a shared document to block another who is adding a comment, as long as consistency can still be achieved. Thus, in order to enable maximal online collaboration and sharing, we need a lock-free mechanism that can maintain consistency among mirror sites on the Web. In this paper, we propose a flexible and efficient method to achieve consistency maintenance in the Web 2.0 world. Our experiments show its good performance improvement compared with existing methods based on distributed lock.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; H.2.4 [**Database Management**]: Systems—*concurrency, transaction processing*

General Terms

Algorithms

1. INTRODUCTION

Many popular web sites, including commercial sites and community sites (e.g. GNU, SourceForge, etc.), have multiple mirrors across the Internet to support hundreds of millions of visits per day. Mirror sites are an easy solution to scalability as users are able to choose the nearest mirrors to get faster response time.

*This paper is partly supported by National Natural Science Foundation of China (NSFC) under Grant No.90612008, No.60736020 and National Grand Fundamental Research 973 Program of China under Grant No.2005CB321905.

WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

Unfortunately, using mirror sites for Web 2.0 applications for scalability may not be effective. In the Web 2.0 environment, each user is an author and publisher. The sheer amount of user interaction, and the fact that shared data are edited simultaneously at mirror sites physically far apart, create a lot of difficulty in consistency control. In fact, performances of some popular Web 2.0 sites (e.g. Wikipedia, Open Directory Project, YouTube, etc.) can be very different across different places at different times.

Traditional distributed database systems maintain consistency by enforcing the following semantics: the result of a concurrent execution must be equivalent to a serial one [16]. A common way to achieve serialized transactions is to use *Two Phase Locking* (2PL) [7]: transactions on different objects are executed simultaneously while transactions on a same object are serialized by using locks.

In the Web environment, the main weaknesses of the above approaches are the following: (1) Due to internet latency between mirror sites, it is very costly to obtain locks and commit transactions. To ensure data integrity during the process, the system must block all other operations (including read-only operations), which reduces performance, and is unacceptable in the Web 2.0 environment. This undoes the purpose of creating mirror sites, which is to increase scalability. (2) The locking mechanism requires all mirror sites to wait for the slowest site to commit the transaction and free the lock. Thus, vulnerability of one mirror site may affect the efficiency of the entire system.

Web 2.0 users need a more flexible concurrency control mechanism. Many current studies on user collaboration [6, 11, 14] show that in order to facilitate free and natural information flow among collaborating users, it is important to allow users to concurrently edit any part of the shared document at any time. Traditional methods are unfit as concurrent operations (in transaction mode) always block one another. Thus, it is clear that lock-based mechanisms are unsuitable for mainitaining consistency of mirror sites that run Web 2.0 applications. This motivates us to explore novel approaches for consistency control in the Web 2.0 environment.

In this paper, we introduce a light-weight, lock-free approach to maintain consistency of mirrored sites. We briefly summarize our approach below.

- CAUSALITY PRESERVATION. Instead of serializability, our approach starts with preserving causality, which is regarded as the bottom line requirement in consistency control. Intuitively, if operation X's input depends on operation Y's output, then Y must be executed before X. Preserving causality is not enough for consistency control. However, because it is much less costly (it does not require locking), we use it as a starting point.

- LOCK-FREE TRANSACTIONS. The concept of transaction

and serialization gives clear semantics to concurrent operations. Although most Web 2.0 operations (concurrent edits by different users in different parts of a document) do not require strict serializability, critical operations by priority users may still need to run in transaction mode. In our work, we extend the causality theory to support transactions without using locks.

- LOCK-FREE NON-TRANSACTIONAL OPERATIONS. For concurrent, non-transactional operations, we show consistency can be achieved without using locks or incurring the cost of supporting transactional operations. In our approach, these operations are executed as soon as they are generated for faster response time. Consistency of mirrored sites is ensured by a *state transformation* technique which is able to recover mirrored data to previous states.

This paper is organized as follows. In Section 2, we describe the replicated architecture of mirrored sites. Section 3 introduces the causality concept. Lock-free consistency control for transactional and non-transactional operations are discussed in Section 4 and 5. Section 6 discusses experiments. We discuss related work in Section 7 and conclude in Section 8.

2. PROBLEM SETTING

In this section, we describe the consistency control problem in the web environment.

2.1 Representations of Data and Operations

Data shared in mirrored sites can be in varied forms. In this paper, we assume the shared data on mirrored sites are XML documents. Consequently, operations on the data are expressed by XML queries and updates.

There are two W3C standards for XML queries: XPath [17] and XQuery [18]. To process an XPath query, we start from the root node, match all paths satisfying the XPath pattern, and return the leaf nodes on this path. An XQuery is represented as a FLWR (For-Let-Where-Return) pattern, and processed in two phases: a twig pattern matching phase defined by FLW and a result construction phase defined by R. XQuery is an extension to XPath and is more powerful than XPath.

Update operations allow users to modify XML documents. Currently, there is no standard for *Update* operations. We use FLWU, an XQuery extension introduced by Tatarinov et. al. [15], to express updates. An FLWU is processed in two phases: in the FLW phase, multiple subtrees that match the pattern specified by FLW are generated, that is similar with the first part of XQuery [18], then in the U (update) phase, certain nodes of the sub-trees are modified. The actions of second phase (U) are based on the result of the first phase (FLW), thus we call the results of the first phase the *execution set* of the operation. In mirrored sites, so long as the FLW process gets the same *execution set*, the final results should be the same across all sites.

Definition 1. (Execution Set). Following previous definitions in [15, 18], we call the results returned in the first phase (FLW) of an FLWU operation the execution set or ES of the FLWU operation.

```
<Root>
    <book @title="Introduction to Algorithm">
        <category>CS</category>
        <tag>Hot</tag>
    </book>
    <book @title="Advanced Statistical Learning">
        <category>UnKnow</category>
    </book>
    <book @title="Linear Algebra">
        <category>Math</category>
    </book>
</Root>
```

In our paper, we use the above XML document consisting 3 books, and the four *Update* operations, U_1, U_2, U_3 and U_4 on the XML document as a running example for explaining consistency control in mirrored sites.

Operation U_1:
Change the *title* "Advanced Statistical Learning" to "Statistical Learning".

```
FOR $title in
  /root//title
WHERE $title =
 "Advanced
 Statistical Learning"
UPDATE $title {
  REPLACE $title WITH
  "Statistical Learning"
}
```

Operation U_2:
Set the *category* of the "Linear Algebra" *book* to "Math".

```
FOR $book in /root/book,
 $title = $book/title,
 $category = $book/category
WHERE $title =
    "Statistical Learning"
UPDATE $book {
  REPLACE $category WITH
  <category>Math</category>
}
```

Operation U_3:
Add a *discount tag* to *book*s in "Math" category.

```
FOR $book in /root/book,
 $category =
    $book/category
WHERE $category = "Math"
UPDATE $book {
  INSERT
  <tag>Discount</tag>
}
```

Operation U_4:
Set the category of the "Linear Algebra" book to "CS".

```
FOR $book in /root/book,
 $title = $book/title,
 $category = $book/category
WHERE $title =
    "Linear Algebra"
UPDATE $book {
  REPLACE $category WITH
  <category>CS</category>
}
```

Figure 1: Four Update Operations

2.2 Architecture

We assume that shared data on each mirrored site is represented as a huge XML document. Our goal is to achieve good query/update response and support unconstrained collaboration. This leads us to adopt a replicated architecture for storing shared documents: the shared document is replicated at the local storage of each mirrored site. A user can operate on the shared data at any replica site, but usually at the site closest to the user. Non-transactional user operations are executed immediately at that site, and then dispatched to other sites. Transactional operations require more complicated scheduling. We discuss both cases in this paper.

As an example, Figure 2 shows three replicas in a groupware environment: SR_1, SR_2 and SR_3. Replica SR_3 receives and executes two operations U_1 and U_2. We assume U_1 and U_2 are causal operations, since U_2 operates on the results of U_1, and they are initiated from the same site, possibly by the same user. Operations U_3, U_4 and T_1 arrive at roughly the same time on SR_1, SR_2, and SR_3 respectively. Here, we assume T_1 is a transaction. U_3, U_4 and T_1 are concurrent operations.

To be clear, given a user operation, we call the replica receiving the operation the *local replica* and other replicas *remote replicas*. And for the local replica, we also call this operation its *local operation*, and operations dispatched from other replicas *remote operation*. In Section 3, we define a causal (partial) ordering relationships on operations based on their generation and execution sequences.

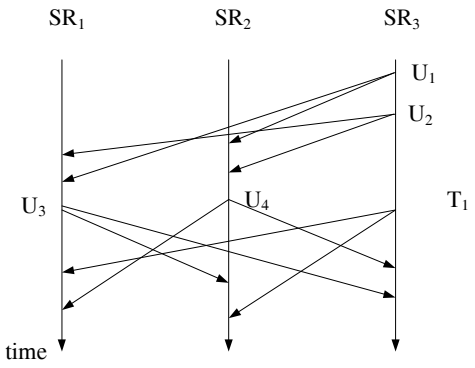

Figure 2: The Replicated Architecture

2.3 The Need for Consistency Control

We use three examples to illustrate the need for consistency control based on the above settings.

1. (CAUSAL RELATIONSHIPS) Consider replica SR_3, where U_1 changes the title of the book from "Advanced Statistical Learning" to "Statistical Learning", and then U_2 changes the category of "Statistical Learning" to "Math". Obviously, there is a causal relationship between U_1 and U_2, as the user intends to execute U_2 based on U_1's result. Thus, operations with causal relationships must be executed in the right order.

2. (TRANSACTIONS) Operation T_1 is critical and must be executed in transactional mode. Here, T_1 arrives in different order with regard to U_3 and U_4: on SR_1, it arrives after U_3, on SR_2, after U_3 and U_4, and on SR_3, it arrives first. To guarantee transaction semantics, we must ensure T_1 is executed in the same order on all sites with regard to its concurrent operations.

3. (CONCURRENT OPERATIONS) Operations U_3 and U_4 are non-transactional and do not have causal relationships. But they still need consistency control. At SR_1, U_3 adds a "Discount" tag to books in "Math" category. But at SR_2, because U_4 arrives earlier and changes one of the "Math" books to "CS" category, U_3 will add the "Discount" tag to the "Machine Learning" book alone. Thus, the results on SR_1 and SR_2 are not consistent.

The three cases require different handling in consistency control. In the rest of the paper, Section 3 discusses consistency control issues for causal relationships, Section 4 for transactions, and Section 5 for non-transactional concurrent operations. Overall, we generalize the above cases and show our method ensures consistency in a replicated Web architecture.

3. CAUSALITY PRESERVATION

In this section, we present the concept of causality, as well as a known solution for causality preservation (ensuring causal operations such as U_1 and U_2 are executed in the same order across all replicated sites) without using locking mechanisms. The techniques we discuss here will be used as building blocks for ensuring consistency of transaction and non-transactional concurrent operations.

Consider operations U_1, U_2 in Figure 1, and replica SR_3 in Figure 2. Operation U_1 changes a book title from "Advanced Statistical Learning" to "Statistical Learning", and then U_2 sets the category of book "Statistical Learning" to "Math". Obviously, U_2 is causally after U_1 since the user intends to execute U_2 based on U_1's result. However, because of internet latency, replica sites such as SR_1 may receive U_2 before U_1. To address this problem, we apply the causality preservation strategy.

Definition 2. (Causal Ordering Relation "$\rightarrow$"). Given two operations O_a and O_b from local replica sites i and j respectively, we have $O_a \rightarrow O_b$, if and only if (1) $i = j$, and O_a is generated before O_b is generated; (2) $i \neq j$, and O_a is executed on site j before O_b is generated; (3) there exists an operation O_x, such that $O_a \rightarrow O_x$ and $O_x \rightarrow O_b$.

Definition 3. (Concurrent Relation "$\|$"). Given two operations O_a and O_b, we say O_a and O_b are concurrent or $O_a \parallel O_b$ iff neither $O_a \rightarrow O_b$, nor $O_b \rightarrow O_a$.

Figure 3 illustrates the 3 cases of causality given by Definition 2.

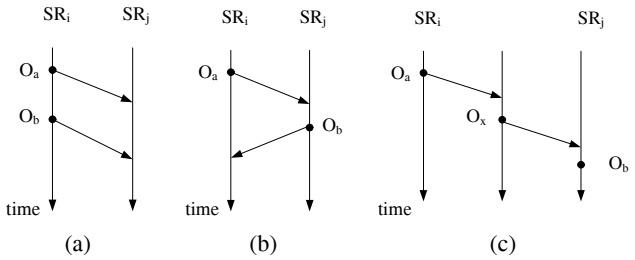

Figure 3: Causal Relationships

Intuitvely, in order for every site to execute causal operations such as U_1 and U_2 in the same order, all they need to know is that U_1 precedes U_2 on the site they are generated, so if U_2 arrives first, they will wait until U_1 arrives. But in a distributed environment, it is difficult to implement a global, precise clock that informs each site the precedence of operations. The challenge is thus to define the "happened before" relation without using physical clocks.

To do this, we use a timestamping scheme based on State Vector (SV) [6, 14]. Let N be the number of replica sites (we assume N is a constant). Assume replica sites are identified by their unique replica IDs: $1, \cdots, N$. Each site i maintains a vector SV_i with N components. Initially, $SV_i = \langle 0, \ldots, 0 \rangle$. After it executes a remote operation dispatched from site j, it sets $SV_i[j] = SV_i[j] + 1$.

In our approach, all (non-transactional) operations are executed immediately after their generation (operations' localized execution ensures good response and supports unconstrained collaboration). Then the operation is dispatched to remote sites with a timestamp (state vector) equal to the local state vector. Specifically, operation O generated from site j is dispatched to other replicas with a state vector timestamp $SV_O = SV_j$.

Definition 4. (Execution Condition). Assume site i executes an operation O and then dispatches it to other replicas with timestamp SV_O. O is causally ready for execution at site j ($i \neq j$) if the following conditions are satisfied:

1. $SV_O[i] = SV_j[i] + 1$
2. $SV_O[k] \leq SV_j[k]$, for all $1 \leq k \leq N$ and $k \neq i$.

Intuitively, the first condition ensures that O is the next operation from site i, that is, no operations originated from site i before O have been missed by site j. The second condition ensures that all operations originated from other sites and executed at site i before the generation of O have been executed at site j already. The two conditions ensure that O can executed at site j without violating causality. [14]

THEOREM 1. *The execution condition ensures that causality as defined in Definition 2 is preserved.*

Take Figure 2 as an example. Assume the state vector of each site is $\langle 0, 0, 0\rangle$ initially. When U_2 is dispatched to SR_1, it carries a state vector with $SV_{U_2}[3] = 2$, because SR_3 has executed 2 local operations U_1 and U_2. However, $SV_1[3] = 0$ because site SR_1 has not executed any operation from SR_3. Thus the first execution condition, $SV_{U_2}[3] = SV_1[3] + 1$, is not satisfied, which means U_2 cannot be executed on SR_1 yet. Now U_1 arrives at SR_1 with state vector $SV_{U_1}[3] = 1$. It is executed immediately and $SV_1[3]$ is increased by 1. Then, U_2 is ready for execution because $SV_{U_2}[3] = SV_1[3] + 1$. Thus, U_1 and U_2 will be executed on SR_1 and SR_2 in the same order as they are executed on SR_3. We have thus solved the first problem in Section 2.3.

We omit the proof of Theorem 1 in the paper. Details related to this result can be found in [6, 10]. Specifically, Lamport [10] first introduced timestamp to describe the causal relationship. C. A. Ellis [6] first proposed an execution condition to preserve the causality among operations.

4. TRANSACTIONS

Some critical operations need to be executed in the transaction mode to guarantee integrity across multi-replicas. For example, when doing a money transfer, if the money is debited from the withdrawal account, it is important that it is credited to the deposit account. Also, transactions should not interfere with each other. For more information about desirable transaction properties, please refer to [7, 16]. In this section, we discuss a light-weight approach to achieve this in a replicated environment.

4.1 Semantics

The transaction model is introduced to achieve concurrent transparency [7, 16]. In the transaction semantics, although the system is running many transactions concurrently, the user can still assume that his/her transaction is the only operation in the system. To achieve the same goal in a distributed environment, we must guarantee a transaction is serializable with other operations (both transactional and non-transactional operations).

Definition 5. (Serialized Transaction). Let T be a transaction, and O be an operation (O may or may not be a transaction). Transaction T is a serialized transaction only if either O is executed before T in all sites, or O is executed after T in all sites.

According to our definition of concurrent operations (Definition 3), it is easy to see that a serialized transaction does not have concurrent operations. This ensures that transaction integrity is guaranteed across multiple replicas. But in a fully replicated architecture, it is very hard to achieve global serialization. In the following, we describe in detail how we achieve global serialization in such a replicated environment.

4.2 A causality based approach

The state vector based approach for preserving causality is elegant and light-weight (Section 3). In this section, we implement transaction semantics based on causal relationships.

Of course, existing causal relationships are not enough to enforce serialization of operations across all sites. To solve this problem, we create dummy operations (NOOPs) to introduce a rich set of *virtual* causal relationships. With these virtual relationships, we implement transaction semantics.

Assume T_1 is submitted to replica SR_3. Because T_1 is a transaction, it is not executed immediately at its local replica, as we must enforce transaction semantics across all sites. Each site SR_i, including SR_3 itself, creates a dummy NOOP$_i$ upon receiving T_1, and dispatches NOOP$_i$ to other sites. This is shown in Figure 4, where a solid line represents the dispatching of a transaction from a local site to a remote site, a dashed line represents the dispatching of a NOOP, and a circle represents the dispatching of a NOOP to the local site itself.

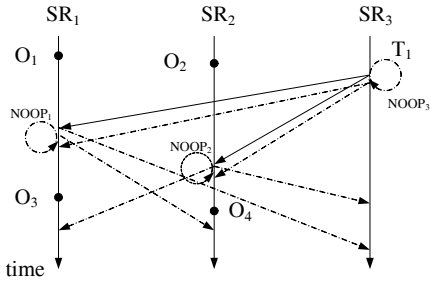

Figure 4: Flexible Transaction Model

The NOOPs introduce a rich set of causal relationships. Assume $NOOP_1$ is initiated at SR_1 between its local operations O_1 and O_3, and $NOOP_2$ is initiated at SR_2 between its local operations O_2 and O_4. The NOOPs create the following causal relationships:

$$O_1 \rightarrow NOOP_1 \rightarrow O_3 \text{ and } O_2 \rightarrow NOOP_2 \rightarrow O_4$$

Under the execution condition, $NOOP_1$ and $NOOP_2$ will be executed after O_1 and O_2, while O_3 and O_4 will be executed after $NOOP_1$ and $NOOP_2$. At any site, we execute T_1 immediately after all NOOPs corresponding to T_1 are executed. It is easy to see that at each site,

- O_1 and O_2 are executed before T_1 (the causal relationships enforce that O_1, O_2 are executed before the NOOPs, and since T_1 is executed after the NOOPs, T_1 must be executed after O_1 and O_2).
- O_3 and O_4 are executed after T_1 (the causal relationships enforce that the NOOPs are executed before O_3 and O_4, and since T_1 is executed immediately after the NOOPs, T_1 must be executed before O_3 and O_4).

The above procedure works not only in this case. It actually enforces serializability in all cases. Intuitively, when NOOP$_i$ is generated on SR_i, it divides concurrent operations local to SR_i into two sets: $Before_i$ and $After_i$, which are concurrent operations that arrive before and after NOOP$_i$ at SR_i, respectively. The causality introduced by NOOP$_i$ at site SR_i can be expressed as

$$Before_i \rightarrow \text{ NOOP}_i \rightarrow After_i$$

On any site SR_i, at the time when T is *causally ready* to execute – i.e., all NOOPs have arrived and are *causally ready* to execute – we can be assured that all concurrent operations in $B = Before_1 \cup \cdots \cup Before_N$ have been executed on SR_i and none concurrent operations in $A = After_1 \cup \cdots \cup After_N$ have been executed on SR_i. Note that $A \cup B$ are the entire set of concurrent operations and $A \cap B = \emptyset$, as any operation is local to one and only one replica. Thus, for any operation O, either $O \in A$ or $O \in B$ must be true, which means O either executes before T or after T on any site SR_i. Thus, T is serialized.

There is one pitfall in the above reasoning: it is possible that on a site, the set of NOOPs will never be *causally ready* to execute.

This happens when there are more than one concurrent transactions. Figure 5 shows an example. Two concurrent transactions arrive at replica sites in different order. Let NOOP_i^T denote the NOOP operation generated for transaction T at replica SR_i. At site SR_1, we have $\text{NOOP}_1^{T_2} \rightarrow \text{NOOP}_1^{T_1}$, and at site SR_3, we have $\text{NOOP}_3^{T_1} \rightarrow \text{NOOP}_3^{T_2}$. Thus, neither the entire set of { $\text{NOOP}_i^{T_1}$ }, nor the entire set of { $\text{NOOP}_i^{T_2}$ }, will ever be causally ready for execution: the two transactions block each other.

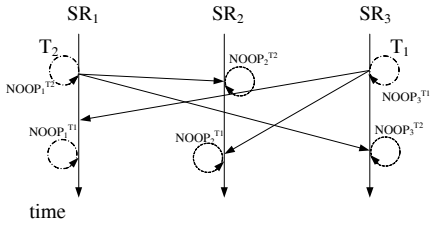

Figure 5: Two Concurrent Transactions

Our goal is to achieve global serialization. Suppose finally we have $T_1 \rightarrow T_2$. Clearly, causality in the form of $NOOP_i^{T_2} \rightarrow NOOP_i^{T_1}$ are inconsistent with this order. The solution is to simply remove such causalities. However, this requires all sites to agree on the final order $T_1 \rightarrow T_2$.

The State Vector timestamping mechanism we introduced in Section 3 creates a virtual global clock, so that each relica site agrees on the "happened-before" relationship for causal operations. This gives us a way to order non-transactional operations. Transactions, however, do not have State Vector timestamps (they are not executed immediately on local replicas). In our case, transactions trigger non-transactional NOOP operations, which enable us to extend the order among non-transactional operations to transactions.

A non-transactional operation is assigned a state vector timestamp at the site it is generated, and dispatched to all other replicas with the same state vector timestamp. We define a $TOrder$ function [14] based on such timestamps.

Definition 6. (TOrder: total order for non-transactional operations). Let O_a and O_b be two non-transaction operation with local replicas a and b. $O_a \prec O_b$ iff (1) $sum(SV_a) < sum(SV_b)$, or (2) $a < b$ when $sum(SV_a) = sum(SV_b)$, where $sum(SV) = \sum_{i=1}^{N} SV[i]$.

It is easy to see that, because each non-transactional operation has the same state vector timestamp at each replica, the total order among non-transactional operations defined above is agreed by all replicas.

We then extend TOrder to create a total order among transactions.

Definition 7. (TOrder: total order for transactions). Let T_1 and T_2 be two transactions. $T_1 \prec T_2$ iff $NOOP_1^{T_1} \prec NOOP_1^{T_2}$.

With a total order among transactions, we can solve the problem above. Algorithm 1 handles local operations, and Algorithm 2 handles operations dispatched from other sites. Finally, Theorem 2 shows that they achieve serializability.

THEOREM 2. *Algorithm 1 and Algorithm 2 ensures that transactions are serialized.*

PROOF. See Lemma 1 and Theorem 1 in the Appendix. □

Algorithm 1 Local(i), algorithm on site SR_i for local transactions

1: **Wait Until** SR_i receives an operation O
2: **if** O is a transaction **then**
3: Generate a $NOOP_i^O$ for O with state vector $SV_{NOOP_i^O} = SV_i$.
4: Dispatch O to other replica sites.
5: Dispatch $NOOP_i^O$ to all replica sites (including SR_i)
6: **end if**
7: Goto 1

Algorithm 2 Remote(i), algorithm on site SR_i for remote transactions

1: **Wait Until** SR_i receives an operation O
2: **if** O is a transaction **then**
3: Generate $NOOP_i^O$ for O with state vector $SV_{NOOP_i^O} = SV_i$.
4: Dispatch $NOOP_i^O$ to all replica sites (including SR_i)
5: **end if**
6: **if** O is $NOOP_j^T$ **then**
7: remove all inconsistent causalities
8: **if** all NOOPs for T have arrived **then**
9: Execute all $NOOP_j^T$ and T, for all SR_j
10: $SV_i[j] \Leftarrow SV_i[j] + 1$, for all SR_j
11: **end if**
12: **end if**
13: Goto 1

5. CONSISTENCY CONTROL

In this section, we discuss how to perform consistency control for concurrent, non-transactional operations.

5.1 Overview

To achieve faster response time, non-transactional operations are executed at their local replicas immediately after they are submitted. Hence, the execution order of concurrent operations are different at different sites. As we show in the 3rd example in Section 2.3, concurrent operations may create inconsistent results on different sites.

To ensure consistency without using locks, we propose an XML storage model that allows us to recover XML documents to states before a previous concurrent operation was executed. Figure 6 gives an example. Figure 6(a) shows the XML document on site SR_2 after the execution of operation U_4, which changes the category of "Statistical Learning" from "CS" to "Math". Later, SR_2 receives operation U_3, and it decides that U_3 has an earlier timestamp than U_4 (as indicated by TOrder). Then, it retraces the state of XML document to the state shown in Figure 6(b), which is the state before U_4 is executed. Finally, U_3 is executed on the retraced XML document, which adds a "Discount" tag to *both* "Statistical Learning" and "Linear Algebra" since both of them are in category "Math", then U_4 is executed to change the category of "Statistical Learning" to "CS". The result XML document, which is shown in Figure 6(c), will be the same as that on site SR_1, where U_3 is generated and executed first.

The high level procedures for the local site and remote site are shown in Algorithm 3 and 4 respectively. Both of the algorithms are concerned with the non-transactional operations, as we have discussed transactions in Section 4. Algorithm 3 executes a local operation immediately once it is generated. Algorithm 4 retraces to earlier states when remote operates arrive with an earlier timestamp. Note that in neither of the two algorithms, the operations are

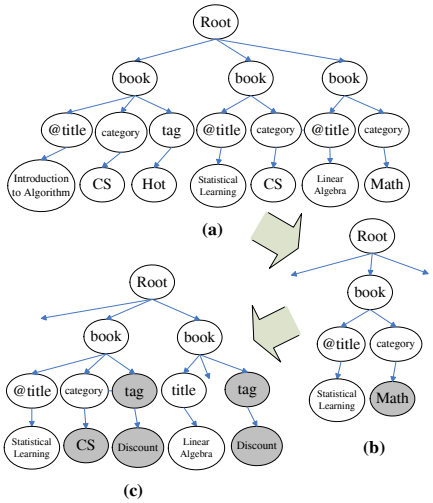

Figure 6: With regard to site SR_2 in Figure 2, (a) is the XML document after the execution of U_4. We retrace to the state shown in (b), which is before the execution of U_4, and (c) is the result of executing U_3 and then U_4 on the retraced state.

Algorithm 3 Local(i): algorithm on site SR_i for local operations

1: $O \leftarrow$ next operation generated at site SR_i
2: **if** operation O is not in transaction mode **then**
3: Execute O directly, and dispatch it to other sites with state vector SV_i as its timestamp.
4: $SV_i[i] \Leftarrow SV_i[i] + 1$
5: **end if**

Algorithm 4 Remote(i): algorithm on site SR_i for remore operation which comes from site SR_j

1: $O \leftarrow$ next operation satisfying the execution condition from remote site SR_j
2: **if** operation O is not in transaction mode **then**
3: Retracing(Doc, SV_O)
4: ES $\leftarrow$ query(Doc, SV_O) /* FLW phase of an FLWU */
5: Update(Doc, O, ES) /* U phase of an FLWU */
6: $SV_i[j] \Leftarrow SV_i[j] + 1$
7: **end if**

forced to wait.

In the rest of the section, we describe in detail the storage model and the retracing algorithm.

5.2 The Storage Model

In this section, we introduce an efficient XML query processing model that supports retracing.

Range Labels and Inverted Lists

XML queries are processed by twig pattern matching [2]. Several approaches exist, including the relational approach [15], the native XML approach [2, 3], and a mixture of the two [19]. In general, native XML data representation has better query efficiency. In this paper, we base our approach on native XML data storage.

The core of XML query processing relies on one fundamental operation: determine the ancestor-descendent relationship between two nodes in an XML document. An efficient way of telling whether one node is an ancestor (descendant) of another node is to use interval-bases labels [4]. In the example shown in Figure 7 (which represents the XML document on site SR_2 after the execution of U_4 but before the execution of U_3), each node n is labeled by an interval $(start_n, end_n)$. Using the labels, we can immediately tell the ancestor-descendent relationships: node x is a descendant of node y if $start_x \in (start_y, end_y)$. The labels in Figure 7 are real values in the range of [0, 1], which allows us to subdivide the range when new nodes are inserted.

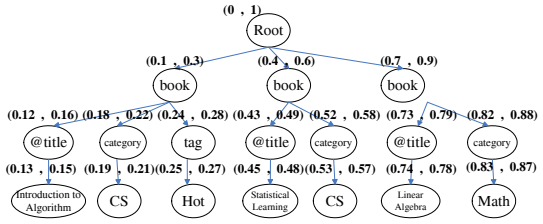

Figure 7: The range labels of XML document nodes

To support efficient twig matching, we employ inverted lists to organize labeled nodes. More specifically, for each element/attribute/value in the XML document, we create a linked list which includes the labels of nodes that have the same element/attribute/value. For instance, in Figure 8, the linked list for the *book* element has three members: $(0.1, 0.3)$, $(0.4, 0.6)$, $(0.7, 0.9)$, corresponding to the 3 books in Figure 7.

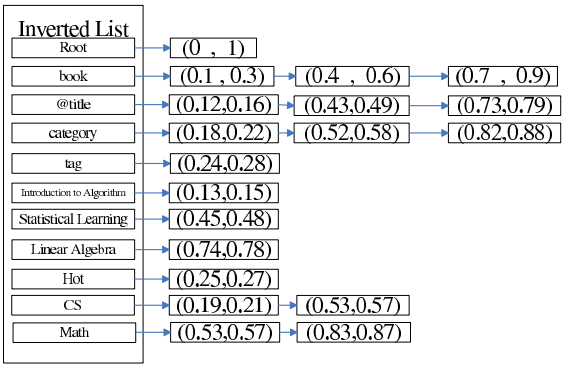

Figure 8: Inverted list for XML in Figure 7

The inverted lists enable us to efficiently find the nodes involved in a query and match the twig pattern by joining the nodes. The inverted lists, however, represent a certain state of the XML document only. In order to retrace its temporal history, i.e., states of the XML document before certain update operations are executed, we need some additional mechanisms.

Timestamps

Assume an XML document has gone through a series of updates $u_1, \cdots, u_i, \cdots, u_k$ (the updates in the series follow TOrder). Now a new update u' comes in, and $u' \prec u_i$. We recover the document to the state before u_i is applied, and then apply $u', u_i, \cdots, u_k$ on the recovered documents.

In order to recover the document to a previous state, we assign each node in the inverted lists a pair of timestamps `(create, delete)`, where `create` and `delete` are the TOrder number of the operation that creates and deletes the node. It is clear that `create` $\prec$ `delete`. In other words, when an operation deletes a node, instead of removing it from the inverted lists, we assign it a `delete` timestamp, but keep it there. In Section 5.4, we show

that we only need to keep a limited number of states in the inverted lists – when an operation has been executed on all sites, it will have no concurrent operations, and its effect is made permanent on the inverted lists. In the next section, we discuss how retracing and query processing are performed on timestamped inverted lists.

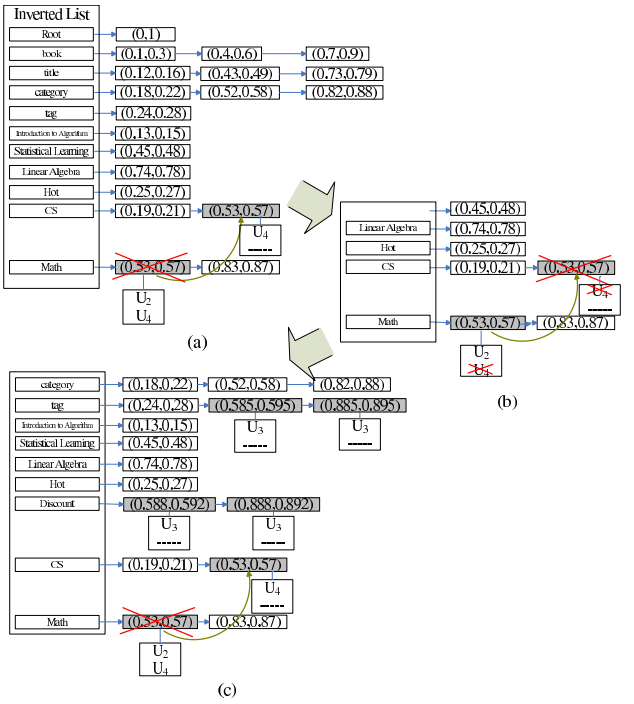

Figure 9: State changes in inverted lists

5.3 Query Processing

In this section, we discuss how to augment traditional XML query processing engines to support retracing, querying, and updating on timestamped inverted lists.

Retracing

The `(create, delete)` timestamps enable us to retrace the temporal history of an XML document. Let us first examine an example. The inverted lists shown in Figure 9 correspond to the XML documents in Figure 6. Figure 9(a) shows the inverted list on site SR_2 after the execution of U_1, U_2, and U_4. The *Math* node $(0.53, 0.57)$ is created by operation U_2 and deleted by operation U_4. Thus, it has timestamp (U_2, U_4)[1]. Since U_4 changes the *Math* node to a *CS* node, a new node with timestamp $(U_4, -)$ is created in the inverted list of *CS*. When U_3 arrives, we find that, according to the total order, $U_3 \prec U_4$. We thus retrace the steps to before U_4 is executed, as we show in Figure 6(b). Since the *Math* node $(0.53, 0.57)$ is timestamped (U_2, U_4), which means it is deleted by U_4, its timestamp is rolled back to $(U_2, -)$, making the node current again. On the other hand, the *CS* node with timestamp $(U_4, -)$ will be removed as it is created at time U_4, which is a future timestamp at time U_3.

In summary, the inverted lists keep mutltiple versions or multiple states for the category of the book "Statistical Learning". This is shown in Figure 10. First, U_2 changes its category from *Unknown* to *Math*, then U_4 changes it again to *CS*. Thus, the three states can be arranged into a sequence. It is clear that there is no overlap in their timestamps, and at any time only one of them is valid.

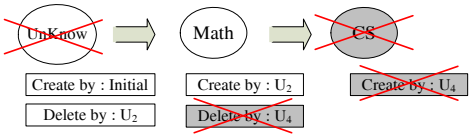

Figure 10: Dependence Relationship

In general, only nodes whose timestamp (t_{create}, t_{delete}) satisfies $t \in (t_{create}, t_{delete})$ are valid at t. For the example above, given U_3, whose timestamp precedes that of U_4, we know only the *Math* state is valid, and the *Unknown* and *CS* states are invalid.

Querying

A common way to tackle twig pattern matching is to decompose the twig pattern into a set of parent-child and ancestor-descendant relationships, query each relationships independently, and then join the results.

We introduce an algorithm called State-Join (Algorithm 5) to process basic parent/child relationships, "A/D", or ancestor /descendant relationships, "$A//D$". The algorithm takes into consideration the states of each XML node defined by its timestamp. As finding the parent-child and ancestor/descendant relationships is the most fundamental operation in XML query processing, it can be used by other XML query algorithms, such as the stack-based algorithm known as Stack-Tree-Desc [2], for twig-pattern matching in a transparent manner.

As the example shown in Figure 8 (which represents the XML document on site SR_2 after the execution of U_4 but before the execution of U_3), in order to process U_3, we first fetch the inverted list of "book", "title", "category" and "Math". Though the category of book "Linear Algebra" has been changed into "CS", the original node (0.53, 0.57) still exists in the inverted list of "Math" and is timestamped (U_2, U_4). According to the total order, $U_2 \prec U_3 \prec U_4$, which means the node is valid as far as U_3 is concerned. Since timestamp checking has $O(1)$ cost, the State-Join algorithm will not increase the complexity of the original Stack-Tree-Desc algorithm.

Algorithm 5 State-Join("A/D" or "$A//D$"). Given a state vector timestamp SV, it returns the *execution set* for sub-query "A/D" or "$A//D$" valid at timestamp SV.

1: $AList, DList \Leftarrow$ the list of potential ancestors or descendants with tag name A or D in the inverted lists, both in sorted order of StartPos.
2: $\forall a$, $a \in AList$, delete a from $AList$ if $SV \notin (a.t_{create}, a.t_{delete})$
3: $\forall d$, $d \in DList$, delete d from $DList$ if $SV \notin (d.t_{create}, d.t_{delete})$
4: Return Stack-Tree-Desc($AList$, $DList$) as ES

Updating

An update operation may change nodes in the inverted lists in two ways: (1) Create new nodes with appropriate range labels and timestamps; (2) Revise the timestamps of existing nodes. The algorithm can be expressed in Algorithm 6.

Instead of modifying a node, what an update operation really does is to create a new state for the node in the state-based inverted

[1]For simplicity, we use U_2 to denote the TOrder number of U_2, which is used as timestamp.

Algorithm 6 Update(Doc, O, ES), execute the U process of operation O based on *execution set* ES.

1: **for all** node n be revised by O in ES **do**
2: **if** n is inserted by O **then**
3: Create a new range label interval $(start_O, end_O)$ for the content of O by Range-Scan function [8].
4: **end if**
5: Revise the states related to O according to the description in "Retracing" part of Section.5.3
6: **end for**

list, and leave the original node intact (except changing its timestamp). For example, in order to execute U_3 at the state of Figure 8, it (i) creates two new nodes (0.585, 0.595) and (0.885, 0.895), and insert them into the inverted list for "tag"; (ii) creates two new nodes (0.588, 0.592) and (0.888, 0.892), and insert them to the inverted list for "Discount"; and (iii) set the timestamps for all of them to (U_3, -).

XML documents are sensitive to the order of sibling nodes under a parent node. If two operations insert two nodes under a same parent node, we must enforce that their order is the same at all sites. We again use the $TOrder$ function to determine the sibling order of nodes inserted by concurrent operations. A serious treatment of this problem may use techniques as the scan function [8] to achieve global consistentancy for insert operations in a linear structure. We omit the proof here. Information for this result can be found in [8].

5.4 The Size of the Inverted Lists

To support rollback, the inverted lists keep multiple states of the documents. One question is, will the size of the inverted lists keeps growing? The answer is negative. The reason is that, when an operation has executed on all replicas, it will no longer be concurrent to any other operation. So can purge all timestamps related to this operation. In our approach, we store operations in an Operation History List (OHL). Each time a replica receives a new remote operation, it will update OHL. When an operation is executed on all replicas, it will be removed from OHL and its related state information will be removed from the inverted list. If it deleted some nodes, we remove those nodes permanently. Since the amount of concurrent operations is limited, the amount of the states we maintain in the inverted list is limited.

5.5 Convergence

Our approach ensures faster response time. Notably, each non-transactional concurrent operation is executed immediately on its local replica. More generally, concurrent operations are executed in different order on different replicas. This means inconsistency exists across mirrored sites. However, we guarantee that at any moment, if all operations have been executed on all replicas, the mirrored data on each replica converges to a same state. In essence, the convergence property ensures the consistency of the final results at the end of a cooperative editing session [14].

THEOREM 3. *At any time when all existing operations have been executed on all replicas, each replica has the same mirrored data (consistent).*

PROOF. See Lemma 2 and Theorem 2 in the Appendix. □

6. EXPERIMENTS

In order to evaluate the efficiency of XML Updates, we build a prototype system and compare it to lock-based distributed system. The experiment is conducted with 100Mbps LAN, 2-4 sites, CPU

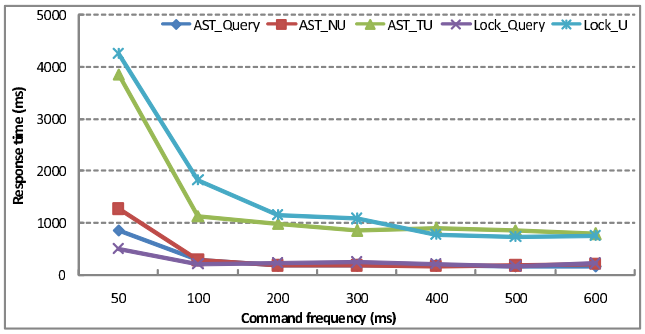

Figure 11: Sending Command Frequency

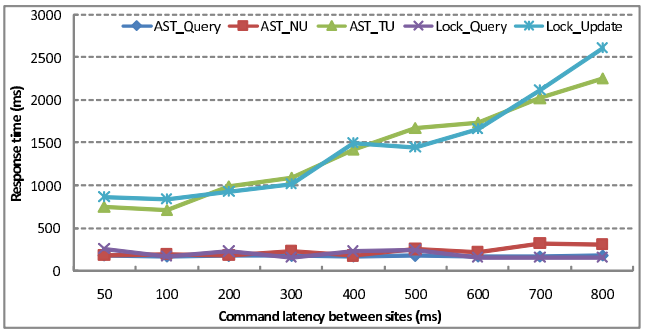

Figure 12: Network Latency Between Sites

P4 2.4G, main memory 1G, and Windows 2003. The prototype system and the lock-based distributed system are both implemented By Java (Version 1.6.0-b105). We adjust operations' frequency to simulate network latency.

Experiment data comes from DBLP website [1], and current size is 133MB. We create 70 *queries* and 70 *updates* and execute them repeatedly. An additional computer is used to submit the above operations to each site with given frequency, and record the average responding time. The process of experiments are described as follows.

There are three types of operations: *Query*, *Non-Transaction Update*, and *Transaction Update*. We label them as AST_{Query}, AST_{NU} and AST_{TU} respectively. For lock-based distributed system, all operations are in transaction mode, and it has only two types of operations: $Lock_{Query}$ and $Lock_{Update}$.

First, we evaluate the effect of different operation frequencies. The parameters are: 3 sites, the proportions of AST_{Query}, AST_{NU} and AST_{TU} are 90%, 8% and 2%, respectively, network latency between two sites is 200ms. The result is shown in Figure 11:

When the frequency is every 50ms or 100ms an operation, since operations can not be executed immediately, they were blocked in the waiting queue. AST_{TU} and $Lock_{Update}$ involves more dependancies than AST_{NU}, AST_{Query} and $Lock_{Query}$ do, so the response time of AST_{TU} and $Lock_{Update}$ rose rapidly. On the other hand, when the frequency drops to 200ms to 600ms an operation, the response time of AST_{NU} is similar to query operation and also far below AST_{TU} and $Lock_{Update}$. By using AST_{NU} instead of AST_{TU} and forcing more operations executed at local site we can significantly improve the response time.

Second, we evaluate the effect of different network latencies. The parameters are: 3 sites, the proportions of AST_{Query}, AST_{NU} and AST_{TU} is 90%, 8% and 2% and the sending command frequency is 200ms. The network latency between sites are simulated

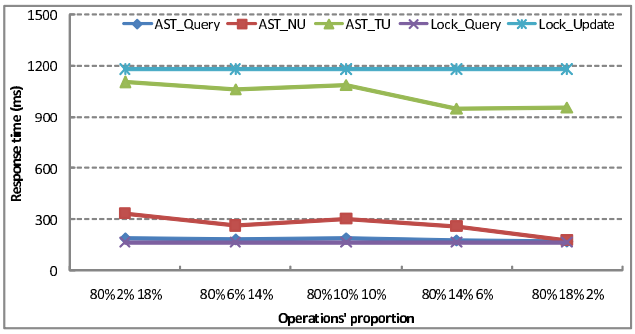

Figure 13: Operations' Proportion

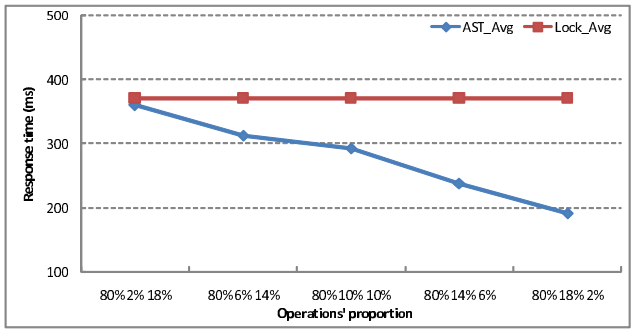

Figure 14: Effect for Average Responding Time in Different Operations' Proportion

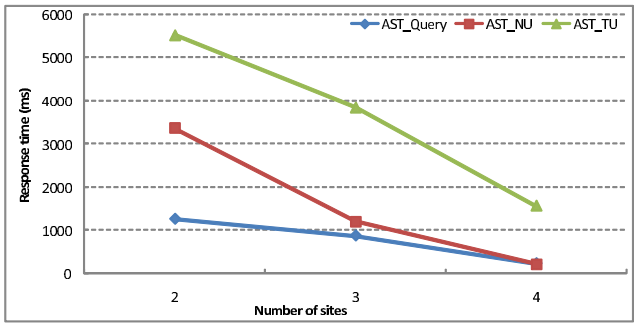

Figure 15: Number of Sites

by force a given delay. The result is shown in Figure 12.

As latency increases, the only notable growth is for AST_{TU} and $Lock_{Update}$, as they need all sites participation. As noted before, one of the benefits to deploy more sites is to enable the user to select the nearest site to access and get higher speed and shorter respond time. So by relying on AST_{NU}, which execute locally, we have a good way to achieve this goal. But the lock-based update operations need to request locks from all replicated sites and will block all other conflict operations. Even thought more sites are deployed, the respond time not only depends on the latency between the user and his/her entry site but also depends on the network latency among all replicated sites. Further more, the network latency in internet is unstable and inevitable.

Third, we evaluate the effect of compositions of different types of operations. Similarly, the parameters are: 3 sites, network latency between sites is 200ms, and the frequency is 200ms per operation. The results are shown in Figure 13 and Figure 14.

The figures show, as the ratio of AST_{NU} increases, the conflicts between AST_{TU} decreases. It lowers the responding time of both AST_{NU} and AST_{TU}. The figures also show that through increasign the ratio of AST_{NU}, we can significantly reduce the average response time.

Finally, we evaluate the effect of number of sites. The parameters are: network latency is 200ms, the average frequency is 50ms per operation, and the proportions of AST_{Query}, AST_{NU} and AST_{TU} operation are 90%, 8% and 2%. The result is shown in Figure 15:

To evaluate the effect for different number of sites, we choose the frequency to 50ms. The result shows as the number of sites increases, the average response time decreases.

In summary, the four experiment results show that the algorithm meet our expectations. By executing AST_{NU} locally and avoiding the dependance of network latency, we can make better load balance and achieve high-speed access and shorter respond time.

7. RELATED WORK

Our approach is similar in spirit to some existing non-locking methods [5, 9, 12, 14] that rely on operation transformation for consistency maintenance, where an operation is executed immediately at its local site and then dispatched to other remote sites. By including or excluding some effects of concurrent operations, a remote operation finds its correct state to execute. However, a major difference is that these approaches only focus on atomic operations (e.g. inserting or deleting a atomic object) no matter which structure they used (liner or tree). They do not support query or update with conditions, nor do they support transactions.

There are a few multi-replicas consistency maintenance solutions. Bayou [13] implements a multi-replicas mobile database, but for concurrent operations, its repeated undo and redo lead to huge system cost. TACT [13] tries to limit the differences between replicas, but when a replica exceeds the limit, operations will be blocked and response time will increase. TSAE [13] uses ack vectors and vector clocks to learn about the progress of other replicas. The execution of an *Update* operation is blocked until it arrives at all replicas. XStamps [20] proposes a timestamp based XML data multi-replica control solution, but some conflicting operations will be aborted. Though this is avoidable in a serial execution, the authors point out that generally, aborting is inevitable in XStamps.

Our work focuses on consistency control of structured data such XML, and is uses the stack-based algorithm for XML query processing. Recent work including Holistic Twig Joins [3] further improves XML query performance. These new approaches rely on the same inverted list as the stack-based algorithm. Thus, they are compatible to our retracing approach.

8. CONCLUSIONS

In this paper, we proposed a lock-free approach for consistency maintance in Web 2.0 environment. Tranditional concurrent operations (e.g. two phase lock, serialization, etc.) lead to the throughput bottleneck among distributed mirror sites and a large portion of user operations is blocked. Our method has two significant features. First, we do not use the locking mechanism, so concurrent operations are executed as soon as possible upon their arrival. Our algorithm ensures the convergence of the state of the shared documents. This satisfies the need for most non-critical user operations in the Web 2.0 environment. Second, we support the transaction semantics for critical operations without using the locking mechanism. Instead, we rely on the lock-free causality preservation approach. We prove our approach implements consistency convergence and transaction semantics in our paper. Experimental results show that

our approach achieves better load balance, high-speed access and shorter respond time.

9. REFERENCES

[1] DBLP Computer Science Bibliography. http://www.informatik.uni-trier.de/ ley/db/, 2007.

[2] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[3] N. Bruno, N. Koudas, and D. Srivastava:. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD*, 2002.

[4] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic XML trees. In *PODS*, 2002.

[5] A. H. Davis, C. Sun, and J. Lu. Generalizing operational transformation to the standard general markup language. In *CSCW*, 2002.

[6] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD*, 1989.

[7] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in a database system. *Commun*, 19(11):624–633, 1976.

[8] N. Gu, J. Yang, and Q. Zhang. Consistency maintenance based on the mark & retrace technique in groupware systems. In *SIGGROUP*, 2005.

[9] C.-L. Ignat and M. C. Norrie. Customizable collaborative editor relying on treeOPT algorithm. In *ECSCW*, 2003.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun*, 7:558ÍC565, 1978.

[11] D. Li and R. Li. Transparent sharing and interoperation of heterogeneous single-user applications. In *CSCW*, 2002.

[12] D. Li and R. Li. Preserving operation effects relation in group editors. In *CSCW*, 2004.

[13] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, Mar 2005.

[14] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, Mar 1998.

[15] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD*, 2001.

[16] I. Traiger, J. Gray, C. Galthier, and B. Lindsay. Transactions and consistency in distributed database systems. *ACM Transactions on Database System*, 7(3):323–342, 1982.

[17] W3C. XML path language (XPath) 2.0. http://www.w3.org/TR/xpath20/, 2005.

[18] W3C. XQuery 1.0: an XML query language. http://www.w3.org/TR/xquery/, 2005.

[19] H. Wang and X. Meng. The sequencing of tree structures for XML indexing. In *ICDE*, 2005.

[20] K. Win, W. Ng, Q. Liu, and E. Lim. XStamps: A multiversion timestamps concurrency control protocol for XMLData. In *ICICS/PCM*, 2003.

APPENDIX

LEMMA 1. *The execution order of all transactional operations is consistent on all replica sites.*

PROOF. Suppose the execution order of $n-1$ transactions at all sites are consistent. We want to extend the result to n transaction operations.

Assume there are n transactions in the system: $T_1, T_2, \cdots, Tn$. Without loss of generality, assume the $T_1 \prec T_i$, for all $1 \leq i \leq n$. There may exist one or more $NOOP_a^{T_i}$ and we have $NOOP_a^{T_i} \rightarrow NOOP_a^{T_1}$ for all $2 \leq i \leq n$. Since $T_1 \prec T_i$, we switch the timestamps of involving NOOPs to reverse the causality. Specifically, $NOOP_b^{T_1}$ will switch its timestamp with all $NOOP_b^{T_i}$ until $NOOP_b^{T_1}$ causally precedes all $NOOP_b^{T_i}$ where $2 \leq i \leq n$.

Thus, T_1 must be executed first among $\{T_1, T_2, T_3, \cdots, Tn\}$ on all replica sites. Then the remain operations are $\{T_2, T_3, \cdots, Tn\}$. According to the assumption, the execution order of $n-1$ operations at every site is consistent. Thus, the execution order are consistent after n operations in all replica site. □

THEOREM 1. *Transactions in our system are serialized (Definition 5).*

PROOF. According to Lemma 1, the execution order of transactions is consistent on all replicas. To prove the current theorem, we only need to prove the order between each transaction and non-transaction is consistent on all replicas. Let T be a transaction and O be a non-transaction operation.

We consider the situation after all timestamp switches have taken place for transactional operations. Without loss of generality, assume O is generated from replica site a. Since the switch rule only applies to NOOP operations from the same replica site, after the switch, T should still have one corresponding NOOP operation from each replica site. Let $NOOP_a^T$ denote the NOOP operation from site a. Since both O and $NOOP_a^T$ are local operations of replica site a, their relationship can only be Causal Relationship (Definition 2). Their execution order is determined by the Execution Condition for remote operation (Definition 4). Since transaction T will be executed together with $NOOP_a^T$ at all sites. So the execution order between T and O is fixed.

In summary, no matter which type the operation is, the execution order is certain. This satisfies the Definition 5. □

LEMMA 2. *Assume the system contains two concurrent operations and two replicas. The results are the same even though the two operations are executed in different order on the two replicas.*

PROOF. Assume $U_1 \prec U_2$, and on one of the replicas, U_2 executes first. We what to show executing U_1 after U_2 leads to the same result. Consider any node n in the document. If n's timestamp is not changed by U_2, then U_2 has no effect on n, so executing U_1 before or after U_2 is the same. Suppose n's timestamp is changed to $(U_2, -)$, that is, n is created by U_2. Since $U_1 \prec U_2$, or $U_1 \notin (U_2, -)$, the retracing algorithm will make n invisible to U_1. Suppose n's timestamp is (U', U_2), i.e., n is deleted by U_2. Assume n's original timestamp is (U', U''), and we have $U' \prec U_2 \prec U''$. There are only two cases: (i) $U' \prec U_1 \prec U''$ and (ii) $U_1 \prec U'$, where in case (i) n is visible to U_1 and in case (ii) n is invisible to U_1, no matter there is U_2 or not. This means, with retracing, U_1 is seeing the document as if U_2 never occurred. □

THEOREM 2. *Assume the system contains n operations and any number of replicas. After the n operations have executed on all replicas, the results on all replicas are the same..*

Lemma 2 proved consistency for two operations in a distributed environment. Based on Lemma 2, we simply apply the results in our previous work (Theorem 3 of [8]) to extend the proof to n operations.