

Relaxed Currency Serializability for Middle-Tier Caching and Replication

Philip A.
Bernstein

Microsoft Corporation
philbe@microsoft.com

Alan
Fekete

University of Sydney
fekete@it.usyd.edu.au

Hongfei
Guo

Microsoft Corporation
hongfeig@microsoft.com

Raghu
Ramakrishnan

University of Wisconsin-Madison
{raghu,pradeep}@cs.wisc.edu

Pradeep
Tamma

ABSTRACT

Many applications, such as e-commerce, routinely use copies of data that are not in sync with the database due to heuristic caching strategies used to enhance performance. We study concurrency control for a transactional model that allows update transactions to read out-of-date copies. Each read operation carries a “freshness constraint” that specifies how fresh a copy must be in order to be read. We offer a definition of correctness for this model and present algorithms to ensure several of the most interesting freshness constraints. We outline a serializability-theoretic correctness proof and present the results of a detailed performance study.

1 INTRODUCTION

The use of copies is widespread in e-commerce sites. For example, consider an auction site; when browsing auctions in a category, it is apparent that the data (e.g., item prices, number of bids) is a little out of date. However, most users understand and accept this, as long as they see completely current data when they click on an individual item. As another example, a purchase transaction may calculate and report to the user an estimated arrival date for the goods, based on reading a log of shipment timings at the cache. Reading slightly stale information might be acceptable given the business requirements, but it is essential to record this promised arrival date at the centralized backend server.

As a concrete example, the following query returns a summary of books that have the specified title:

```
SELECT I.ID, I.COST, I.SUBJECT, I.ISBN,  
       I.INSTOCK, A.FNAME, A.LNAME  
FROM   ITEM I, AUTHOR A  
WHERE  I.AID = A.AID AND I.TITLE = "SQL"
```

Different users might have different freshness requirements for this query. For example, user U_1 is about to purchase all of these books and calls transaction T_1 to get an up-to-date query result. User U_2 is browsing and runs transaction T_2 , which offers quick response time, even at the cost of allowing the `I.INSTOCK` column to be out-of-sync. A third user, U_3 , is studying the relationship between the cost of a book and the number of copies in stock by periodically running the query T_3 and recording the results. In this case, it is acceptable for the result to be stale as long as it reflects a consistent database snapshot, i.e., a database state that is consistent at a certain point of time. In fact, a weaker

version of this guarantee might suffice, requiring only that all rows retrieved for a given item reflect the same snapshot, with different items possibly coming from different snapshots.

This example illustrates the main usage scenarios that we want to tackle. They typically arise because of application or middle-tier caching layered on a database system. However, it is problematic to allow update transactions to read the out-of-date values. The standard correctness criterion for transactions is one-copy serializability, which says that an execution should be equivalent to a serial execution on a one-copy database. The transactions for users U_2 and U_3 in the above example violate this criterion. For example, suppose that a transaction, T_4 , places an order for an item (thus changing `I.INSTOCK` for that item), and another transaction, T_5 , then updates the cost of that item. If transaction T_2 issued by user U_2 now reads an older, cached value of `INSTOCK`, along with the current value of `COST`, it sees a database state that can never arise in a serial execution on a one-copy database.

We investigate concurrency control for a transactional model that allows an update transaction to read out-of-date copies. Each read operation carries a “freshness constraint” [13] that bounds how up-to-date a copy must be in order to be read. Up-to-date-ness may be relative to read-time, reader’s commit-time, or freshness of copies of other items that are read. Our main contributions are:

- An extended serializability model called *Relaxed Currency (or RC-) Serializability* that accounts for out-of-date reads that are justified by user-specified freshness constraints.
- Concurrency control protocols that allow update transactions to read out-of-date data safely, that is, read data that satisfies the associated freshness constraints. Our protocols accumulate constraints on the updater’s time, which can be checked at read-time or as part of commit processing.
- A proof sketch that the protocols produce RC-serializable executions.
- A detailed simulation study of the protocols. It shows that performance is similar to the weakest protocol used in practice, read-committed, which offers no freshness guarantee.

The paper is organized as follows. In Section 2 we introduce our model of correctness for interleaved executions of transactions that read cached and possibly out-of-date copies. We define our models of a replicated database and transactions that can read out-of-date copies in Section 3, and several classes of freshness constraints in Section 4. We present several concurrency control algorithms to ensure RC-serializability in Section 5 and prove their correctness in Section 6. We present a performance study in Section 7 and discuss implementation extensions in Section 8. Section 9 covers related work. Section 10 is the conclusion.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD 2006, June 27-29, 2006, Chicago, Illinois, USA.
Copyright 2006 ACM 1-59593-256-9/06/0006...\$5.00.

2 RC SERIALIZABILITY

We define a new user model of computation where transactions may update data yet read cached copies, subject to various freshness constraints. We also define implementations that realize it. To relate the user model and implementation, we use serializability theory [6]. However, all forms of correctness stated in that theory require each transaction to read values as in a serial one-copy execution, which does not hold if an update transaction reads slightly stale replicas. We need a weaker correctness property that captures our intuition of what a transaction does when it reads stale values.

As in traditional serializability theory, we define the correctness of an implemented *physical system* by saying that its behavior must be indistinguishable from that of a simpler, *ideal system* (i.e., the user’s model) where transactions execute serially. In the ideal system, we model a **database** as a collection of **data items**. Since the freshness constraints are part of the user’s model, the user is aware that transactions read out-of-date values of data items. Such out-of-date values are called **versions** of a data item. Thus, in the ideal system a data item is a sequence of versions and a computation is a multiversion execution. That is, the user must understand that (i) each time a transaction updates a data item a new version of that data item is (conceptually) created, and (ii) when a transaction reads a data item, it may read an old version of that item, not the most recent one. Each read operation’s freshness constraint tells whether a version is satisfactory.

We therefore define the **ideal system** to be a one-copy multiversion database. By **one-copy**, we mean there is a multiversioned master copy of each data item and no replicas (i.e., cached copies). We define a **correct execution of the ideal system** to be a serial multiversion execution in which each read operation satisfies its freshness constraint. A **correct execution of a physical system** is one that is “equivalent” to a correct execution of the ideal system. A correct execution of a physical system is called **relaxed-currency (RC) serializable**.

To our knowledge, this is the first time a multiversion database has been taken to be the user’s model of computation. By contrast, the well-known correctness definition for multiversion and replicated databases is one-copy serializability, which uses a one-copy single-version database as the ideal system.

3 THE PHYSICAL SYSTEM MODEL

One of the main questions we consider in this paper is how to extend the theory of transactions and serializability to account for freshness requirements. To do this, we need to model the physical system consisting of transactions, data managers, and databases. Since cached copies are replicas of a master copy, we can model the system as an ordinary non-versioned replicated database. This section defines our **replicated database model** (see Figure 1).

We assume that each data item has a unique identity. A **transaction** is modeled as a set of partially-ordered actions, which include **reads** and **writes** of data items, and, possibly, a final **commit** or **abort**. For simplicity, we do not allow inserts or deletes of items, since they would lead to many notational complexities. We consider them and the associated issue of phantoms as an extension in Section 8. We denote the set of items read by transaction T as $readset(T)$ and the set written by T as $writeset(T)$.

Every data item has a set of physical **copies**: one **master** and zero or more **replicas**. The collection consisting of masters of all of the data items is called the **master database**. A **cache** is a collection of replicas of one or more data items. There is only one version of each physical copy, which contains its latest value.

We assume that the master database has a **master manager** that controls the execution of reads and writes to the master. Similarly, each cache has a **cache manager** that controls the execution of reads and writes to the cache.

We distinguish between **user-transactions** (or simply **transactions**) that are issued by users, and **copy-transactions** (or simply **copiers**) that are issued by the master manager. Each transaction issues each read operation either to the master manager or to a particular cache manager. For now, we assume that it only modifies the master database, so it sends all of its write operations to the master manager, which is where they are processed.

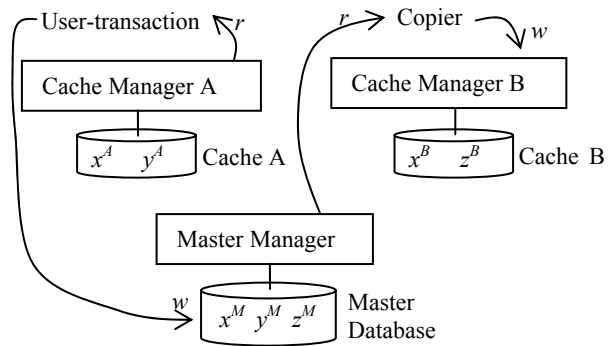


Figure 1: Replicated Database Model

After a transaction T commits, the master manager invokes a copier for each pair $[x, P]$ where x is a data item updated by T and P is a cache that has a replica of x . A copier reads the value of the master x^M and writes that value into the corresponding replica x^P in cache P . Note that a copier runs outside of the transaction that caused the copier to be invoked. For now, we leave open whether the copiers from the master to a given cache are pipelined in commit order or grouped (e.g., one copier contains reads and corresponding writes for multiple $[x, P]$ pairs).

The techniques in this paper are applicable when the master database includes materialized views of base data and when replicas of those materialized views are stored in caches. In this case, we assume that each transaction maintains the consistency of materialized views with respect to base data. Whether this is done by user code or system code executing within the transaction is unimportant to our model. What is important is that we assume that the master and cache managers treat updates to materialized views the same as those on base data. Solutions that relax this assumption may be useful, but are outside the scope of this paper.

4 FRESHNESS CONSTRAINTS

To capture the concept of freshness introduced in Section 1, we require that each read request from a transaction includes a **freshness constraint** [13]. The cache manager or master manager that services a read must return a value of the requested item that satisfies the read’s freshness constraint. Freshness constraints are defined based on the time-drift or value-drift of an item from its correct value, or on the mutual consistency of item values. To

define these precisely, we use the well-known concepts of timestamps and snapshots.

We assume a **timestamp domain**, which is a subset of the natural numbers. The master manager assigns to every committed transaction T a unique **transaction timestamp** $ts(T)$ greater than all timestamps assigned to previously committed transactions.

We assume that the master manager and cache managers can calculate a user-comprehensible wall-clock time from each timestamp and vice versa. Thus, the system can translate a freshness constraint expressed using wall-clock time into one expressed using timestamps. In practice, real-time clocks are fine-grained and accurate enough to enable this. Since timestamps are generated only by the master manager, clock skew can be ignored. Hereafter we do not distinguish timestamps from wall-clock time.

Every copy x^A (i.e., a master or replica) has an associated **last-modified** timestamp, $last-modified(x^A)$, which is the timestamp of the last committed transaction that wrote x^A . The value of a copy written by a committed transaction is called a **committed version**.

Each committed version x_i of a copy x^A of data item x is said to be **valid** over the half-open interval $[t, t')$ where $t = last-modified(x_i)$ and t' is either the next larger last-modified timestamp of another committed version of x^A or ∞ if no version of x^A has a larger last-modified timestamp. We call $[t, t')$ the **valid interval** of x_i .

A **snapshot** is a database state produced by a certain set of transactions—usually, the committed transactions in the prefix of an execution. A snapshot is associated with a timestamp t and maps each data item x to the version of x whose valid interval includes t .

The ideal system, which is the user’s model, is just like the physical replicated database model except there is just one copy of each data item (and hence no copiers), and all of its committed versions (from all past writes) are visible to user-transactions. Thus, the semantics of freshness constraints can refer to versions of data items (since there is just one copy of each). We classify freshness constraints into three classes B , D , and M as follows.

B. Time-Bound and Value-Bound Constraints

B1. Time-Bound Constraints. We can limit the **currency bound**, which is the difference between the time a version becomes invalid and the time it is read (the **read-time**). For example, we can specify that it is acceptable to read a version of x that is up to 10 seconds out of date: $bound(x:10)$. If $bound(x:0)$, then the read must be given a valid version of x . For simplicity, we omit the time unit and assume it is seconds.

B2. Value-Bound Constraints. We can specify that the value read for x is within a certain percent of the value of the valid version: $bound(x:10\%)$.

D. “Drift” Constraints on multiple data items. When a transaction T reads more than one data item, we can specify constraints over a subset S of $readset(T)$.

D1. Snapshot consistency. We can require that data items in S must be read from the same snapshot, denoted $snapshot(S)$.

D2. Limited Time-Drift. The constraint $drift(S, b)$ says that for every two items x and y in S , the versions of x and y that are read are within b seconds of each other. That is, if the transaction reads version x_i of x and y_j of y , then there are

timestamps t_x and t_y such that x_i ’s valid interval includes t_x , y_j ’s valid interval includes t_y , and $|t_x - t_y| \leq b$. Observe that snapshot consistency is the special case of $b=0$.

D3. Limited Aggregate Value Drift: We can require that an aggregate computed over (a subset of) the readset of T be within a certain percentage of a “recent” value (even though individual data items may be further out of sync). We use the notation $drift(S, AGG, b\%, w)$, where AGG is an aggregate operation, and w is a time window. It means that $AGG(S)$ must be within $b\%$ of $AGG(S')$, where S' denotes the value(s) of committed versions of items in S at some instant less than w seconds prior to the current time.

Constraints of type B and D can be combined. For example, a transaction can set $bound(x:10)$ and $snapshot(\{x, y\})$ [13]. We can further classify these constraints using two orthogonal criteria:

- **Granularity:** Constraints over individual data items, sets of items, aggregates over sets of items, subsets of snapshots, and complete snapshots.
- **Unit of skew:** Specify the skew using timestamps, number of subsequently committed transactions, or value.

M. Multi-statement Constraints

We can specify “session level” constraints that refer to points in time external to the execution of the current statement. For example, a transaction’s reads must see changes made by preceding statements in this transaction or in this session.

When evaluating constraint types $B1$, $D2$, and M , there are two possible notions of read-time: the time when the read executes, or when its transaction (later) commits. Even in the user’s model of computation (a serial execution of the ideal system), these read-times are not the same unless the transaction has negligible execution time. For most of this paper, we use a transaction’s commit time to be the read-time of all of the transaction’s reads. We consider the other semantics in Section 8.

5 ENFORCING RC SERIALIZABILITY

Enforcing RC-serializability requires ensuring that transactions are serializable and satisfy their freshness constraints. Enforcing RC-serializability is non-trivial because these requirements are not independent: freshness constraints are affected by factors related to transaction order, such as transaction timestamps. So the different transaction orders in two equivalent serial executions might affect whether certain freshness constraints are satisfied. The goal of RC-serializability is to ensure that each execution has at least one equivalent serial execution in which freshness constraints are satisfied.

The problem of proving RC-serializability is also non-trivial. As in one-copy serializability, the notion of equivalence in the definition of RC-serializability requires mapping between two different kinds of executions—multiversion executions of the ideal system and replicated data executions of the physical system—and proving that they have the same behavior. We will do this in Section 6. But first we introduce some algorithms that produce RC-serializable results for various kinds of freshness constraints. We describe how to deal with transactions that include both reads and writes; read-only transactions (queries) can use the techniques of [13].

5.1 Baseline Synchronization

We first present a general concurrency control algorithm, called the **baseline synchronization algorithm** (or **BSA**). It provides a common framework for the algorithms we discuss later that enforce freshness constraints, but does not by itself enforce such constraints. In summary, user-transactions read replicas using read-duration locks and write masters using commit-duration locks. Thus, user-transactions run at the **read committed** isolation level. Commit and abort are processed at the master site, which generates transaction timestamps for committed transactions. For copiers, the master pipelines writes to caches in timestamp order.

We describe BSA in terms of how it handles the following four actions: reads, writes and commits of user-transactions, and copiers. See the boxed algorithm descriptions below and their descriptions that immediately follow.

BSA-Read(x) // issued by a user-transaction
 Choose a manager R from which to read x ;
 // R can be the master for x or a cache
 read-lock x at R ;
 read x^R ;
 release read-lock;

BSA-Write(x) // issued by a user-transaction
 write-lock x^M at the master for x ; // lock held until commit
 write x^M ;

BSA-Commit(T) // issued by a user-transaction
 // and processed at the master site
 $ts(T) \leftarrow$ new transaction timestamp;
 // $ts(T) >$ all timestamps previously issued
 for all items x in $writeset(T)$
 $last-modified(x^M) \leftarrow ts(T)$;
 release all locks;

BSA-Copy(x, P) // a copier transaction
 read-lock x^M at the master for x ;
 read x^M and $last-modified(x^M)$;
 release read-lock;
 write-lock x^P at cache manager P ;
 $x^P \leftarrow x^M$;
 $last-modified(x^P) \leftarrow last-modified(x^M)$;
 // assumes writes are applied in last-modified order
 release write-lock

Writes set exclusive locks on master copies. The locks are released only after the transaction commits and its updates are applied at the master. Thus, for each unlocked master item x^M , $last-modified(x^M)$ is the timestamp of the last committed transaction that updated x .

Reads set short-duration shared locks. This ensures that each read operation sees a committed value. It also ensures that the data and $last-modified$ value are mutually consistent at the time the read occurs, since there may be a gap between a transaction updating a copy's data value and updating its $last-modified$ timestamp.

Copiers are generated by the system. They set short-duration shared locks to read the master copy of an item before propagating the value to a target cache. Updates from copiers are pipelined to each cache in timestamp order, which is the same as

commit order. This is how primary-copy replication works in most of today's commercial database products.

Since updates are propagated in timestamp order to each cache, successive reads of the same data item in a cache see time moving forward. But if a transaction reads the same item from different caches, this may not hold. To avoid this, assume all updates of each transaction are applied atomically at each cache, and define $last-modified(A)$ for cache A to be the largest value of $last-modified(x^A)$ of any copy x^A at A . Each transaction T_i remembers the maximum value m_i of $last-modified(A)$ over all caches it read from and attaches it to all reads. Before processing a read of y^B from T_i , cache B checks $last-modified(B) \geq m_i$. If not, it can wait till more updates arrive so the check is satisfied, or it can reject the read. The rest of this paper is independent of whether this is done.

At this point, we make no assumptions about how and when each copier is generated. It might be generated on behalf of a particular transaction T to copy $writeset(T)$, or periodically for each item or set of items, or using some other strategy.

5.2 Algorithms for Ensuring Freshness

5.2.1 Algorithm BSA-FC

We now propose tests to determine whether a given set of copies satisfies reads with various freshness constraints. The freshness tests we propose are implemented as additional steps in BSA-Read, BSA-Write and BSA-Commit. In BSA, every copy x^A has an associated timestamp $last-modified(x^A)$. Our tests use another timestamp associated with each copy, called $valid-till(x^A)$. Intuitively, $valid-till(x^A)$ is the smallest timestamp that could be associated with the next version of x^A . Regardless of how it is maintained, we require the following property to be preserved:

Property VT The valid interval of the value held in a copy x^A must include the closed interval $[last-modified(x^A), valid-till(x^A)]$.

Recall that the valid interval of version x_i of x is the half-open interval $[t, t')$ where $t = last-modified(x_i)$ and t' is either the next larger last-modified timestamp associated with another committed version of x or ∞ if no other version of x has a larger last-modified timestamp. For the master copy x^M , we can take $valid-till(x^M)$ to be the largest timestamp issued so far, say t'' . This works because the next larger timestamp t' of a version of x will surely be larger than t'' , so Property VT holds. For a replica x^A , we can take $valid-till(x^A)$ to be $last-modified(x^A)$, for which Property VT trivially holds. However, since updates are propagated to each cache in timestamp order, and assuming updates of each transaction are applied atomically at each cache, $valid-till(x^A)$ can be taken as $last-modified(A)$. In this case, $valid-till$ has the same value for all items in a cache and can be maintained at cache granularity.

When a read or group of reads is performed, the system uses the read's freshness condition, values of $last-modified$ and $valid-till$, and perhaps other information to deduce a constraint on the timestamp of the reader's transaction. For each transaction T , these timestamp constraints are remembered. When T is ready to commit, the master site assigns T 's timestamp and then checks if it satisfies all of T 's timestamp constraints. If any constraint is false, then T aborts. Alternatively, it could be backed out to a savepoint preceding the read for which the system deduced the failed constraint, or if aborted it could be restarted. But these stra-

gies are independent of the approach to RC-serializability, so to simplify the algorithm descriptions we ignore them in what follows.

The rest of this section describes variations of BSA that handle Time-Bound and Limited-Time-Drift freshness constraints (classes B1 and D2 in Section 4). Section 5.2.2 shows how to reduce commit-time work at the master. Section 5.2.3 briefly discusses other classes of constraints.

To test freshness conditions associated with read operations of transaction T_i the following checks are added to *BSA-Read*:

- a) Time Bound $bound(x:b)$: Add the constraint “ $ts(T_i) \leq vt+b$ ” to transaction T_i ’s constraint Con_i , where vt is the value of $valid-till(x^A)$ associated with the value being read. The simplest implementation uses $valid-till(x^A)$ at the moment the read is performed; we discuss alternative implementations below.
- b) Limited Time-Drift Bound $drift(S, b)$: Recall that S denotes a subset of T_i ’s readset. Let $max(last-modified(S))$ denote the largest *last-modified* timestamp associated with any of the copies in S that are read by T . Similarly, let $min(valid-till(S))$ denote the smallest *valid-till* timestamp of a copy in S that is read by T . Add the constraint “ $max(last-modified(S)) < min(valid-till(S))+b$ ” to the transaction’s constraint Con_i .

And the following change is made to *BSA-Commit* for T_i :

- If possible, assign a timestamp to T_i that is greater than all previously issued timestamps and also satisfies all of its constraints in Con_i . Otherwise, abort T_i .

We will refer to BSA with the above changes as **BSA-FC** (for “freshness checking”).

To understand Step (a), recall that $bound(x: b)$ says that the value read by T_i can be at most b seconds out-of-date. Since the value of x^A that was read is valid until at least vt , if $ts(T_i) \leq vt+b$, then $bound(x: b)$ is satisfied.

To enforce $drift(S, b)$, Step (b) must ensure that there is a timestamp within b seconds of the valid interval of every version read by T_i . To see how it accomplishes this, consider any two data items $x \in S$ and $y \in S$. Let the valid intervals of the versions read by T_i be $[t_x', t_x'']$ and $[t_y', t_y'']$. Without loss of generality, suppose $t_x' \leq t_y'$. As shown in Figure 2, either (i) $t_y' \leq t_x''$ and the intervals overlap or (ii) $t_x'' < t_y'$ and the intervals are disjoint. In case (i) $drift(\{x,y\}, b)$ is satisfied. In case (ii) we need $t_y' - t_x'' \leq b$, or $t_y' \leq t_x'' + b$, to satisfy $drift(\{x,y\}, b)$. But this follows immediately from the check in Step (b), since $t_y' \leq max(last-modified(S)) \leq min(valid-till(S))+b$, and $min(valid-till(S)) \leq t_x''$. Since we chose x and y arbitrarily, $drift(S, b)$ holds for all x and y in S .

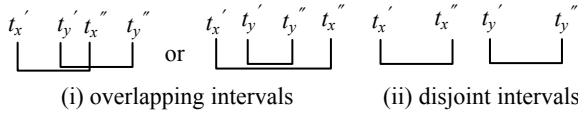


Figure 2: Valid Intervals for Checking $drift(S, b)$

The tests described above cover freshness constraints of types B1 and D2. Type D1 is a special case of D2 where $(b = 0)$ and is therefore covered as well.

5.2.2 Variants of BSA-FC

This section describes several optimizations or variant implementations of BSA-FC. One optimization is that when a constraint C is generated, if the system knows that C cannot be satisfied, then it can reject the operation immediately. For example, suppose the check for $bound(x:b)$ takes vt as the value of $valid-till(x^A)$ at the time the read occurs. The cache manager might know that the most recently committed transaction’s timestamp is already greater than $vt+b$, because it saw an update with timestamp greater than $vt+b$. Since the constraint will not be satisfied when the commit is eventually attempted, the read should be rejected. However, by using more expensive implementations, one can get larger correct values of vt , which reduce the number of unnecessary aborts in four scenarios where vt may increase over time:

- If the master is a multiversion database, then it knows the exact value of *valid-till* of every version of every data item, which is at least vt . So the constraint $vt+b$ can be replaced by $[x^A, lm, b]$ where lm is *last-modified*(x^A) of the value read. Using this, the master can calculate $vt+b$ at commit time, where vt is *valid-till* of the version of x^A identified by lm .
- If the transaction includes *last-modified*(x^A) with its update, and if the master finds *last-modified*(x^M) to be the same, then it can use its local value $valid-till(x^M)$ as vt when checking the constraint.
- The cache could be made to participate in two-phase commit (which it may do anyway if the transaction updated the cache). The phase-one processing can re-read *last-modified*(x^A) and *valid-till*(x^A). If *last-modified*(x^A) is unchanged from the time the transaction read x^A , but *valid-till*(x^A) has increased (which is likely) then the new value of *valid-till*(x^A) can be returned in the phase-one reply and used by the master in the constraint check.
- The cache could remember the identity of every replica read by every active transaction. When a replica x^A is updated by a copier, the value of *valid-till*(x^A) is recorded and frozen for all active transactions that read it. This is more accurate than the previous method, but adds bookkeeping cost. Again, phase-one processing obtains the latest values of *valid-till*.

If a time-bound constraint $bound(x, b)$ is applied to each x read by transaction T , then the processing of Step (a) can be optimized for the common case when a transaction’s read operations are issued in sequence. The system maintains a counter *high* on behalf of T_i , initialized to ∞ and updated to $min(high, valid-till(x^A)+b)$ after each read, where x^A is the copy that was read. At commit time, it ensures that $ts(T_i) < high$, a single constraint that implies $ts(T_i) \leq valid-till(x^A)+b$ for every data item x^A that it accessed.

A similar optimization applies to $drift(S, b)$. The system maintains two counters on behalf of T_i , *high* (as above) and *low*. It initializes *low* to $-\infty$. For each read of (say) x^A by T_i , if $max(low, last-modified(x^A)) > min(high, valid-till(x^A)+b)$, then it rejects the read. Otherwise, it performs the read, sets $low = max(low, last-modified(x^A))$ and sets *high* as above. At commit time, it knows that $low < high$, so $drift(S, b)$ is surely satisfied.

Insofar as *last-modified* is used for drift constraints, these timestamps can be maintained at coarse granularity, reducing the space required in the cache for storing these values. The downside

is that in place of the accurate *last-modified*(x^A) we use the largest *last-modified*(y^A) across all y^A in x^A 's granule. Thus, $\max(\text{low}, \text{last-modified}(x^A))$ may be larger than it needs to be, increasing the chance that $\max(\text{low}, \text{last-modified}(x^A)) > \min(\text{high}, \text{valid-till}(x^A) + b)$, causing the read of x^A to be rejected. That is, it may increase the number of aborts, but does not lead to erroneous results.

5.2.3 Handling Other Classes of Constraints

Value-Bound conditions can be processed if there is a known bound on the rate at which the value of an item can change. For example, the location of a vehicle will not change faster than the vehicle's speed allows. Given such a rate bound, we can deduce a bound on timestamp drift from one on value drift and enforce the deduced time-drift bound. Limited aggregate-drift bounds can be handled similarly, given bounds on the rate of change of values.

Multi-statement constraints were discussed in Section 4. Such a constraint might say that a transaction sees all updates made by previous transactions in the same session. To implement this, the system can remember the maximum timestamp *session-max-ts* of any committed transaction in the session. When it executes a read of x^A , it checks that $\text{valid-till}(x^A) \geq \text{session-max-ts}$.

A more challenging multi-statement constraint is that a transaction T_i 's reads must see the results of its preceding writes. To enforce this, we can maintain a list L_i of all writes executed by T_i . Every read from T_i is checked against L_i . If T_i previously wrote the item to be read, then the read must be executed at the master so that it sees the prior write. We discuss this further in Section 8.

6 CORRECTNESS

Intuitively, BSA-FC is correct because it serializes transactions in timestamp order based on write-write conflicts and checks that each read satisfies its freshness constraints relative to that order. To prove this, we need to formally define the physical and ideal systems and prove that every execution of the former is equivalent to a correct execution of the latter. Due to space limitations, we only sketch the proof here. A complete proof is in [7]. In this section we assume a basic knowledge of serializability theory.

The physical system is modeled by a **physical history**, which is a replicated data history. The ideal system is modeled by an **ideal history**, which is a multiversion history. To define equivalence, we extend the usual reads-from relation to include the semantics of copiers: T_i **takes-x-from** T_k if either T_i reads-x-from T_k , or there is a copier C_j such that T_i reads-x-from C_j and C_j reads-x-from T_k . A physical history is **equivalent** to an ideal history if they have the same takes-from relation and final writes.

We first prove that a history H is **bare-RCSR**, meaning that H is equivalent to some serial ideal history H' . To do this, we define an RC serialization graph (RCSG) over H , whose nodes are transactions and whose edges capture takes-from relationships and write-write conflicts. We then characterize BSA's physical histories and prove every such history has an acyclic RCSG.

Theorem 1 For any primary-copy physical history H , if $\text{RCSG}(H)$ is acyclic then H is bare-RCSR. \square

Proposition 1: A physical history H produced by BSA has the following properties, where $<_H$ denotes the order of operations in H :

A. Every committed transaction T_i has a unique timestamp $ts(T_i)$

B. $ts(T_j) < ts(T_k)$ iff $c_j <_H c_k$

C. If T_i reads-x-from T_k then $ts(T_k) < ts(T_i)$

D. If $r_i[x^M], w_i[x^A] \in H$ for copier C_i , master x^M , and some copy x^A , then $r_i[x^M] <_H w_i[x^A]$

E. If T_i takes-x-from T_k then $ts(T_k) < ts(T_i)$

F. If $w_k[x^M] < w_i[x^M]$, then $ts(T_k) < ts(T_i)$ \square

Theorem 2 For every physical history H produced by BSA, $\text{RCSG}(H)$ is acyclic. \square

By Theorems 1 and 2, every physical history H produced by BSA is bare RCSR. In fact, from Prop. 1B, 1E and 1F, H is **strictly** bare RCSR, meaning there is an equivalent serial ideal history H' such that commits in H appear in the same order as in H' .

Corollary 1 Every physical history H produced by BSA is strictly bare RCSR. \square

BSA-FC extends BSA with commit-time checks of time-bound and limited time-drift bound constraints. To prove these constraints are satisfied, we need to refer to the timestamps of transactions in the ideal history. We say that a *timestamp assignment for ideal history H'* is valid if, for all T_j, T_k , $ts(T_j) < ts(T_k)$ iff $c_j <_{H'} c_k$. The following proposition follows from Corollary 1.

Proposition 2: If a history H is strictly bare-RCSR, then a valid timestamp assignment for H is also a valid timestamp assignment for an ideal history equivalent to H . \square

Theorem 3: BSA-FC is RCSR.

Proof: By Prop. 2, every physical history H produced by BSA-FC is equivalent to an ideal history H' with the same timestamp assignment. We need to show freshness constraints hold in H' .

First, consider a time-bound constraint $\text{bound}(x, b)$ on read operation $r_i[x^A]$. Let x_1 be the version of x^A read by $r_i[x^A]$. If x_1 is the last version of x^A , then x_1 never becomes invalid so $\text{bound}(x, b)$ holds. Else let x_2 be the next version of x^A — the one whose transaction has next larger timestamp among those that updated x^A . By definition, $\text{bound}(x, b)$ holds if $ts(T_i) < ts(x_2) + b$. By Property VT, $vt \leq ts(x_2)$. BSA-FC ensures the constraint $ts(T_i) \leq vt + b$ is satisfied when $ts(T_i)$ is allocated, thus $ts(T_i) < ts(x_2) + b$ as desired.

Next, consider a time-drift constraint $\text{drift}(S, b)$ and two reads $r_i[x^A]$ and $r_i[y^B]$, where $x \in S$ and $y \in S$. To satisfy $\text{drift}(S, b)$, if $r_i[x^A]$ and $r_i[y^B]$ read versions x_i of x and y_j of y respectively, then there must be timestamps t_x and t_y such that x_i 's valid interval includes t_x , y_j 's valid interval includes t_y , and $|t_x - t_y| \leq b$. This was proved at the end of Section 5.2.1, so we are done. \square

7 PERFORMANCE

This section evaluates the performance of the BSA-FC protocol with time-bound constraints. It shows the potential gains when stale data is acceptable, and the cost of guaranteeing currency bounds. With currency bound infinity, BSA-FC behaves like BSA, allowing the most concurrency. At the other extreme, with currency bound 0, BSA-FC guarantees one-copy serializability. We address the following issues:

- How does BSA-FC perform with increased data contention and resource contention in the system? (Section 7.3)

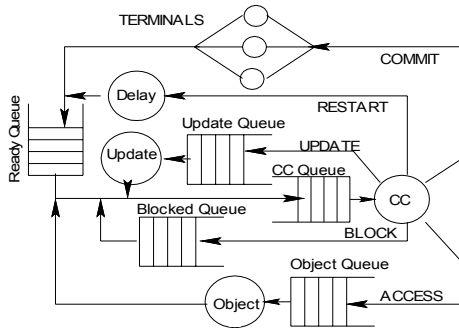


Figure 3: Single-Site Logical Queuing Model

- What is gained by executing reads of update transactions, in addition to read-only transactions, at caches? How does refresh interval impact throughput? (Section 7.4)
- How does performance scale as we add resources and caches? (Section 7.5)
- How does BSA-FC perform with different cache configurations and compare with a no-cache setting? (Section 7.6)

Our results show that superior performance and good scalability can be achieved with relatively tight currency bounds. We begin by describing our performance model in Section 7.1 and the workloads and metrics we consider in Section 7.2.

7.1 Performance Model

Our performance model covers two scenarios, single-site and master-cache. We implemented it in about 8000 lines of code using CSIM [9], a process-oriented discrete event simulation package. The performance model captures the relevant characteristics of the database system, users, and transactions. The database system model includes the physical resources (CPUs and disks), their associated schedulers, the mechanism for controlling the number of active transactions, and the concurrency control algorithm. The user model captures the arrival rate of transactions. The transaction model captures the reference behavior and processing requirements in the workload.

Our single-site model is the closed queuing model of Agrawal, Carey, & Livny [2] and is outlined in Section 7.1.1. (See [2] for details.) We extend it to the master-cache setting in Section 7.1.2.

7.1.1 Single-Site Setting

As shown in Figure 3, which is reproduced from [2], transactions originate from a fixed number of terminals. The number of active transactions in the system is bounded by the **multiprogramming level (MPL)**. A transaction is active if it is either receiving or waiting for service. When a new transaction arrives, if the system is full, it enters the ready queue and waits for an active transaction to commit or abort. It then enters the CC queue and requests its first lock. Upon success, it enters the object queue, accesses the object, and re-enters the CC queue for its next lock. If the lock request is blocked, the transaction enters the blocked queue and waits. If the lock request causes a deadlock, the transaction aborts and enters the ready queue after a random restart delay. Once all the accesses are done, the concurrency control algorithm may choose to commit the transaction. If the transaction is read-only, it is finished. Otherwise, it enters the update queue and writes its

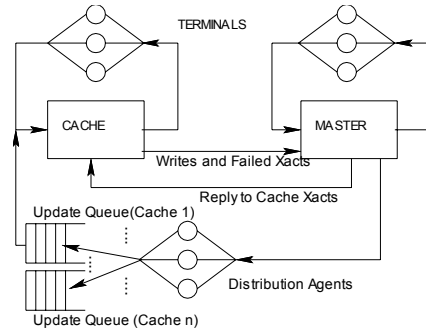


Figure 4: Master-Cache Logical Queuing Model

deferred updates into the database. For BSA-FC, the system notes the timestamps needed for checking transaction timestamp constraints, as in Section 5.2. In all experiments BSA-FC uses time bound constraints with the same currency bound for all reads.

Underlying this logical model is a physical model consisting of CPU and IO Resources, where a transaction accesses CPU and IO resources a certain number of times. For details, see [2].

7.1.2 Master-Cache Setting

This setting comprises a single master and multiple caches. We make the simplifying assumption that the master and all caches have the same set of objects. The logical model is shown in Figure 4. Each terminal is associated with a specific site and submits user transactions to that site. Each site processes transactions as described in the single-site setting, with the changes described below.

Each cache executes two types of transactions: user transactions and copiers. Further, the cache distinguishes read-only from read-write user transactions. As in [13], read-only transactions that execute entirely at a cache are validated there, instead of at the master, and are sent to the master only if the validation fails. For a read-write user transaction, the cache executes the read part and notes the timestamps needed for transaction timestamp constraints, as in Section 5.2. At commit time, it sends the writes and timestamp constraints to the master, where either the transaction receives a timestamp and commits or it aborts. The transaction then returns to the cache, which returns the result to the terminal.

The master executes three types of transactions: user transactions submitted from local terminals, read-only transactions submitted from caches (due to failed freshness constraint checks), and (possibly parts of) read-write transactions submitted from caches. All three types of transactions are handled as in the single-site case.

Copiers are implemented using the standard log-sniffing approach. The master has a distribution agent per cache. It wakes up after every refresh-interval and sends the tail of the log to the cache. When a cache receives the log, for each transaction in the log it runs a copier to refresh its local database. Each copier updates the valid-till timestamp of the cache and the last-modified time of each object written by the logged transaction. As for the single-site setting, the master-cache logical model has a corresponding physical model (similar to the one for single-site).

Increasing the number of resources at the master increases the parallelism, leading to more read-write transactions being committed. After transactions are committed, their writes should be replayed at each cache in commit order. This sequential execution leads to poor performance of copiers at caches. We address this

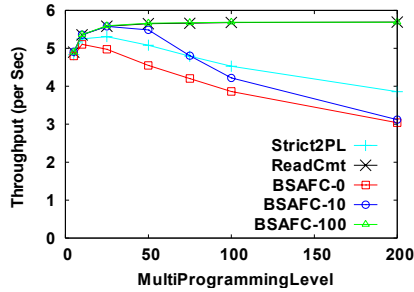


Figure 5: 1 Resource Unit¹

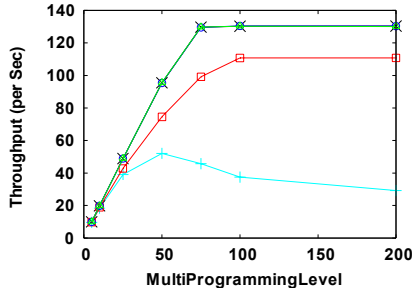


Figure 6: Infinite Resource Units

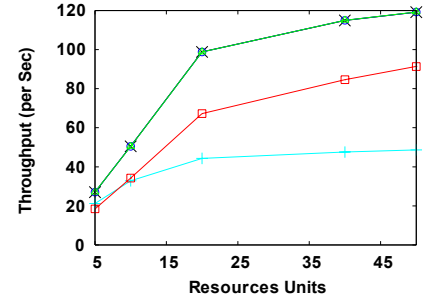


Figure 7: Multiple Resource Units

issue with two optimizations. First we increase the priority of copiers executing at a cache. This reduces the disk waiting time for copiers. Second, we execute a batch of copiers in parallel. That is, we consolidate multiple copiers into one multi-threaded transaction; locks are released together at commit. If an object is written by multiple copiers, we ensure the value and timestamp of the last copier in the batch is reflected. We set the number of consolidated copiers to twice the number of disks in the system.

7.2 Workload Parameters and Metrics

Major parameter settings, shown in Table 1, were adopted from [2]. Although they do not reflect current technology, the ratios are still sensible. Using them makes our measurements comparable to those in [2] for validation, and avoids long simulation times without affecting the relative performance of the methods. For example, substituting speeds of today's disks, throughputs increase by a factor of 10 but the shapes of the curves remain the same. We added more parameters to capture system characteristics specific to our context, such as currency bounds and distribution costs.

Table 1: Parameter Settings

Parameter	Value	Parameter	Value
db_size	1000 Pages	restart_delay	adaptive
tran_size	8-page readset	num_terminals	200
max_size	12-page readset	ext_think_time	1 second
min_size	4-page readset	obj_io	35 ms
write_prob	0.25	obj_cpu	15 ms

We briefly describe the main parameters. The number of objects in the database is *db_size*. Each transaction's reads are uniformly distributed between *min_size* and *max_size*. Each object read by a transaction is written with probability *write_prob*. The number of user terminals in the system is *num_terminals*. *Obj_cpu* and *obj_io* give the service times of CPU and IO resources respectively. If a transaction restarts due to deadlock, it is delayed by an amount controlled by an exponential distribution whose mean is average response time times $MPL/num_terminals$, to ensure that transactions are not delayed for long at lower MPLs. After each transaction, a terminal waits for a time that is exponentially distributed with mean *ext_think_time*. For BSA-FC, we varied the currency bound from 0 to 100 seconds. Distribution cost involves reading the log. We assume that the log is sequential and has an IO cost of 5 milliseconds (ms) per page plus 30 ms seek time.

¹ The legend applies to next two figures too.

Further, we assume that about 100 updates can be stored in a page. We assume network cost similar to IO cost, but we only delay the corresponding action instead of charging the IO resources.

Performance Metrics: We only present system throughput. But to verify that our results are consistent, we measured other performance metrics too, including response time, resource utilization, conflict, and restart ratios. We ran each experiment 10 times, with sufficiently long simulation time to produce 90 percent confidence intervals, which we omit from the graphs for clarity.

7.3 Single-Site Experiments

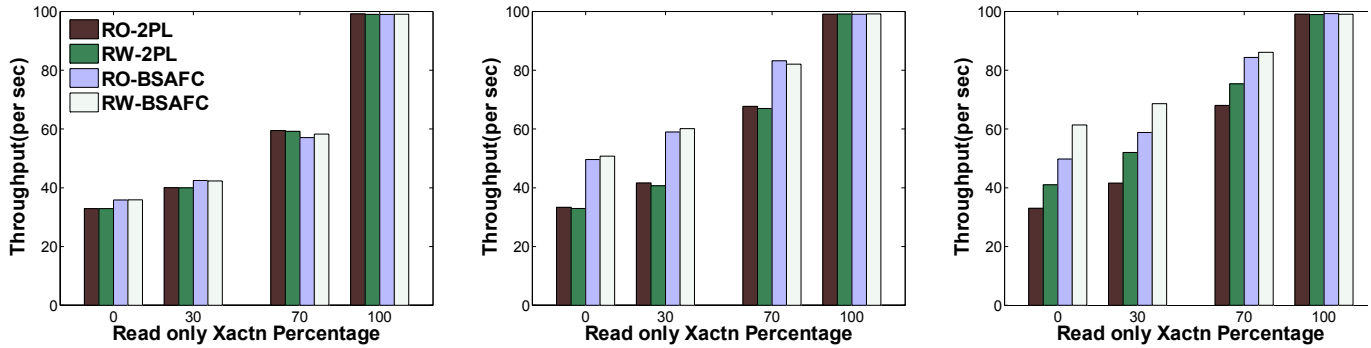
The goal of our single-site experiments is to compare different protocols with respect to data and resource contention. We compare BSA-FC (with three different currency bounds), Strict 2PL (two-phase locking with write locks held till after commit) and Read Committed (which is BSA without copiers). We control data contention by varying the MPL. We control resource contention by varying the number of resources. Our 1st experiment examines high resource and data contention. The 2nd and 3rd look at data contention and resource contention in isolation. We found similar differences between Strict 2PL and Read Committed as in [8].

7.3.1 Expt 1: Limited Resources

A resource unit is 1 CPU and 2 disks. We fixed the number of resource units to 1 and varied the MPL. The throughput is shown in Figure 5. Resource contention increases significantly as the MPL increases. This causes long response times (40~60s), and hence more constraint failures for BSA-FC with tight currency bounds (≤ 10), leading to Strict 2PL outperforming BSA-FC. On the other hand, BSA-FC with a large enough currency bound (100s) performs as well as Read Committed, i.e., the guarantees on freshness come “for free”. Here, the difference between the time of invalidation and the commit-time of the reading transaction could be the length of the transaction. (On average, this is 40-60s, but there is no limit on outlier cases.)

7.3.2 Expt 2: Infinite Resources

We set the number of resources in the system to a large number (“infinity”) to eliminate resource contention. Although this is not realistic, it serves as an upper bound. As shown in Figure 6, BSA-FC outperforms Strict 2PL even with currency bound 0. This is because 2PL's throughput is limited at low MPLs due to its blocking behavior, and at higher levels due to lock thrashing. In contrast, BSA-FC achieves increasing throughput until it reaches a performance plateau. The conflict ratio (number of times a transaction is blocked) is 2.7 at MPL 200 for 2PL but less than 0.1 for BSA-FC. Response time at this point was observed to be about



0.7 sec. With 1 sec think time between transactions and 200 terminals, increasing MPL above this level does not increase throughput as the remaining terminals are in the think state. In a single-site setting, the only way constraint checks can fail in BSA-FC is if a transaction reads a value and takes longer to complete than the associated currency bound. Since the response time in this experiment is 1~2 sec, BSA-FC with currency bound 10 does as well as Read Committed.

7.3.3 Expt 3: Multiple Resource Units

This experiment fills the gap between the previous two by varying the number of resource units from 5 to 50. Another important parameter here is the MPL. Setting it to lower values results in the underutilization of resources. Setting it to higher values results in thrashing. We fixed its values for each protocol to maximize its throughput, based on the previous experiments and results in [2]. Strict 2PL's performance peaks at a MPL of 50. BSA-FC reaches a plateau above 100. The throughput results are shown in Figure 7. Increasing the resources in the system increases the throughput thanks to increased parallelism. But the increase is more for BSA-FC, because there is less blocking compared to Strict 2PL (conflict ratio 0.33 vs. 0.05 with 50 resource units).

In summary, in the single-site scenario BSA-FC outperforms Strict 2PL if the system has sufficient resources. BSA-FC also gives much higher throughput (a factor of 3) with currency bounds of a few times the response time, which we believe is appropriate for many practical situations. As currency bounds are relaxed, BSA-FC achieves the throughput of Read Committed, which is the expected ideal, performance-wise.

7.4 Master-Cache Experiments

In the approach proposed in [13], only read-only transactions can execute at a cache. The RCSR theory developed in this paper allows us to improve on this by doing reads at the cache even for transactions that also do updates (which, of course, are still sent to the master). In this section, we evaluate the performance gains due to this improvement. We also study how the value of the refresh interval relative to currency bounds affects performance.

7.4.1 Expt 4: Reads of RW Transactions at Caches

In [13], the authors proposed extending SQL with currency and consistency constraints for caches that only handle read-only transactions. What do we gain by executing the reads of read-write transactions at the caches? To simulate their scenario, we run Strict 2PL at the master and BSA-FC at the cache. In scenario RO-2PL, the cache only handles read-only transactions and forwards all read-write transactions to the master. In scenario

RW-2PL, the cache also handles reads from read-write transactions (master checks their freshness). In another setting, we modify these two scenarios by using BSA-FC at the master, and refer to them as RO-BSAFC and RW-BSAFC respectively.

In this set of experiments, there is one master and one cache. The master has 10 resource units and 200 terminals. The cache has 5 resource units and 100 terminals. We add a new parameter, *read-only transaction percentage*; decreasing it increases data contention. The refresh interval is set to 0, i.e., we continuously refresh the cache. Each site's MPL is set to the optimum value for its read-only transaction percentage, as in Section 7.3.3.

The results are shown in Figure 8, Figure 9 and Figure 10, corresponding to currency bound 0, 5 and 10 seconds, respectively. In each figure, we show the throughput of scenarios RO-2PL to RW-BSAFC as the read-only percentage increases.

As the read-only percentage increases, throughput increases. Also as the currency bound increases, throughput increases except for the 100% read-only case. Similar to the single-site experiments, BSA-FC dominates Strict 2PL. Even for small currency bounds, BSA-FC gains 20-35% throughput over Strict 2PL. When the currency bound is zero (Figure 8), all the scenarios behave similarly as most transactions fail the freshness constraint check. As the currency bound increases, BSA-FC improves over Strict 2PL (Figure 9). Further, with currency bound 10 (Figure 10), the scenarios with read-write transactions reading at the cache dominate those executing only read-only transactions at the cache.

7.4.2 Expt 5: Refresh Interval vs. Currency Bound

Clearly, the performance of BSA-FC depends greatly on the time between cache refreshes, relative to currency bounds. In this experiment, we study the effect of varying the refresh interval from 0 to 100 seconds, while also varying currency bounds from 0 to 100 seconds. We consider the same settings as the previous experiment except that we fix the read-only percentage to 70. The results are shown in Figure 11. Once the refresh interval exceeds the currency bound, throughput drops drastically; at this point, the throughput is solely due to the master. Thus, it is important to refresh more often than typical currency bounds in transactions.

7.5 Scalability Experiments

Earlier sections showed the benefit of BSA-FC in single-site and master-cache scenarios. Next, we study scalability. For *scale-out*, we increase the number of caches. For *scale-up*, we increase the number of resource units at a single site. Our results show that caching with BSA-FC achieves good scale-up and scale-out. Since caching is not very useful in a write-dominated workload,

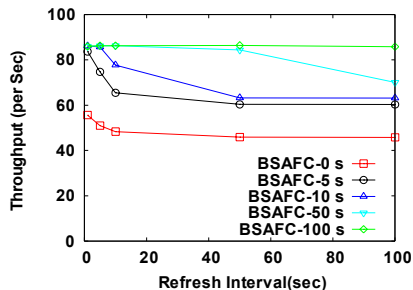


Figure 11: Refresh Int. Vs Cur. Bound

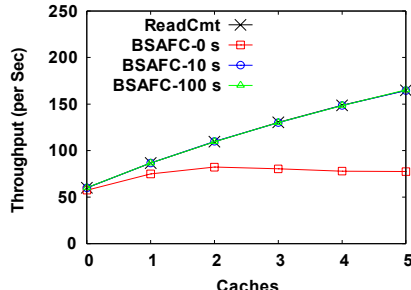


Figure 12: Scale Out

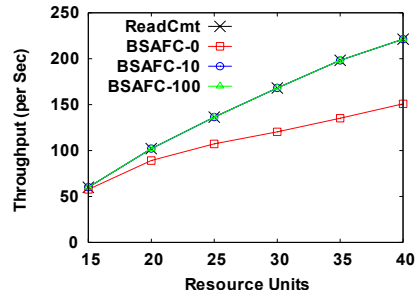


Figure 13: Scale Up

we target read-dominated ones, where reads frequently occur in transactions that contain some update operations.

7.5.1 Experiment 6: Scale-Out

We increased the number of caches in the system from 0 to 5. We associated 100 terminals with each additional cache. The master has 15 resource units and 100 terminals. We fix the read-only transaction percentage to 70. The throughput results are shown in Figure 12. They show a linear scale-out when the currency bounds are more than 10 seconds. With a currency bound of 0 seconds, freshness constraint checks often fail at caches as it takes time to propagate changes. With a currency bound of 10 seconds or higher, the updates are propagated and the freshness requirements are satisfied. Since we start with no caches, this experiment also sheds light on the gains due to caching.

7.5.2 Experiment 7: Scale-Up

The set-up for this experiment is very similar to our single-site ones, except that we varied the number of terminals in the system. We also varied the number of resources from 15 to 40. Throughput results are shown in Figure 13. The corresponding points in Figure 12's x-axis have the same number of resources in the system; the master with 0 caches has 15 resources and each cache has 5 more resources. As in the master-cache scenario, BSA-FC is scalable. The throughput increases with increased workload and resources. However, when the system has the same number of resources, single-site always performs better for the given workload, mainly because of replication overhead (copier, distribution and network) and concentrating resources at one site smoothes bursts in the workload. If resources are divided across cache sites, some sites have resource contention even while others have idle resources.

7.6 Master-Cache vs. Single Site

In this experiment we compare three caching configurations with a single-site configuration, to study how each deployment of the same resources performs on throughput and throughput-per-dollar. All configurations use 64 resource units (64 CPUs and 128 disks). The first configuration has 48 caches with 1 unit each, the second has 12 caches with 4 units each, and the third has 3 caches with 16 units each; the remaining units in each case are at the master.

The throughput results with 70 percent read-only transactions are shown in Figure 14. The 48-cache configuration performs poorly because the caches are spending all their resources executing copiers from the master. Observe that with fewer caches, and therefore more resources per cache, throughput steadily increases. To validate our results we repeated the experiment with 90% read-only transactions, and the throughput is uniformly higher

(Figure 15) because of the lower copier overhead. In particular, as the percentage of read-only transactions increases, throughput for the cache scenarios approaches that of the single-site configuration. Although a single-site system with the same number of resources achieves better performance, as expected, the cost of a single-site system grows much faster than the cost of a loosely-coupled master-cache system with the same number of resources. Thus, the price-performance achievable with caching and BSA-FC is significantly better than for a single-site system. To make this point, we show the throughput per dollar for each of these settings at a 90% read-only level (Figure 16), using system costs from the TPC website (www.tpc.org). We believe the benefits of caching can be considerably increased by caching only portions of the database, rather than the entire database at each cache; however, such intelligent caching strategies are not considered in this paper.

8 OTHER IMPLEMENTATION ISSUES

As in any concurrency control algorithm, many variations and optimizations of BSA-FC are possible. We mentioned many of them throughout the paper, and briefly describe a few others here.

Reflexive Reads: A read is reflexive if its transaction previously wrote the same item. We briefly introduced them in Section 5.2.3, suggesting they be executed at the master. This requires that the system identify them. One way is to tag a transaction that might perform a reflexive read and run it entirely at the master. Another is to have the master tell the cache which items the transaction writes. When the cache receives a read, it checks whether the transaction previously wrote it. To limit the overhead, an item could be coarse-grained, such as a table. It could even be the entire database, which means all of a transaction's reads that follow its first write execute at the master. This simplifies the bookkeeping but forces some reads to execute at the master unnecessarily.

Insertions and Deletions: Our model can easily be extended to insertions and deletions. Since inserts and deletes do not return data to their caller, and do not carry freshness constraints they can be executed at the master like any other update. This can include setting index-range locks or other locks used to avoid phantoms. Predicate-based queries, such as a SQL `Select`, use indices and other auxiliary structures for deciding what to access and for avoiding phantoms (e.g., using index-range locking), not just in a master but also in a cache. Copiers need to include a *last-modified* timestamp on these structures, since these timestamps are used when generating timestamp constraints for time-drift constraints.

For deletes, copiers need to leave a tombstone (i.e., empty object) that includes the deleted item's *last-modified* and *valid-till* time-

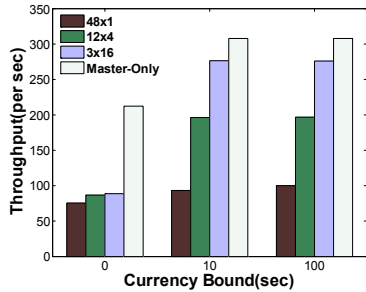


Figure 14: 70% Read Only Xactns¹

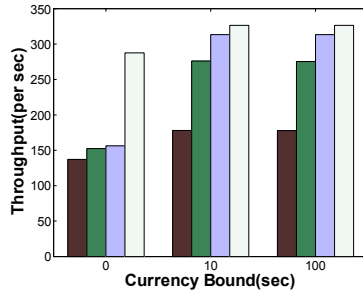


Figure 15: 90% Read Only Xactns

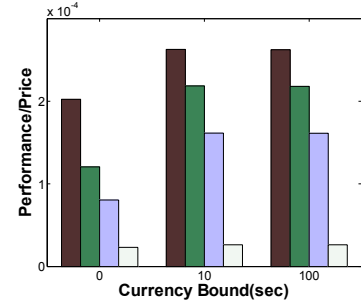


Figure 16: 90% Read Only Xactns

stamps. To see why, consider a transaction T that issues a query that reads all objects satisfying a predicate P and suppose a deleted object satisfies P . Then the deleted object’s timestamps must be considered when generating constraints on T ’s timestamp.

Read time currency check: Suppose we evaluate constraints relative to the timestamp when the read executes or when the transaction starts executing, rather than transaction commit time. Thus, constraints can be evaluated during the execution of the read, rather than being delayed till the commit time. As this allows cache managers to assign timestamps, some clock synchronization between the master and caches is needed to avoid timing anomalies.

Thomas Write Rule: Thomas’ Write Rule (TWR) can be used to avoid requiring updates to be pipelined from the master to caches [27]. This is commonly used in multi-master replication. In TWR a write $w_i[x^A]$ is applied to x^A only if $ts(T_i) \geq ts(x^A)$. In our case, $w_i[x^A]$ is issued by a copier so T_i is the timestamp of the user-transaction T_i that induced the copier execution.

The key property of TWR is that given a set of writes into x^A , the final value of x^A is the value written by the copier with largest timestamp, which is independent of the order in which the writes are applied. Thus, TWR is ordinarily used to avoid requiring that writes be applied in timestamp order. If we no longer require that copiers apply updates to each cache in timestamp order, then we can no longer use the timestamp of the latest update to the cache as the value of *valid-till*. However, the four alternative techniques for determining *valid-till* in Section 5.2.2 still apply.

9 RELATED WORK

Tradeoffs between data freshness and availability, concurrency, and maintenance costs have been explored extensively. We discuss the most closely related work in this section.

Guo et al. proposed that applications be allowed to specify relaxed currency and consistency requirements [13]. They introduced freshness constraints and corresponding extensions to SQL syntax, and showed how to modify query optimization and execution to handle queries with such constraints. In [12] this was extended to deal with row-level caching and currency/consistency constraints rather than table-level. However, they only considered read-only queries. So they did not address the semantics of interleaved executions or concurrency control issues that arise when update transactions are allowed to see out-of-date copies. Their algorithms check whether a copy is suitable at compile time or when the read occurs, without commit time checks as in our work.

FAS [25] also allows a query to specify a currency bound. It only supports database-granularity currency control, using a freshness

index that is loosely-correlated to the delay of cache refresh. The implementation uses middleware that routes each query to a database whose freshness is appropriate. In [3] this is extended to allow table-granularity decisions, and to allow a query to access multiple databases. In both papers, queries need commit-duration locks, unlike the short-duration locks we take. Because this work demands exact consistency (in our terminology, zero drift) the correctness criterion is one-copy serializability, not RCSR.

Techniques to improve concurrency by relaxing data freshness requirements were introduced in [11][14][19]. They allow use of out-of-date copies for hot-spots, which are frequently-updated items that can be incremented or decremented, and where reads use the range of an item’s value rather than its exact value. In [1], this approach was generalized to arbitrary objects through consistency assertions on the database. The main idea is that in an equivalent serial execution, transactions are only required to see the same consistency assertions, rather than the same database states.

In [21] the authors aim to increase concurrency while reducing abort rate in a multiversion system. They explicitly require full serializability for update transactions, i.e., they do not allow divergence for reads in update transactions. They measure database-level inconsistency (fuzziness), while we measure divergence in terms of user-specified currency limits on individual objects or groups of objects read by a transaction.

Our use of error bounds is related to epsilon-serializability [20], which allows queries to read concurrently updated data while bounding the resulting inconsistency. However, their notion of inconsistency is limited to changes made by concurrently executing transactions. Thus, a query that does not read objects modified by concurrent transactions has no inconsistency in their sense. But it could still read an out-of-date copy created by a transaction that committed prior to the reader. In our setting, this read may or may not be allowed, depending on its freshness constraint. Further, epsilon-serializability has a quantitative model of inconsistency. It assumes there is a measure of “distance” between two database states and transactions can accumulate inconsistency up to a bound; there is no such notion in our approach. The only formal accounting of epsilon-serializability is [22], which only considers histories where updaters are not allowed to see any inconsistency.

Of the approaches that use integrity constraints on databases for improved concurrency, the closest to our work is probably the *predicate-wise serializability (PWSR)* model [16][17][24] which includes features needed by long transactions in design applications. The key idea is that often only projections of histories that access subsets of the database need to be serialized, and these data subsets can be specified by integrity constraints. Korth and

Speegle [17] proposed extensions of PWSR to handle versions and nested transactions. One of their correctness classes, multiversion serializability, is similar to RCSR, in that it allows each read to be given any version that satisfies the read's precondition and allows different reads in a transaction to be given versions of the same item. In this respect, our model specializes theirs. Their predicates can refer only to data items and constants, not to versions or timestamps, and thus cannot express freshness constraints. Their model allows a broad class of predicates over database values as pre-and post-conditions for transactions; in contrast, ours does not allow post-conditions, and only reads have associated pre-conditions. Moreover, their model does not consider replication or distinguish the user's execution on a multiversion database from the system's execution.

The following papers study availability and autonomy of distributed databases when local data is allowed to diverge from the master copy: [10][5][28][4][29]. They differ from each other in divergence metrics, the concrete update protocols, and the guaranteed divergence bound; none of these approaches support a transaction containing both updates and divergence-bounded queries.

Divergence caching [15] and Olston's work [18] consider how to set optimal bounds for approximate values given queries with precision bounds and an update stream. For real-time data updated by sensors, [23] discusses the frequency of updates needed to keep items acceptably fresh. [26] considers how to ensure that reads see fresh enough data in a content-distribution network by optimizing the allocation of many reads with value-bounds among copies whose value-divergence is known. These papers allow precision or freshness bounds only on the queries.

10 CONCLUSION

We presented a new transaction model that allows update transactions to read stale data satisfying their freshness constraints. Our main results are: (i) a formal correctness criterion for this model, called RC-serializability, which requires an execution to be equivalent to a serial execution on a multiversion database where the freshness constraints of reads are satisfied; (ii) new algorithms that implement a variety of constraints; (iii) a proof that these algorithms are correct; and (iv) a simulation study that shows that the new algorithms perform like Read Committed for reasonable freshness levels, while also guaranteeing freshness constraints. We also discussed some optimization and implementation issues.

11 REFERENCES

- [1] D. Agrawal, A. Abbadi, A.K. Singh. Consistency and Orderability: Semantics-Based Correctness Criteria for Databases. *TODS* 18, 3 (1993).
- [2] R. Agrawal, M. Carey, M. Livny. Concurrency Control Perf. Modeling: Alternatives and implications. *TODS* 12, 4 (1987)
- [3] F. Akal, C. Tuerker, H.-J. Schek, Y. Breitbart, T. Grabs, L. Veen. Fine-Grained Replication and Scheduling with Freshness and Correctness Guarantees. *VLDB*, 2005.
- [4] R. Alonso, D. Barbará, H. Garcia-Molina, S. Abad. Quasi-Copies: Efficient Data Sharing for IR Systems. *EDBT* 1988.
- [5] D.Barbará and H. Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Linear Arithmetic Constraints in Distributed Database Systems. *EDBT*, 1992.
- [6] P. Bernstein, V. Hadzilacos, N. Goodman. *Concurrency Control and Recovery in DB Systems*. Addison-Wes., 1987.
- [7] P. Bernstein, A. Fekete, H. Guo, R. Ramakrishnan, P. Tamma. Relaxed Currency Serializability for Middle-Tier Caching (extended version). Microsoft Research Technical Report, MSR-TR-2006-34, 2006.
- [8] P.M. Bober, M. J. Carey: On Mixing Queries and Transactions via Multiversion Locking. *ICDE* 1992.
- [9] CSIM: <http://www.mesquite.com>
- [10] H. Garcia-Molina and G. Wiederhold. Read-Only Transactions In A Distributed Database. *TODS*, 1982.
- [11] D. Gawlick and D. Kinkade. Varieties of Concurrency Control in IMS/VIS FastPath. *IEEE Database Engg.*, 1985.
- [12] H. Guo, P.A. Larson, R. Ramakrishnan. Caching with "Good Enough" Currency, Consistency and Completeness. *VLDB* 2005.
- [13] H. Guo, P.A. Larson, R. Ramakrishnan, J. Goldstein, Relaxed Currency and Consistency: How to Say "Good Enough" in SQL. *SIGMOD* 2004.
- [14] T. Haerder. Handling Hot-Spot Data in DB-Sharing Systems. *Information Systems*, 1988.
- [15] Y. Huang, R. Sloan, and O. Wolfson. Divergence Caching in Client Server Architectures. *PDIS*, 1994.
- [16] H.F. Korth, W. Kim and F. Bancilhon, A Model of CAD Transactions. *VLDB*, 1985.
- [17] H.F. Korth and G.D. Speegle, Formal Aspects of Concurrency Control in Long-Duration Transaction Systems Using the NT/PV Model. *TODS*, 1994.
- [18] C. Olston, B. Loo, and J. Widom. Adaptive Precision Setting for Cached Approximate Values. *SIGMOD*, 2001.
- [19] P.E. O'Neil. The Escrow Transaction Method. *TODS* 1986.
- [20] C. Pu and A. Leff. Replica Control In Distributed Systems: An Asynchronous Approach. *SIGMOD*, 1991.
- [21] C. Pu, M.K. Tsang, K.-L. Wu, P.S. Yu: Multiversion Divergence Control of Time Fuzziness. *CIKM* 1994.
- [22] K. Ramamritham and C. Pu, A Formal Characterization of Epsilon Serializability. *TKDE*, 1995.
- [23] K. Ramamritham, S. Son, L. Dipippo. Real-Time Databases and Data Services. *Real-Time Systems*, 28, 2004.
- [24] R. Rastogi, S. Mehrotra, Y. Breitbart, H.F. Korth, A. Silberschatz. On Correctness of Non-Serializable Executions. *JCSS*, 1998.
- [25] U. Röhm, K. Böhm, H. Schek, and H. Schuldt. FAS—A Freshness-Sensitive Coordination Middleware for a Cluster of OLAP Components. *VLDB*, 2002.
- [26] S. Shah, K. Ramamritham, C. Ravishankar. Client Assignment in Content Dissemination Networks for Dynamic Data. *VLDB*, 2005.
- [27] R. H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *TODS*, 1979.
- [28] G. Wiederhold and X. Qian. Modeling Asynchrony in Distributed Databases. *ICDE*, 1987.
- [29] H. Yu and A. Vahdat. Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services. *TOCS*, 23, 3, (2002).