Database Replication Using Generalized Snapshot Isolation

Sameh Elnikety†, Fernando Pedone‡, and Willy Zwaenepoel†

†School of Computer and Communication Sciences
EPFL
Lausanne, Switzerland

‡Faculty of Informatics
Università della Svizzera Italiana,
Lugano, Switzerland

Abstract

Generalized snapshot isolation extends snapshot isolation as used in Oracle and other databases in a manner suitable for replicated databases. While (conventional) snapshot isolation requires that transactions observe the "latest" snapshot of the database, generalized snapshot isolation allows the use of "older" snapshots, facilitating a replicated implementation. We show that many of the desirable properties of snapshot isolation remain. In particular, read-only transactions never block and they do not cause update transactions to block or abort. Moreover, under certain assumptions on the transaction workload the execution is serializable.

An implementation of generalized snapshot isolation can choose which past snapshot it uses. An interesting choice for a replicated database is *prefix-consistent snapshot isolation*, in which the snapshot contains at least all the writes of locally committed transactions. We present two implementations of prefix-consistent snapshot isolation. We conclude with an analytical performance model of one implementation, bringing out the benefits, in particular reduced latency for read-only transactions, and showing that the potential downsides, in particular change in abort rate of update transactions, are limited.

Keywords: distributed transaction processing, databases, distributed objects and middleware systems.

1 Introduction

There is increasing interest in replicating the generation of dynamic Web content [2, 15] for many Web services to achieve efficiency and fault tolerance. The user's request is executed on a "nearby" replica, thereby avoiding long roundtrip delays and distributing the load over the replicas. In addition, replication increases the availability of a Web service because if a replica fails, information can still be accessed at other replicas. In a centralized dynamic content Web site, the dynamic data is most often stored in a database [8, 3]. Therefore, one of the key problems in replicating dynamic Web content generation is replicating the database and keeping the replicas consistent.

Ideally, we would like to provide the same database consistency as in the centralized database case. We focus here on databases providing snapshot isolation [6]. In such a database, a transaction T obtains at the beginning of its execution the latest snapshot of the database, reflecting the writes of all transactions that have committed before transaction T starts. At commit, the database checks that the writeset of transaction T does not intersect with the writesets of the transactions that committed since T's snapshot. If there is a non-zero intersection, transaction T aborts; otherwise, it commits.

Snapshot isolation is popular for a number of reasons, not the least of which is the pragmatic reason that Oracle [9, 20] and other database vendors [26, 33, 19] use it: many databases offer snapshot isolation as their strongest transaction isolation level. More fundamentally, snapshot isolation never requires read--only transactions to be blocked or aborted, and read-only transactions do not cause update transactions to be blocked or aborted. This advantage is significant for workloads with a large fraction of read-only transactions (such as those resulting from the generation of dynamic Web content). Snapshot isolation provides a weaker form of consistency than serializability, but one that programmers understand and use. Moreover, Fekete et al. [12, 13] have recently demonstrated that under certain conditions on the workload transactions executing on a database with snapshot isolation produce serializable histories.

1.1 Generalized Snapshot Isolation (GSI)

Extending snapshot isolation to replicated databases is not straightforward. Intuitively, the problem stems from the requirement that a transaction must see the "latest" snapshot when it starts execution. In contrast with the centralized case, the notion of "latest" is not a priori well-defined in a distributed setting. Implementations of an ordering that defines the notion of "latest" and makes the corresponding snapshot available may impose a delay at the start of a transaction. Imposing such a delay at the beginning of read-only transactions voids one of the main benefits of snapshot isolation.

Generalized snapshot isolation is based on the observation that a transaction need not necessarily observe the "latest" snapshot. It can observe an older snapshot, and many properties as those in (conventional) snapshot isolation continue to hold. Conditions can be identified that guarantee serializable execution. With a suitable choice of "older", read-only transactions execute without delay or aborts, and they do not cause update transactions to be delayed or aborted. Read-only transactions, however, may observe somewhat older data. To commit an update transaction, its writeset must be checked against the writesets of recently committed transactions, as before. The probability of an abort increases, as it becomes more likely that at the time of its commit an update transaction finds that another transaction has written to the same data as it did since the time of its snapshot.

The ability to start from an older snapshot gives rise to an interesting spectrum of possibilities with attendant performance tradeoffs. At one end of the spectrum is the conventional notion of snapshot isolation, further referred to in this paper as *conventional snapshot isolation,* in which each transaction reads the latest snapshot. This is clearly appropriate in a centralized setting where the latest snapshot is trivially available, and where using the latest snapshot minimizes the probability of aborts. At the other end of the spectrum is the trivial solution in which all transactions observe the initial snapshot (i.e., the database state as in the beginning of execution).

1.2 Prefix-Consistent Snapshot Isolation (PCSI)

In a replicated setting, an interesting positioning in this spectrum is for a transaction to take as its initial snapshot the latest snapshot that is locally available on its replica. We capture this idea in *prefix-consistent snapshot isolation.* As an instance of generalized snapshot isolation, prefix-consistent snapshot isolation maintains the desirable properties of generalized snapshot isolation. Read-only transactions never block or abort, and they do not cause update transactions to block or abort. Moreover, transactions observe at least all the writes that have committed on the local replica before they start. This property is important in a workflow of transactions, in which a user submits a sequence of transactions to a replica.

1.3 Contributions

The contributions of this paper are as follows: **(1)** introducing generalized snapshot isolation, **(2)** establishing two sufficient conditions (one dynamic condition and one statically checkable condition) that guarantee serializable execution under generalized snapshot isolation, **(3)** introducing prefix-consistent snapshot isolation and two implementation strategies for replicated databases, and **(4)** characterizing the relative performance of prefix-consistent snapshot isolation to conventional snapshot isolation.

1.4 Roadmap

This paper is organized as follows: We present the database model in Section 2 and develop generalized snapshot isolation in Section 3. Then we discuss how to guarantee serializability in Section 4, and provide a dynamic and a static condition for serializability under generalized snapshot isolation.

In section 5, we present prefix-consistent snapshot isolation, which is an instance of generalized snapshot isolation that is suitable for replicated databases. In section 6, we present a model for a distributed system with replicated databases. We first show that conventional snapshot isolation must impose a delay on read-only transactions in that model, and then we develop two algorithms that implement prefix-consistent snapshot isolation , one using centralized certification and the other using distributed certification. In Section 7, we compare analytically the performance of prefix-consistent snapshot isolation to conventional snapshot isolation. Finally, we discuss related work in Section 8 and derive our conclusions and future research directions in Section 9. We prove the propositions and correctness of algorithms in a technical report [10].

2 Database Model and Definitions

2.1 Database Model

We assume that a *database* is a collection of uniquely identified data items. Several versions of each data item may co-exist simultaneously in the database, but there is a total order among the versions of each data item. A *snapshot* of the database is a committed state of the database.

A *transaction* T_i is a sequence of read and write operations on data items, followed by either a commit or an abort operation. We denote T_i's write operation on item X by $W_i(X_i)$. If T_i executes $W_i(X_i)$ and commits, then a new version of X, denoted by X_i, is added to the database. Moreover, we denote T_i's read operation on item X_j by $R_i(X_j)$. T_i's commit or abort is denoted by C_i or A_i, respectively. Finally, to simplify the presentation, we assume that transactions do not contain redundant read or write operations[1]: A transaction reads any item at most once and writes any item at most once. And, if a transaction writes an item, it does not read that item afterwards.

A transaction is *read-only* if it contains no write operation, and is *update* otherwise. The readset of transaction T_i, denoted $readset(T_i)$, is the set of data items that T_i reads. Similarly, the writeset of transaction T_i, denoted $writeset(T_i)$, is the set of data items that T_i writes. We add additional information to the writesets to include the new values of the written data items.

A history h over a set of transactions $\mathcal{T} = \{T_1, T_2, ..., T_n\}$ is a partial order $\prec$ such that **(a)** h contains the operations of each transaction in $\mathcal{T}$; **(b)** for each $T_i \in \mathcal{T}$, and all operations O_i and O_i' in T_i: if O_i precedes O_i' in T_i, then $O_i \prec O_i'$ in h; **and (c)** if T_i reads X from T_j, then $W_j(X_j) \prec R_i(X_j)$ in h [7].

2.2 Impacting Transactions

To simplify some definitions in the paper, we assign a distinct time to each database operation, resulting in a total order consistent with the partial order $\prec$ for history h.

In generalized snapshot isolation, each transaction observes a snapshot of the database that is taken at some time denoted $snapshot(T_i)$. If transaction T_i sees a snapshot of the database taken at time $snapshot(T_i)$, then this snapshot includes the updates of all transactions that have committed before $snapshot(T_i)$. To argue about the timing relationships among transactions, we use the following definitions with respect to transaction T_i:

- $snapshot(T_i)$: the time when T_i's snapshot is taken.
- $start(T_i)$: the time of the first operation of T_i.

[1]This assumption is not restrictive: These redundant operations can be added to the model, but they will complicate the presentation (e.g., the rules D1 and D2 of GSI). Moreover, in practice if a transaction contains redundant operations, then they can be removed using local variables in the program that issues the transaction [23].

- $commit(T_i)$: the time of C_i, if T_i commits.
- $abort(T_i)$: the time of A_i, if T_i aborts.
- $end(T_i)$: the time of either C_i or A_i.

Notice that $snapshot(T_i) \leq start(T_i) < end(T_i)$. The time of the first operation of T_i defines $start(T_i)$ since we do not use an explicit transaction begin operation. Next, we define the relation *impacts* for update transactions.

- $\underline{T_j \textit{ impacts } T_i}$ **iff**
  $writeset(T_i) \cap writeset(T_j) \neq \emptyset$, **and** $snapshot(T_i) < commit(T_j) < commit(T_i)$

From this definition, only committed update transactions may *impact* update transaction T_i. Read-only transactions and uncommitted transactions cannot *impact* T_i. When committing an active update transaction T_i, we say "T_j impacts T_i", which means that if T_i were to commit now, then T_j would impact T_i.

3 Generalized Snapshot Isolation (GSI)

Generalized snapshot isolation (GSI) has two rules: the first regulates read operations, and the second regulates commit operations. For any history h created by GSI, the following two properties hold: (The definitions below implicitly assume that indexes i, j, and k are different values.)

- **D1. (GSI Read Rule)**
  $\underline{\forall T_i, X_j \text{ such that } R_i(X_j) \in h :}$
  1- $W_j(X_j) \in h$ **and** $C_j \in h$;
  2- $commit(T_j) < snapshot(T_i)$;
  3- $\forall T_k$ such that $W_k(X_k), C_k \in h$:
  $[commit(T_k) < commit(T_j)$ **or** $snapshot(T_i) < commit(T_k)]$.

- **D2. (GSI Commit Rule)**
  $\underline{\forall T_i, T_j \text{ such that } C_i, C_j \in h :}$
  4- $\neg(T_j \text{ impacts } T_i)$.

The read rule D1 ensures that each transaction reads only committed data, that is, each transaction observes a committed snapshot of the database. This snapshot could be any snapshot that has been taken before the transaction starts, which can be provided efficiently in a distributed system. Part one and two of D1 allow T_i to read the data items written by transactions that committed before T_i's snapshot. The third part of D1 prevents holes in the snapshot that T_i receives: If T_i sees a snapshot at time t and reads data item X, then it reads X from the last committed transaction at time t that updated X.

The commit rule D2 prevents any update transaction T_i from committing if it will be impacted by another committed update transaction. From the definition of the relation *impacts*, D2 is trivially satisfied for read--only transactions. This rule gives the "first-committer-wins behavior" [7]: If two active transactions update the same item, then the transaction that commits first succeeds at updating that item; whereas the other transaction aborts.

When an active update transaction T_i requests to commit in a database system running GSI, the database performs a test to decide if T_i can commit or must abort. This test is called *certification.* In other words, to dynamically enforce the commit rule D2 when deciding whether an update transaction T_i can commit, the database *certifies* T_i: the database checks whether $writeset(T_i)$ intersects with the writeset of any update transaction that committed after $snapshot(T_i)$. If all such intersections are empty, T_i commits; otherwise T_i aborts.

Conventional snapshot isolation (CSI) is a special case of GSI where each transaction sees the *last* snapshot with respect to its starting time, i.e., $snapshot(T_i) = start(T_i)$. This is referred to in literature simply as "snapshot isolation". On a system composed of a single site, it is easy to implement conventional snapshot isolation, as the latest snapshot is available for use without delay. Using the latest snapshot, in general, minimizes the number of transactions that may impact an update transaction. Therefore in a centralized database, CSI typically reduces the abort rate of update transactions and provides read-only transactions with the most recent snapshots.

4 Serializability under GSI

We now turn to discussing serializability under GSI. Serializability is important; it is the basic correctness criteria in much work on databases [7, 23]. If applications need serializability, then providing a weaker form of consistency makes those applications much harder to develop. GSI provides a weaker form of consistency than serializability.

Both GSI and CSI may produce non-serializable histories. Here is an example of the write-skew anomaly [6]. Assume that X and Y are the balances of two bank accounts for the same person and that the bank enforces the constraint $(X + Y) \geq 0$. The next two procedures deduct an input amount, $A1$ and $A2$, from X and Y, respectively.

```
$Proc_i$ = {  input $A1$;  read $X$ and $Y$;
              if $((X+Y) < A1)$  then {abort} else $\{X = X - A1;$  write $X$;  commit}  }
$Proc_j$ = {  input $A2$;  read $X$ and $Y$;
              if $((X+Y) < A2)$  then {abort} else $\{ Y = Y - A2;$  write $Y$;  commit}  }
```

Running these procedures concurrently as transactions T_i and T_j can produce the following history under CSI. Assume that X_0, Y_0, $A1$, and $A2$ are 50, 50, 60, and 60, respectively. h_1 = $R_i(X_0 = 50)$, $R_i(Y_0 = 50)$, $R_j(X_0 = 50)$, $R_j(Y_0 = 50)$, $W_i(X_i = -10)$, C_i, $W_j(Y_j = -10)$, C_j. History h_1 is not serializable.

4.1 Dynamic Serializability Condition (D3)

D3 is a dynamic condition: it adds a new rule to GSI. Transactions that satisfy condition D3 produce serializable histories under GSI.

- **D3. (Dynamic Serializability Rule)**
  $\forall$ update transactions T_i, T_j such that $C_i, C_j \in h$ **and** $snapshot(T_i) < commit(T_j) < commit(T_i)$:
  $readset(T_i) \cap writeset(T_j) = \emptyset$.

PROPOSITION 1. *GSI is serializable under D3.*

To use this condition we add rule D3 to the two rules, D1 and D2, of GSI. Thus, rules D1, D2, D3 are applied at runtime. In this case, when certifying an update transaction T_i, first GSI requires T_i not to have any impacting transaction, which can be verified by comparing $writeset(T_i)$ to the writesets of transactions that committed since T_i's snapshot, i.e., $\{T_j : snapshot(T_i) < commit(T_j)\}$. Secondly, condition D3 requires $readset(T_i)$ not to intersect with the writesets of the same set of committed update transactions $\{T_j : snapshot(T_i) < commit(T_j)\}$. Therefore to certify T_i, both its readset and writeset must be available for the dynamic checks, as well as the writesets of the update transactions which committed since T_i's snapshot.

D3 is sufficient but not necessary. Consider this history: h_2 = $W_i(X_i)$, C_i, $W_j(X_j)$, $R_k(X_i)$, $W_k(Y_k)$, C_j, C_k. History h_2 is serializable but does not satisfy D3.

If more information is made available (in particular the readsets and writesets of committed and active transactions), then we can ensure serializability, for example by building the serializability graph [7, 24]. Using the readsets, however, is likely to incur a higher overhead than using only writesets, especially in a distributed setting. Next we present static condition S1, which avoids requiring the readset to certify an update transaction. Roughly speaking, "if we can determine the readsets and writesets of transactions statically, then we can apply condition D3 on all transaction pairs statically rather than dynamically". S1 allows only a subset of the histories allowed by D3.

4.2 Static Serializability Condition (S1)

S1 is a static condition; it adds no extra dynamic check to GSI. This condition can be *checked* statically by examining the mix of transactions and query templates in application programs. It can also be *enforced* by changing the transactions as described later.

- **S1. (Static Serializability Condition)**
  $\forall$ update transactions T_i, T_j :

$$\left[\begin{array}{c} [writeset(T_i) \cap writeset(T_j) \neq \emptyset] \\ \vee \\ \Big(\; [readset(T_i) \cap writeset(T_j) = \emptyset] \quad \wedge \quad [writeset(T_i) \cap readset(T_j) = \emptyset] \; \Big) \end{array}\right]$$

PROPOSITION 2. *GSI is serializable under S1.*

S1 is sufficient but not necessary. For example, consider this history: h_3 = $W_i(X_i)$, C_i, $W_j(X_j)$, $R_k(X_i)$, $W_k(Y_k)$, C_k, C_j. History h_3 does not satisfy S1 as $writeset(T_j) \cap readset(T_k) \neq \emptyset$. However, h_3 is serializable and is allowed by condition D3.

Read-only transactions trivially satisfy condition S1. Therefore in practice, to use condition S1 all possible update transactions must be known. That is usually the case in many database applications where database transactions are hidden from clients who interact with the system through a standard interface (e.g., application forms which contain SQL templates). In such a system, ad–hoc update transactions must be prevented, unless they are checked to satisfy condition S1. Fortunately, in many situations such ad–hoc update transactions are forbidden because they need to follow the procedures of the business logic. Nonetheless, ad–hoc read-only transactions are allowed as they always satisfy S1.

To check if an application satisfies condition S1, we need to conservatively check every pair of update transactions. Some optimizations are possible if application specific information are available. For example, we do not need to check pairs of transactions that never run concurrently (i.e., are not active at the same time) in the database application. Such cases happen when two transactions always run sequentially in a single workflow under PCSI as explained in Section 5, or when one transaction runs during business hours and the other runs at night.

If a pair of two update transactions do not satisfy S1, it must be that they do not write any data item in common and one of them writes a data item, X, and the other reads that data item. We can change the second transaction, which reads X, to include an identity write on item X, i.e., change $R(X)$ into $R(X)$, $W(X)$. This change makes their writesets intersect; therefore, they now satisfy S1.

The process of checking update transactions—and altering them if needed—can be automated [12, 13]. Moreover in some applications, all transactions already satisfy S1. For example, the TPC-C and TPC--W benchmarks [31] are the industry-standard for evaluating on–line transaction processing systems and e-commerce systems, respectively. We, as well as other researchers [13], verified that both benchmarks satisfy condition S1 by examining all update transactions from the specification documents: TPC-C and TPC-W produce serializable histories under GSI without any modification.

5 Prefix-consistent Snapshot Isolation (PCSI)

GSI's rules D1 and D2 do not specify which snapshot of the database a transaction should observe. The range of possible snapshots a transaction can receive varies from the initial database state until the latest snapshot as in CSI.

Although applications can in general trade snapshot *freshness* for performance, especially in distributed environments, in some cases a transaction should be guaranteed to receive a snapshot that contains the updates of specific previous transactions. This happens, for example, when a transaction may need to observe the effects of previous transactions in the same application workflow. In order to capture this additional requirement on database snapshots, we introduce prefix-consistent snapshot isolation (PCSI). We define PCSI in this section, and in the next section we present a model for replicated databases and instantiate the workflow relation and give an example to show its importance. Then we discuss how to implement PCSI in replicated databases.

PCSI is parameterized by an equivalence relation $\sim$ among transactions. If transactions T_i and T_j are in the same workflow, denoted by $T_i \sim T_j$, and T_j commits before T_i starts, then T_i's snapshot must be recent enough to contain at least T_j's updates, that is, if $T_i \sim T_j$ and $commit(T_j) < start(T_i)$ then $commit(T_j) < snapshot(T_i)$. We add a new conjunction, part 4 below, to GSI rules D1 and D2 to obtain the following two properties which must hold for any history generated by PCSI.

- **P1. (PCSI Read Rule)**
  $\underline{\forall T_i, X_j \text{ such that } R_i(X_j) \in h :}$
  1- $W_j(X_j) \in h$ **and** $C_j \in h$;
  2- $commit(T_j) < snapshot(T_i)$;
  3- $\forall T_k$ such that $W_k(X_k), C_k \in h$:
  $[commit(T_k) < commit(T_j)$ **or** $snapshot(T_i) < commit(T_k)]$;
  4- $\forall T_k$ such that $W_k(X_k), C_k \in h$ **and** $T_i \sim T_k$:
  $[commit(T_k) < commit(T_j)$ **or** $start(T_i) < commit(T_k)]$.

- **P2. (PCSI Commit Rule)**
  $\underline{\forall T_i, T_j \text{ such that } C_i, C_j \in h :}$
  5- $\neg(T_j$ impacts $T_i)$.

Since GSI does not allow holes in the snapshot, it follows that T_i's snapshot includes the effects of all transactions that are in T_i's workflow and that have both executed and committed before T_i starts. This does not imply that transactions executing under PCSI will see only up-to-date snapshots of the database as in CSI (and pay the corresponding implementation cost): T_i's snapshot may not include the updates of some transactions that both have committed before T_i starts and are not in T_i's workflow.

6 Replicated databases

In this section we present a model for a distributed system with replicated databases. We show that read-only transactions under CSI must block in this model. Then, we instantiate PCSI's workflow relation and discuss two algorithms that implement PCSI. Both algorithms implement only PCSI; if serializability is needed, then condition D3 or S1 should be applied.

6.1 System Model and Definitions

We consider an asynchronous distributed system composed of a set of database sites $S_1, S_2, ..., S_i, ...$, which communicate through reliable message passing. We make no assumptions about the time it takes for sites to execute and for messages to be transmitted. Each database site has a full copy of the database. A transaction is (globally) committed as soon as it is locally committed at any site.

CSI requires each transaction to observe the latest snapshot. As stated in Proposition 3 below, in this model any implementation of CSI requires transactions to *block*. Transactions may block if there are periods of time during which sites cannot start new transactions. In a non-blocking implementation, transactions can start at any time. Blocking voids one of the main benefits of CSI and motivates using PCSI.

PROPOSITION 3. *There is no non-blocking implementation of CSI in an asynchronous system, even if database sites never fail.*

Although Proposition 3 holds in the absence of failures,[2]we assume the more general model, crash-recovery, where sites may fail independently by crashing, and subsequently recover an unbounded number of times. But we do not consider Byzantine failures.

We present two algorithms that implement PCSI. The first algorithm uses a central certifier to certify update transactions. The central certifier is a single point of failure, and this motivates the second algorithm which replicates the certifier using the state-machine approach [28]. The second algorithm employs an atomic broadcast abstraction [18] that is defined by the primitives broadcast(m) and deliver(m), which satisfy these properties: if a site delivers a message m, then all sites eventually deliver m *(agreement)*; if two sites, S_i and S_j, both deliver messages m_1 and m_2, then they do so in the same order *(total order)*; if a site broadcasts a message and does not crash, then all sites deliver the message *(validity)*; and a message is only delivered if it was previously broadcast by some site *(integrity)*.

In both algorihtms, every transaction executes its database operations at a single site—except for the commit operation of an update transaction as it requires remote communication to certify the transaction. Upon the commit of an update transaction, its writeset will be executed by other replicas, according to the

[2]If transactions are allowed to block and database sites do not fail, then there exists a simple algorithm which implements CSI in an asynchronous system [10]

algorithms.

We define the workflow relation $\sim$ as $\{(T_i, T_j)$: T_i and T_j execute in the same database site$\}$. To illustrate the need for the workflow relation consider the following scenario: A client first issues (and successfully commits) a transaction to database site S_i requesting to buy a book. Then, she issues a second transaction to the same site requesting a list of her orders. The above instantiation of the workflow relation guarantees that the second transaction will observe the effects of the first transaction; the list of orders will contain the order for that book.

We use version numbers to approximate global time. The database goes through a number of versions, each identified by a monotonically increasing version number. When a transaction starts, it obtains one of these versions as its initial snapshot. We use the notation $snapshotVer(T_i)$ for the version number of the initial snapshot of transaction T_i and $commitVer(T_i)$ for the version number of the database produced after the commit of transaction T_i. These two version numbers correspond to the global times $snapshot(T_i)$ and $commit(T_i)$, respectively.

6.2 Centralized Certification Algorithm

One site plays the role of master (or central certifier). The master certifies the commits of update transactions, and it contains the writesets of all committed update transactions. Replicas communicate only with the master using reliable message passing and do not communicate among each other.

The master maintains a persistent Log which is a set of ($version\ number$, $writeset$) tuples; Log is not lost in the event of a crash . Each tuple in Log contains the $writeset$ of the transaction that produced the database version with that particular $version\ number$. In addition, the master maintains the current version number V_{master}, initially set to zero.

Each replica S_i maintains its own version number V_i, which indicates the version number of its current database version, and which may be different from the current version at the master or at other replicas. For each transaction T that is active at a replica, the replica maintains the version number of its starting snapshot, which is denoted $snapshotVer(T)$.

At the start of transaction T, S_i provides the transaction with a snapshot equal to its current database version, and assigns V_i to $snapshotVer(T)$. Reads and writes execute locally against this snapshot, without any communication. When a read-only transaction completes, nothing further needs to be done. When an update transaction completes, it needs to be certified to commit successfully, as described in Algorithm 1.

When a database site recovers from a failure, it reads its last committed snapshot from the disk. Then, it sends the version of that snapshot to the certifier to obtain the most recent updates.

There are several methods to garbage collect the Log data structure by deleting old entries. For example

Algorithm 1 Centralized Certification Algorithm

```
When receiving $(V_j, wset_j)$ from $S_j$, $S_{master}$ executes:
  if $\exists(V, wset) \in Log \ni wset \cap wset_j \neq \emptyset \wedge V > V_j$ then
    send (abort,-,-) to $S_j$
  else
    $V_{master} \leftarrow V_{master} + 1$
    $Log \leftarrow Log \cup \{(V_{master}, wset_j)\}$
    $wsList \leftarrow \{(V, wset) : (V, wset) \in Log \wedge V > V_j\}$
    send (commit, $V_{master}, wsList$) to $S_j$

When ready to commit $T$, $S_i$ executes:
  send $(snapshotVer(T), writeset(T))$ to $S_{master}$
  wait until receive $(result, V_{new}, wsList)$ from $S_{master}$
  if $result$ = commit then
    db-apply-writesets($wsList$)
    $V_i \leftarrow V_{new}$
    db-commit($T$)
  else
    db-abort($T$)

When recovering from a failure $S_i$ executes:
  read the latest version $V_i$ from the database
  resume normal operation by executing the when clause above
    with $T_0$: $writeset(T_0) = \emptyset$, $snapshotVer(T_0) = V_i$
```

Algorithm 2 Distributed Certification Algorithm

```
When ready to commit $T$, $S_i$ executes:
  broadcast $(T, snapshotVer(T), writeset(T))$

When delivering $(T_j, V_j, wset_j)$, $S_i$ executes:
  if $\exists(V, wset) \in Log \ni wset \cap wset_j \neq \emptyset \wedge V > V_j$ then
    if $T_j$ executed at $S_i$ then db-abort($T_j$)
  else
    $V_i \leftarrow V_i + 1$
    $Log \leftarrow Log \cup \{(V_i, wset_j)\}$
    db-apply-writesets($wset_j$)
    if $T_j$ executed at $S_i$ then db-commit($T_j$)

When delivering ("$S_j$ recovering from $V_j$"), $S_i$ executes:
  $missing \leftarrow \{(V, wset) : (V, wset) \in Log \wedge V > V_j\}$
  send ($missing$) to $S_j$

When recovering from a failure $S_i$ executes:
  read the latest version $V_i$ from the database
  broadcast ("$S_i$ recovering from $V_i$")
  wait until deliver ("$S_i$ recovering from $V_i$")
  wait until receive ($missing$) from $S_j$
  db-apply-writesets($missing$)
  $V_i \leftarrow$ maximum $V$ in $missing$
  resume normal operation
```

to allow a replica to recover by exchanging information with the master and without communicateing with other replicas, only the versions older than the minimum V_i for all replicas can be safely deleted at the master. In any case when certifying a transaction, if any writeset needed to certify that transaction has been deleted from Log, the transaction is aborted to preserve the safety properties ensured by certification. This is a trade-off between the abort rate and the size of persistent storage.

6.3 Distributed Certification Algorithm

The distributed algorithm is a natural extension to the centralized one, using the state machine approach [28, 24]. Each replica both executes transactions and certifies update transactions. The algorithm uses an atomic broadcast abstraction [18] to deliver the writesets for certification to all replicas.

The key difference from the centralized implementation is that all replicas execute certification. Thus, all replicas now maintain the data structures that were maintained by the master in the centralized implementation. Transaction operations, except for commits, are handled as in the centralized algorithm. Finally, to simplify recovery, we assume that the Log data structure is kept in stable storage, that is, its contents are not lost in the event of a crash.

After executing an update transaction, the replica broadcasts a certification request, containing $snapshotVer(T)$ and $writeset(T)$ to all replicas. As depicted in Algorithm 2, when S_i delivers the certification request, it applies the commit rule P2 and checks that no committed transaction impacts T. If so, T

Symbol	Meaning
CW	length of the conflict window
$CW(T_i)$	conflict window of transaction T_i, which is $[snapshot(T_i), end(T_i)]$
D	age of the snapshot that the transaction receives when it begins execution
$DBSize$	database size (total number of data items)
L	number of seconds needed to execute a transaction on a single database
N	number of database sites
RR	request-reply delay: delay for replica to send message to master and receive response, it includes round-trip, data transfer and message processing
TPS	number of transactions per second
W	number of data items updated by each transaction

Table 1: Parameters of the analytical model.

is aborted; otherwise, T is committed.

When a database site crashes and later recovers, it needs to apply the effects of all messages which it did not deliver or which it delivered but did not process due to the crash. On recovery, the database site reads its latest snapshot from the database to determine V_i. Then it broadcasts a recovery message. The replica ignores all messages it delivers until it delivers that recovery message. Then the replica waits for a message with the updates it missed while it was down, applies them to the database, and continues processing delivered messages as usual.

7 Performance Analysis of PCSI

This section assesses analytically the relative performance of PCSI to CSI. More specifically, we show that under certain assumptions transaction abort rate is a linear function of both transaction length and the age of the snapshot that the transaction observes.

We use a simple model to estimate the abort rate of update transactions. This model is used to predict the probability of waits and deadlocks in centralized [17, pp. 428] and replicated databases [16]. We assume that the abort rate is small. This is a self–regulating assumption: If the abort rate is too high, then snapshot isolation algorithms are not suitable for the workload.

Initially, we consider a single-site database to contrast GSI to CSI. Then, we consider a replicated database over multiple sites to compare the abort rate of PCSI to that of CSI using an implementation based on centralized certification. Finally, using that implementation we compare the response times of read-only and update transactions in both PCSI and CSI.

7.1 Model

We consider a single database site that uses GSI under the following assumptions. The database has a fixed set of data items. The total number of these data items is $DBSize$. Here we consider only update

Metric	prefix-consistent snapshot isolation	conventional snapshot isolation	ratio of PCSI/CSI
System abort rate of update transactions	$((N * TPS_{update} * W)^2/DBsize)*$ $(D + L + 0.5 * RR)$, 127 trans/sec (%1 of update trans)	$((N * TPS_{update} * W)^2/DBsize)*$ $(L + RR)$, 58 trans/sec (%0.5 of update trans)	$(D + L + 0.5 * RR)/$ $(L + RR)$, 2.2
Response time of update transactions	$(L + RR)$, 250 ms	$(L + 2 * RR)$, 450 ms	$(L + RR)/$ $(L + 2 * RR)$, 0.55
Response time of read-only transactions	(L), 50 ms	$(L + RR)$, 250 ms	$(L)/(L + RR)$, 0.2

Table 2: Summary of performance metrics. with values: D = 400 ms, $DBSize$ = 10,000,000, L = 50 ms, N = 8 nodes, RR = 200 ms, TPS = 10,000 trans./sec (update trans. ratio %15, TPS_{update} = 1500), W=4.

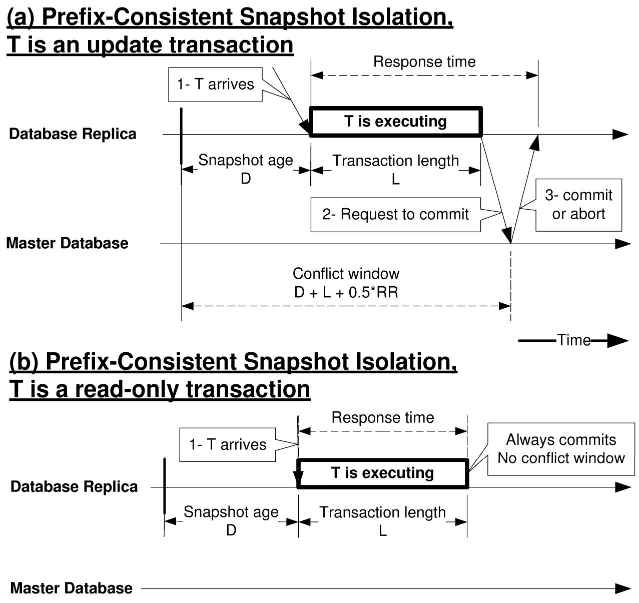

Figure 1: Transaction execution in a replicated database using PCSI with master database and centralized certification.

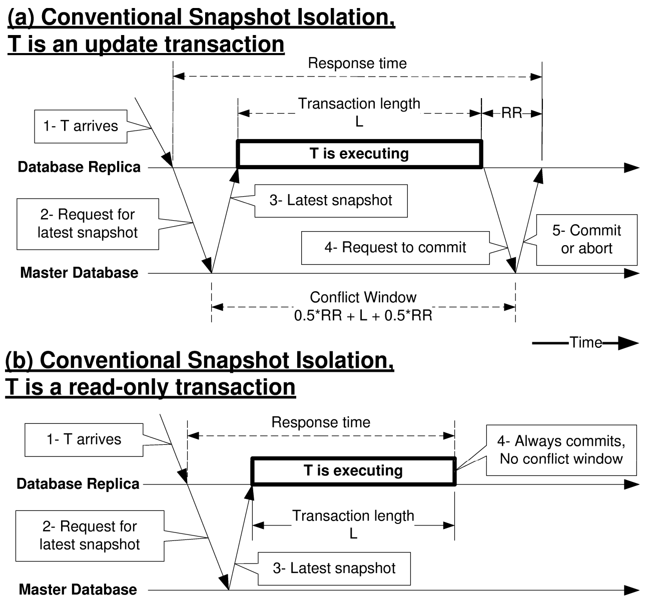
(a) Conventional Snapshot Isolation, T is an update transaction
Response time
Transaction length L
RR
1- T arrives
T is executing
Database Replica
3- Latest snapshot
2- Request for latest snapshot
4- Request to commit
5- Commit or abort
Master Database
Conflict Window 0.5*RR + L + 0.5*RR
Time
(b) Conventional Snapshot Isolation, T is a read-only transaction
Response time
1- T arrives
4- Always commits, No conflict window
T is executing
Database Replica
Transaction length L
2- Request for latest snapshot
3- Latest snapshot
Master Database

Figure 2: Transaction execution in a replicated database using CSI with master database and centralized certification.

transactions; read-only transactions are ignored. The database originates TPS_{update} update transactions per second. Each transaction updates W data items, takes L seconds to finish, and observes a snapshot of the database that is D seconds old. Table 1 lists the model parameters.

We define the time interval $CW(T_i)$, the *conflict window* of transaction T_i, such that $CW(T_i) = [snapshot(T_i), end(T_i)]$. The length of the conflict window is denoted simply as $CW = (L + D)$. According to the commit rule D2, while certifying T_i any transaction T_j will force T_i to abort if $commit(T_j) \in CW(T_i)$ and T_j wrote a data item that is also written by T_i.

In this model, we assume that access to data items is uniform - there is no hotspot - and that $DBSize >> (TPS_{update} * L * W)$, which means that at any instant the number of data items accessed (for update operations) is much smaller than the database size.

7.2 Centralized Database Abort Rate

In this subsection, we consider a centralized database using GSI to compute the abort rate of update transactions. The database *artificially* gives each update transaction a snapshot that is D seconds old, even though the most recent snapshot is available. First we compute the probability that a single transaction, T_i, has to abort. The number of transactions that commit in the conflict window $CW(T_i)$ is approximately $= (TPS_{update} * CW)$. Hence, the number of writes in that window $= W * (TPS_{update} * CW)$. The probability that a specific update (i.e., a write operation in T_i) conflicts with one of the writes in $CW(T_i)$ = (number of writes) / (database size) $= (W * TPS_{update} * CW)/DBsize$. If any such conflict occurs, transaction T_i must abort according to the commit rule D2. Since T_i has W of these updates, the probability that T_i has to abort is $W*$ (probability of a single conflict) $= (W^2 * TPS_{update} * CW)/DBSize$.

Then we compute the transaction abort rate at the database site. The rate of aborted transactions = (rate of transactions) * (probability that one transaction must abort) $= TPS_{update} * (W^2 * CW * TPS_{update})/DBSize$. Therefore, the abort rate at the database site $= (TPS^2_{update} * W^2/DBsize) * (CW)$. This rate is directly proportional to the length of the conflict window (CW =D + L).

Under CSI each transaction observes the latest snapshot of the database. Hence, $D = 0$ and $CW = L$. The node abort rate $= (TPS^2_{update} * W^2/DBsize) * (L)$.

For a single-site database the relative increase in node abort rate due to using GSI instead of CSI $= (1 + (D/L))$, which depends on D, the snapshot age: The older the snapshot, the higher the relative increase in abort rate.

7.3 Replicated Database Abort Rate

We now turn to replicated databases and estimate the abort rate of update transaction under PCSI and CSI. We consider a database replicated over N sites, where each site has a full database replica and originates TPS_{update} update transactions per second. We use a simple implementation based on a master database to perform centralized certification for both PCSI and CSI. Hence, our performance evaluation is valid only for this implementation. The request-reply delay, RR, is the time necessary for a replica to send a message to and receive the response from the master. RR includes the network round-trip delay, time necessary to transfer the data, and message processing delay. In wide-area networks, the round-trip delay constitutes a major part of RR.

From the previous subsection, the system abort rate is approximately $= ((N*TPS_{update})^2 * W^2/DBsize) * (CW)$. For the replicated case, the abort rate rises rapidly as the number of replicas increases because the abort rate is a quadratic function of N. It remains to estimate the length of the conflict window CW.

Under PCSI each update transaction observes the most recent snapshot available locally at the replica

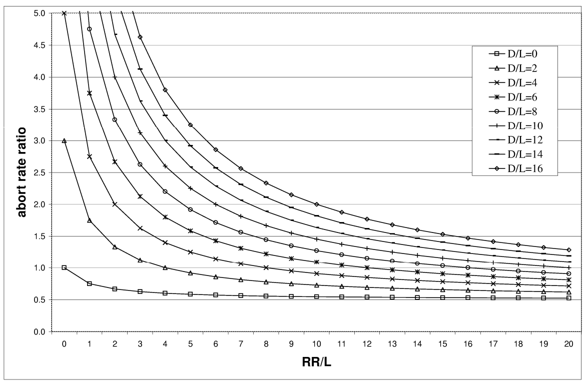

Figure 3: Relative update transactions abort ratio of PCSI to CSI on a replicated database with centralized certification, parameterized by D/L, (X-axis is RR/L).

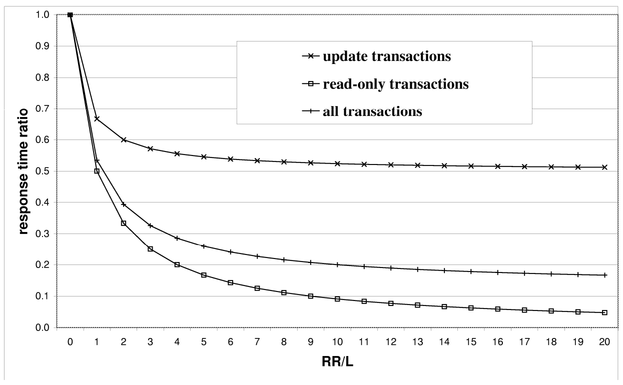

Figure 4: Relative response time ratio of PCSI to CSI on a replicated database with centralized certification, (X-axis is RR/L).

without any delay as depicted in Figure 1. At commit time, the database replica sends the transaction writeset for certification to the master database in order to check that there is no impacting transaction. The length of the conflict window is $CW = (D + L + 0.5 * RR)$.

Under CSI each update transaction must observe the latest snapshot in the system as it arrives at a database site. We assume that this can be done by sending a message to the master database and by receiving the reply as in Figure 2. This reply either indicates that the site already has the latest snapshot, or includes the missing updates, which the database installs to obtain the latest snapshot. Next the replica sends the writeset of the new transaction to the master database for certification to ensure that it is not impacted by any transition in the system. The length of the conflict window is $CW = (0.5 * RR + L + 0.5 * RR)$. The relative increase in node abort rate due to using PCSI instead of CSI is $= (D + L + 0.5 * RR)/(L + RR)$.

7.4 Replicated Database Response Time

Here we consider both read-only and update transactions and estimate their response times. We define the response time of a transaction to be the time taken from receiving the transaction at a database site until knowing the commit or abort status at the same database site. For PCSI in Figure 1, the response times of update and read-only transactions are $(L + RR)$ and (L), respectively. Similarly for CSI in Figure 2, each transaction has to incur a round-trip delay to check on the latest snapshot before it begins execution. The response times of update and read-only transactions are $(L + 2 * RR)$ and $(L + RR)$, respectively.

The response times of PCSI are lower, at the cost of **(1)** letting read-only transactions observe less recent data and **(2)** having a *potentially* higher abort rate for update transactions.

7.5 Numerical Evaluation

To get numeric estimates, we used "typical" numbers from a recent TPC-W disclosure report [30] and a request-response delay of 200 msec [5]. We used these numbers because we intend to use PCSI in wide-area replicated database and TPC-W is the industry standard benchmark for e-commerce systems [8, 31]. We, however, vary the ratio between some of these parameters in Figures 3 and 4.

Table 2 summarizes our findings. In that particular environment, the abort rate of update transactions in both PCSI and CSI is small, and PCSI's abort rate is twice CSI's abort rate. In addition, the response time of update transactions in PCSI is half of CSI's response time of update transactions. The response time of read-only transactions in PCSI is one fifth of CSI's response time of read-only transactions.

Figure 3 shows the ratio between the abort rate of update transactions in PCSI to that in CSI. The graph contains several curves corresponding to different values of D/L. Initially, PCSI aborts more transactions than CSI. Then as the ratio of RR/L increases along the X-axis, the trend reverses and CSI starts to abort more update transactions than PCSI. The ratio of abort rates reaches 0.5 as RR/L gets larger and larger for all curves. Although this is contrary to intuition to have CSI aborting more update transactions than PCSI, it can be explained by the fact that when the response-request delay is so large, waiting to get the latest snapshot increases the conflict window. This increase leads to the higher relative abort ratio of CSI to PCSI.

Figure 4 shows the ratio of response times of different transaction types in PCSI to those of CSI. There are three curves corresponding to read-only transactions, update transactions, and "all transactions" which corresponds to the ratio of average response times of all transactions in PCSI to those of CSI. Both PCSI and CSI give the same response times when RR=0 (centralized environment); however, as RR/L increases along the X-axis, PCSI gives substantially better response times for all transaction types. The ratio for update transactions approaches 0.5 quickly as RR/L increases. Under PCSI, read-only transactions may not observe the latest snapshot. RR is constrained by the speed of light and L is reduced with faster computers, so RR/L increases as technology advances favoring PCSI to CSI in both abort rates and response times.

7.6 Summary

For centralized databases, the abort rate under GSI is higher than that of CSI in which read-only transactions observe the latest snapshot. This makes CSI a clear winner in centralized systems.

However, for replicated databases, there is a tradeoff. Under PCSI, read-only transactions are not delayed, but they may not observe the latest snapshot. The update transactions' abort rate of PCSI may be higher or lower than that of CSI, depending on the particular implementation and system parameters. If the workload is dominated by read-only transactions, PCSI is more suitable than CSI.

8 Related Work

Despite the undebatable popularity and practical relevance of CSI, relatively few papers have discussed its properties. CSI was first introduced in 1995 [6, 20]. The authors show that CSI prevents many of the well--known concurrency control anomalies in [6], and that CSI is weaker than serializability: CSI allows some forms of constrain violation anomalies such as write-skew [6] and predicate-write-skew [13]. Schenkel et al. [27] have discussed using CSI in federated databases where global transactions access data that are distributed across multiple sites. Their protocols guarantee CSI at the federation level.

More research is needed for guaranteeing serializability both when using weaker isolation levels and when using application specific knowledge. Adya et al. [1] provided a theoretical foundation to formally specify practical isolation levels for commercial databases. Atluri et al. [4] studied the serializability of weaker isolation levels such as the ANSI SQL isolation levels for centralized databases. Shasha et al. [29] presented the conditions that allow a transaction to be chopped into smaller sub-transactions that release locks earlier than the original transaction under traditional locking policies. Kemme et al. [22] discussed how to implement different isolation levels (including serializability) in replicated databases using group communication primitives, in addition they implemented Postgres-R [21]. Due to the practical importance of snapshot isolation, two systems, Postgres-R(SI) [32] and Ganymed [25], have been built to replicate databases that use snapshot isolation.

Fekete et al. [12, 13, 11] have studied the conditions under which transactions executing on a centralized database with CSI produce serializable histories. They developed a syntactic condition similar to condition S1. They also show how database applications can be tested against the syntactic condition by building an extended Sibling–Conflict graph [29]. If certain cycles exist in this graph, the execution of the corresponding transaction may be non-serializable. They also showed how applications can be modified to satisfy the condition if necessary. They used *promotions* that upgrade read operations into identity write operations in order to break the corresponding cycles in the Sibling-Conflict graph. They also conducted an experimental evaluation showing that the overhead of promotions is small. Condition S1 is an extension to their syntactic condition, and it applies to GSI in centralized and distributed environments.

Dahlin et al. [14, 15] discussed application-specific data replication techniques for Web edge services, with the objectives of reducing response time, scaling up system performance, and enhancing its availability. This method is not transparent; application programmers have to specify a replication policy for each "data object". Also, there is no guarantee of serializability. Our methods have the same objectives. However, they are systematic and transparent to application programmers. Serializability of arbitrary transactions can be guaranteed by either rewriting the transactions in the application to satisfy condition S1, or by enforcing condition D3 at run-time.

9 Conclusions and Future Work

This research presents *generalized snapshot isolation (GSI)*, which is an extension of conventional snapshot isolation (CSI). GSI does not delay read-only transactions, but they may observe an old snapshot of the database. Serializability under GSI can be guaranteed. We showed a dynamic (D3) and a static (S1) condition that make transactions running under GSI produce serializable histories.

We presented *prefix-consistent snapshot isolation (PCSI)*, an instance of GSI that is suitable for multi-site replicated databases. PCSI uses the most recent snapshot currently available on a database site, such that each transaction sees the update of all transactions that have executed and committed at that site. The most important benefit of using PCSI is that it does not delay read-only transactions and executions can be made serializable.

We developed two implementations of PCSI: one uses centralized certification and the other uses distributed certification. We used an analytical model to compare the performance of PCSI to CSI, when using the centralized certification approach. The model shows that the abort rate of update transactions in PCSI and CSI depends on system parameters - PCSI may give a lower abort rate than CSI, and that the response times of both update and read-only transactions when using PCSI are smaller than those when using CSI.

We plan to use PCSI to use a geographically distributed network of proxies to replicate Web services. Web services are particularly suitable for PCSI: **1-** Their workload is dominated by read-only transactions. **2-** Most update transactions are short (with small writesets). **3-** The templates of all update transactions are known in advance, which allows using S1 to guarantee serializability.

References

[1] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. Generalized isolation level definitions. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 67–78, 2000.

[2] Akamai Technologies, Inc. EdgeSuite Architecture for Advanced E-Business. `http://www.akamai.com/en/html/services/edgesuite.html`.

[3] C. Amza, E. Cecchet, A. Chanda, Alan L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic Web site benchmarks. In *Proceedings of the 5th Workshop on Workload Characterization*, Austin, Texas, November 2002.

[4] Vijayalakshmi Atluri, Elisa Bertino, and Sushil Jajodia. A theoretical formulation for degrees of isolation in databases. *Elsevier Science*, 39 No.1, 1997.

[5] Gerco Ballintijn, Maarten van Steen, and Andrew S. Tanenbaum. Characterizing internet performance to support wide-area application development. *Operating Systems Review*, 34(4):41–47, 2000.

[6] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 1–10, May 1995.

[7] Philip Bernstein, Vassos Hadzilacos, and Nat Goodman. *Concurrency Control and Recovery in Database Systems*. Addison–Wesley, Reading, MA, 1987.

[8] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, and Willy Zwaenepoel. Performance comparison of middleware architectures for generating dynamic Web content. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, Rio de Janeiro, Brazil, June 2003.

[9] Data Concurrency and Consistency, Oracle8 Concepts, Release 8.0: Chapter 23. Oracle corporation. 1997.

[10] Sameh Elnikety, Fernando Pedone, and Willy Zwaenepoel. *Generalized Snapshot Isolation in Replicated Databases*. EPFL Technical Report IC/2004/21. `http://ic2.epfl.ch/labos/gsi.pdf`.

[11] Alan Fekete. Mixing snapshot isolation with two-phase locking: How can serializability be guaranteed. Paper submitted for publication in 2004.

[12] Alan Fekete. Serialisability and snapshot isolation. In *Proceedings of the Australian Database Conference*, pages 201–210, Auckland, New Zealand, January 1999.

[13] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. `http://www.cs.umb.edu/~isotest/summary.html`.

[14] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Improving availability and performance with application-specific data replication. *IEEE Transactions on Knowlege and Data Engineering*, 17(1):106–120, Jan 2005.

[15] Lei Gao, Mike Dahlin, Amol Nayate, Jiandan Zheng, and Arun Iyengar. Application specific data replication for edge services. In *Proceedings of the twelfth international conference on World Wide Web*, pages 449–460. ACM Press, 2003.

[16] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, June 1996.

[17] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, CA, 1993.

[18] Vassos Hadzilacos and Sam Toueg. *Fault-Tolerant Broadcasts and Related Problems*. Distributed Systems (2nd Ed.), ACM Press / Addison-Wesley Publishing Co., 1993.

[19] InterBase Documentation. Borland Software Corporation. 2004.

[20] K. Jacobs. Concurrency control, transaction isolation and serializability in SQL92 and Oracle7. Technical report, Oracle Corporation, Redwoord City, CA, July 1995. White paper A33745.

[21] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB 2000), Cairo, Egypt, September 2000.*

[22] Bettina Kemme and Gustavo Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings 18th International Conference on Distributed Computing Systems (ICDCS), Amsterdam, The Netherlands, May 1998.*

[23] Christos Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[24] Fernando Pedone, Rachid Guerraoui, and Andre Schiper. The database state machine approach. *Distributed and Parallel Databases*, (14):71–98, 2003.

[25] Christian Plattner and Gustavo Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware 2004)*, Toronto, Canada, October 2004.

[26] PostgreSQL 7.4 Documentation. PostgreSQL Global Development Group. 2004.

[27] Ralf Schenkel, Gerhard Weikum, Norbert Weißenberg, and Xuequn Wu. Federated transaction management with snapshot isolation. *Lecture Notes in Computer Science*, 1773:1–25, January 2000.

[28] Fred Schneider. *Replication Management using the State-Machine Approach*. Distributed Systems (2nd Ed.), ACM Press / Addison-Wesley Publishing Co., 1993.

[29] Dennis Shasha, François Llirbat, Eric Simon, and Patrick Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems*, 20(3):325–363, 1995.

[30] The Transaction Processing Council (TPC). Full disclosure report for Dell PowerEdge 6650 1.6GHz w/PowerEdge 1650 1.4GHz. `http://www.tpc.org/results/FDR/tpcw/PE6650_10K_TPCW_FDR_020822.pdf`.

[31] The Transaction Processing Council (TPC). The TPC-C and TPC-W Benchmarks. `http://www.tpc.org/`.

[32] Shuqing Wu and Bettina Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *International Conference of Data Engineering*, 2005.

[33] Yukon Release Microsoft SQL Server. Microsoft Corporation. 2005.