View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

Generalized Snapshot Isolation and a Prefix-Consistent Implementation

Sameh Elnikety, Fernando Pedone, and Willy Zwaenepoel
School of Computer and Communication Sciences
EPFL,
Lausanne, Switzerland

EPFL Techical Report IC/2004/21
1 March 2004

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, 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. As an instance of generalized snapshot isolation, it inherits all of its properties. In addition, read-only transactions never block, and consecutive transactions submitted in a single workflow on a particular replica observe the updates of their predecessors in the workflow.

We present two implementation strategies of prefix-consistent snapshot isolation. We conclude with an analytical performance model of one of the implementations, bringing out the benefits, in particular reduced latency for read-only transactions, and showing that the potential downsides, in particular the change in abort rate of update transactions, are limited.

1. INTRODUCTION

1.1 Motivation

There is increasing interest in replicating the generation of dynamic Web content [3, 11]. The user's request is executed on a "nearby" replica, thereby avoiding long roundtrip delays and distributing the load over the replicas. In a centralized dynamic content Web site, the dynamic data is most often stored in a database. Therefore, one of the key problems in replicating dynamic 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 the transaction T starts. At commit, the database checks that the writeset of the transaction T does not intersect with the writesets of any other 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 and other database vendors use it [1, 6, 10, 15]. More fundamentally, snapshot isolation never requires read-only transactions to be blocked or aborted. For workloads with a large fraction of read-only transactions (such as those resulting from dynamic Web content generation) this advantage is significant. Snapshot isolation provides a weaker form of consistency than serializability, but one that programmers understand and are capable of dealing with. Moreover, Fekete et al. [9, 10] have recently demonstrated that under certain conditions on the workload transactions executing on a database with snapshot isolation produce serializable histories.

1.2 Generalized Snapshot Isolation

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 of the same properties 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, although they 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 the 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 the execution).

1.3 Prefix-Consistent Snapshot Isolation

In a replicated setting, an interesting positioning in this spectrum is for a transaction to take as its initial snapshot the latest snapshot available locally on its replica. We refer to this method as *prefix--consistent snapshot isolation.* As an instance of generalized snapshot isolation, prefix-consistent snapshot replication maintains the desirable properties of generalized snapshot isolation. Read-only transactions never block or abort. Moreover, transactions observe at least all the writes that committed before it on the local replica. This property is important in a workflow of transactions, in which a user submits a sequence of transactions to a replica. We present two implementation strategies for prefix-consistent snapshot isolation, and we analyze the performance of one implementation analytically.

1.4 Contributions and Paper Outline

The contributions of this paper are as follows:

- introducing generalized snapshot isolation,
- establishing two sufficient conditions (one statically checkable and one dynamic) that guarantee serializable execution for transactions running under generalized snapshot isolation,
- introducing prefix-consistent snapshot isolation and two implementation strategies, and
- analytically characterizing the relative performance of prefix-consistent snapshot isolation to conventional snapshot isolation.

The reminder of this paper is organized as follows: Section 2 presents the database model and necessary definitions. Section 3 presents generalized snapshot isolation, and Section 4 relates generalized snapshot isolation to serializability. Section 5 presents prefix-consistent snapshot isolation. Section 6 presents two implementations of prefix-consistent snapshot isolation, one using centralized and one using distributed certification. Section 7 compares analytically the performance of prefix-consistent snapshot isolation to conventional snapshot isolation. Section 8 discusses related work. Finally, section 9 presents our conclusions and future research directions.

2. MODEL AND DEFINITIONS

2.1 Database Model

We assume that a *database* is a collection of uniquely identified data items. Several versions of the same data item may co-exist simultaneously in the database. There is a total order among the versions of each data item. A *snapshot* of the database at time t is the committed state of the database which includes only the latest committed versions of all data items at time t. A *transaction* T_i is a sequence of read and write operations on data items, followed by either a commit or an abort operation, and executes on a single machine. We denote T_i's write operation on data item X by $W_i(X_i)$. If T_i executes $W_i(X_i)$ and commits, a new version of X, denoted by X_i, is added to the database. Moreover, we denote T_i's read operation on data item X_j by $R_i(X_j)$. Finally, T_i's commit or abort is denoted by C_i or A_i, respectively.

A transaction is *read-only* if it contains no write operations, and is *update* otherwise. The readset of transaction T_i, denoted $readset(T_i)$, is the set of data items that T_i reads. 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 with ordering relation $\prec$ 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 Overlapping Transactions

To simplify the definitions presented in the paper, we assume that every database operation in a transaction T_i (i.e., $R_i(X_j)$, $W_i(X_i)$, C_i, and A_i) happens at a distinct time. We also assume the existence of a discrete global clock to define the ordering relation $<$ between events. This clock is a fictional device and access to it is not necessary to implement generalized snapshot isolation. We show in Section 6 that in a distributed system in which the database is replicated over several sites, global time is not needed to implement prefix-consistent snapshot isolation.

In generalized snapshot isolation, each transaction observes a snapshot of the database that is taken at some time in the past. If transaction T_i sees a snapshot of the database taken at time t, this snapshot includes the updates of all transactions that have committed before t. 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 starting time of T_i.
- $commit(T_i)$: the time when T_i is committed.
- $abort(T_i)$: the time when T_i is aborted.
- $end(T_i)$: the time when T_i is committed or aborted.

Notice that $snapshot(T_i) \leq start(T_i) < end(T_i)$. Next, we define the relation *overlap* for update transactions as follows:

- T_i *overlaps* with T_j **iff** $writeset(T_i) \neq \emptyset$ and $writeset(T_j) \neq \emptyset$, **and** $snapshot(T_i) < commit(T_j) < end(T_i)$

From the above definition, an update transaction T_i overlaps with committed update transactions only. T_i does not overlap with read-only transactions and uncommitted transactions. T_i's overlapping transactions are the set $\{T_j : T_i \ overlaps \ with \ T_j\}$. As presented next, T_i does not see the updates of its overlapping transactions.

3. GENERALIZED SNAPSHOT ISOLATION (GSI)

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

- **G1. (GSI Read Rule)**
  $\forall T_i, X_i$ such that $R_i(X_i) \in h$:
  1- $\exists W_i(X_i) \in h$ such that $W_i(X_i) \prec R_i(X_i)$ in h.
  AND
  $\forall T_i, X_j$ such that $R_i(X_j) \in h$:
  2- $\nexists W_i(X_i) : W_i(X_i) \prec R_i(X_j)$ in h;
  3- $commit(T_j) < snapshot(T_i)$; **and**
  4- $\forall T_k$ such that $W_k(X_k), C_k \in h$: $(commit(T_k) < commit(T_j)$ **or** $snapshot(T_i) < commit(T_k))$.

- **G2. (GSI Commit Rule)**
  $\forall T_i, T_j$ such that $C_i, C_j \in h$ **and** $snapshot(T_i) < commit(T_j) < commit(T_i)$:
  5- $writeset(T_j) \cap writeset(T_i) = \emptyset$.

The read rule ensures that each transaction initially observes a committed snapshot of the database. This snapshot could be any snapshot that has been taken before the transaction starts. Rule G1 has four parts. Part one and two force T_i to see its own updates; if T_i issues $W_i(X_i)$, then the next read operation for data item X must return X_i, i.e., $R_i(X_i)$. The third part allows T_i to read only the data items written by transactions that committed before T_i's snapshot. Updates of T_i's overlapping transactions are invisible to T_i because they commit after $snapshot(T_i)$. The fourth part 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 that updated X before t.

The commit rule prevents an update transaction T_i from committing if any overlapping transaction have already written into T_i's writeset. This rule gives the "first-committer-wins behavior": If two active transactions update the same data item, the transaction that commits first succeeds at updating that item; whereas the other transaction has to abort.

Conventional snapshot isolation (CSI) is a special case of GSI where each transaction sees the *last* snapshot, with regard to its starting time, i.e., $snapshot(T_i) = start(T_i)$. This is referred to in literature as simply "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 minimizes the number of overlapping transactions. 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

Serializability is the basic correctness criteria in much work on databases [18, 7]. GSI provides a weaker form of consistency than serializability. Both GSI and CSI may produce non-serializable histories. Here is an example the write-skew anomaly [6]. Assume that X and Y are the balances of two bank accounts for the same person and 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 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. h_1 is non-serializable.

The next subsections present two conditions, C1 and C2, which make generalized snapshot isolation produce only serializable histories.

4.1 Static Serializability Condition (C1)

Transactions that satisfy condition C1 and run under GSI produce serializable histories. Condition C1 is sufficient and is not necessary. 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 below. C1 only leaves the writesets to be checked dynamically which is already done in GSI.

GSI produces serializable histories if the following holds:

- **C1. (Serializability)**
  $\forall T_i, T_j : T_i$ overlaps with T_j:

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

Proposition 1 states that if condition C1 is satisfied then GSI produces only serializable executions.

PROPOSITION 1. *Generalized snapshot isolation is serializable under C1.*

PROOF: The proof shows that every history h that satisfies rules G1 and G2 and condition C1 has an acyclic multi-version serialization graph (MVSG)[7]. And if $MVSG(h)$ is acyclic, then h is view equivalent to some serial execution of the same transactions [7].

MVSG is a directed graph, in which the nodes represent committed transactions. There are three types of directed edges in MVSG: (a) read-from edges, (b) version-order edges type I, and (c) version-order edges type II. These types are described below. We initially consider update transactions only, i.e. edges that connect two update transactions. Then we consider read-only transactions.

Update transactions. From the algorithm, the commit order of transactions induces a version order on every data item, that is, if $\ll$ is an order relation on the versions, and both T_i and T_j update X, we have $commit(T_i) < commit(T_j) \Leftrightarrow X_i \ll X_j$. To show that $MVSG(h)$ has no cycles, we prove that for every edge $T_i \rightarrow T_j$ in $MVSG(h)$, it follows that $commit(T_i) < commit(T_j)$. The proof continues by considering each kind of edge in $MVSG(h)$.

1. (Read-from edge.) If T_j reads data item X_i from T_i (i.e., $R_j(X_i)$), then $T_i \rightarrow T_j \in MVSG(h)$. We have to show that $commit(T_i) < commit(T_j)$. From rule G1, T_j only reads a committed state of the database, and therefore, $commit(T_i) < snapshot(T_j)$. Since $snapshot(T_j) < start(T_j) < commit(T_j)$, it follows that $commit(T_i) < commit(T_j)$.

2. (Version-order edge type I.) If both T_i and T_j write X such that $X_i \ll X_j$, then $T_i \rightarrow T_j \in MVSG(h)$. Since the commit order induces the version order, we have that $X_i \ll X_j \Leftrightarrow commit(T_i) < commit(T_j)$.

3. (Version-order edge type II.) If T_i reads X from T_k, and both T_k and T_j write X such that $X_k \ll X_j$, then $T_i \rightarrow T_j \in MVSG(h)$. From G2, since both T_k and T_j update X, it must be that T_j's snapshot was taken after T_k committed (otherwise T_j would abort). From G1, since T_i reads X from T_k, T_i must have a snapshot that was taken after T_k committed: $commit(T_k) < snapshot(T_i)$.

   There are only two possibilities: either (a) $commit(T_i) < commit(T_j)$ or (b) $commit(T_j) < commit(T_i)$. We show that (b) is impossible. For a contradiction, assume that (b) holds. It must be that $snapshot(T_i) < commit(T_j)$, otherwise T_i would read X from T_j and not from T_k (from G1). Thus, we have $snapshot(T_i) < commit(T_j) < commit(T_i)$, and so, T_i must overlap with T_j.

   Moreover, from C1, $readset(T_i) \cap writeset(T_j) = \emptyset$ and $writeset(T_i) \cap readset(T_j) = \emptyset$. But since T_i reads X and T_j writes X, we reach a contradiction that concludes the proof for update transactions.

Read-only transactions. Let T_q be a read-only transaction in h. Since T_q does not update any data item, any edge involving T_q in $MVSG(h)$ is of the kind (a) $T_j \rightarrow T_q$ or (b) $T_q \rightarrow T_i$, where T_i and T_j are update transactions. We initially show that the former implies $commit(T_j) < snapshot(T_q)$ and the latter implies $snapshot(T_q) < commit(T_i)$.

1. (a) Read-from edge: $T_j \rightarrow T_q$. Since $T_j \rightarrow T_q \in MVSG(h)$, T_q must read some data item updated by T_j. Since snapshots only contain committed data, it follows that $commit(T_j) < snapshot(T_q)$.

2. (b) Version-order edge type II: $T_q \rightarrow T_i$. For $T_q \rightarrow T_i$ to be in $MVSG(h)$, there must exist some transaction T_k such that both T_i and T_k update the same data item X, $X_k << X_i$, and T_q reads X from T_k. It follows from rule G1 (part 2), that either $commit(T_i) < commit(T_k)$ or $snapshot(T_q) < commit(T_i)$. From the first part of the proof (for update transactions) and the fact that $X_k << X_i$, it must be that $commit(T_k) < commit(T_i)$, and we conclude that *snapshot*$(T_q) <$ *commit*(T_i).

The proof continues by contradiction: assume T_q is involved in a cycle c in $MVSG(h)$. Then, for each edge $T_a \rightarrow T_b \in c$, $commit(T_a) < commit(T_b)$ if both T_a and T_b are update transactions (from the first part of the proof), $commit(T_a) <$ *snapshot*(T_b) if T_b is a read-only transaction (from case (a) above), and $snapshot(T_a) < commit(T_b)$ if T_a is a read-only transaction (from case (b) above). Thus, we conclude that if c exists, it follows that $snapshot(T_q) < snapshot(T_q)$, a contradiction that concludes the proof. □

Condition C1 is only sufficient; it is not necessary for GSI histories to be serializable. For example, let history $h_2 = W_i(X_i), C_i, W_j(X_j), R_k(X_i), W_k(Y_k), C_k, C_j$, that is, T_j overlaps with T_k. History h_2 is serializable even though *write-set*$(T_j) \cap readset(T_k) \neq \emptyset$. Later, we present a weaker serializability condition which allows h_2.

In practice, condition C1 requires all possible update transactions to be known. That is usually the case in many database applications where the database transactions are hidden from users 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 the condition. Fortunately, in many situations ad-hoc update transactions are forbidden because they need to follow the workflow of the business logic. Nonetheless, ad-hoc read-only transactions are allowed as they do not affect any serializability result in this subsection.

To check whether condition C1 holds for a set of update transactions (or transaction templates), each pair that can potentially overlap must satisfy the condition. This test must be applied conservatively for each and every pair, unless the two transactions in the pair cannot overlap in the specific application or database implementation.

If a pair of update transactions does not satisfy C1, 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 to include an identity write on item X, i.e., change $R(X)$ into $R(X), W(X)$. This change makes their write-sets intersect; therefore, they satisfy C1.

The process of checking the transactions and altering them if necessary can be automated [9, 10]. Moreover, in some applications, all transactions already satisfy C1. For example, both the TPC-C and TPC-W benchmarks [24]; which are the industry–standard for evaluating on–line transaction processing systems and e–commerce systems, respectively; satisfy condition C1. TPC-C and TPC-W produces serializable histories under GSI without any modification.

4.2 GSI into Perspective

By definition, any history that satisfies CSI also satisfies GSI; the converse, however, is not true: Let history $h_3 = W_i(X_i), C_i, W_j(X_j), C_j, R_k(X_i), W_k(Y_k), C_k$. Clearly, h_3 is allowed by GSI (e.g., it suffices for $snapshot(T_k) = commit(T_i)$) but it is not allowed by CSI.

We define an *update history* to be a history that includes only update transactions and no read-only transactions. Proposition 2 shows that any update history that satisfies GSI condition C1 must satisfy CSI. To see why this holds intuitively, consider transactions T_i and T_j such that T_i updates X and T_j reads X. Under GSI, if T_j starts after T_i commits, T_j should get a snapshot with T_i's update. This is not the case under GSI, as long as T_j overlaps with T_i. However, condition C1 states that since $X \in writeset(T_i) \cap readset(T_j)$, T_j cannot overlap with T_i, and so, it will also see a snapshot that includes T_i's update.

PROPOSITION 2. *Any update history h that satisfies GSI's rules G1 and G2 and condition C1 also satisfies CSI.*

The proof of Proposition 2 is in Section A of the Appendix.

If GSI and condition C1 are enforced, the execution of an update transaction that succeeds in committing is view-equivalent to the execution of the same transaction starting from the latest snapshot.

There are CSI histories that do not satisfy GSI's rules G1 and G2 and condition C1. Consider for example history $h_4 = W_i(X_i), C_i, W_j(X_j), R_k(X_i), C_j, W_k(Y_k), C_k$. Although h_4 satisfies CSI, it violates C1.

By definition, if an update history satisfies CSI and condition C1, it also satisfies GSI and condition C1. The converse also holds: Every update history h that satisfies GSI and condition C1, also satisfies CSI and condition C1.

4.3 Dynamic Serializability Condition (C2)

History $h_2 = W_i(X_i), C_i, W_j(X_j), R_k(X_i), W_k(Y_k), C_k, C_j$ is serializable because T_k could be ordered before T_j in some equivalent serial order. Based on this observation, we define condition C2 as follows:

- **C2. (Serializability)**
  $\forall T_i, T_j : T_i$ overlaps with T_j:

$$\left[\begin{array}{c} readset(T_i) \cap writeset(T_j) = \emptyset \\ \wedge \\ \left(\begin{array}{c} writeset(T_i) \cap readset(T_j) = \emptyset \\ \vee \\ commit(T_j) < commit(T_i) \end{array}\right) \end{array}\right]$$

PROPOSITION 3. *Generalized snapshot isolation is serializable under C2.*

The proof of Proposition 3 is in Section B of the Appendix.

C2 is a sufficient condition and is not necessary as described below. Also, C2 is a dynamic condition: It has no static checks. When committing an update transaction T_i, GSI requires that $writeset(T_i)$ does not intersect with the writesets of T_i's overlapping transactions. In addition, C2 requires that $readset(T_i)$ does not intersect with the writesets of T_i's overlapping transactions. Therefore, to commit T_i, both its readset and writeset must be available for the dynamic checks. However, the readsets of already committed transactions need not be available. Notice that C2 may abort a transaction even though that transaction can be serialized.

Condition C2 takes the commit order of transactions into account. To see why this is needed, assume that T_j is allowed to commit before T_k commits in h_2, and consider that T_k updates some data item Y. Then we could have some transaction T_q that gets a snapshot including T_j's updates but not T_k's. Thus, T_q perceives T_j before T_k in any equivalent serial history, but for T_k, T_j should come next to it, and so, no equivalent serial history exists.

GSI with condition C2 allows more histories than GSI with C1 (e.g., h_2) but it is still not necessary. Consider history $h_5 = W_i(X_i), C_i, W_j(X_j), R_k(X_i), W_k(Y_k), C_j, C_k$, which is h_2 after switching the commit order of T_j and T_k. History h_5 is still serializable but does not satisfy C2.

5. PREFIX-CONSISTENT SNAPSHOT ISOLATION (PCSI)

GSI's rules G1 and G2 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 state of the database until the latest snapshot, as in CSI. However in general, the older the snapshot, the higher the probability that an update transaction will have to abort because it will probably overlap with more update transactions.

Prefix-consistent snapshot isolation (PCSI) is designed for replicated databases. Transactions may see an older snapshot of the database. They must, however, see the updates of all transactions that have committed on the same replica. For example, if transaction T_i starts at some site S, then site S has to make sure that the snapshot received by T_i includes the updates of all transactions that have executed at site S and have already committed. We add an extra part (part 5) to GSI's read rule to have the following two PCSI rules:

- **P1. (PCSI Read Rule)**
  $\forall T_i, X_i$ such that $R_i(X_i) \in h$:
  1- $\exists W_i(X_i) \in h$ such that $W_i(X_i) \prec R_i(X_i)$ in h.
  AND
  $\forall T_i, X_j$ such that $R_i(X_j) \in h$:
  2- $\nexists W_i(X_i) : W_i(X_i) \prec R_i(X_j)$ in h;
  3- $commit(T_j) < snapshot(T_i)$;
  4- $\forall T_k$ such that $W_k(X_k), C_k \in h$: $(commit(T_k) < commit(T_j)$ **or** $snapshot(T_i) < commit(T_k))$; **and**
  5- $\forall T_k$ such that $W_k(X_k), C_k \in h$ **and** both T_i and T_k execute on the same database site : $(commit(T_k) < commit(T_j)$ **or** $start(T_i) < commit(T_k))$.

- **P2. (PCSI Commit Rule)**

$\forall T_i, T_j$ such that $C_i, C_j \in h$ **and**
$snapshot(T_i) < commit(T_j) < commit(T_i)$:
6- $writeset(T_j) \cap writeset(T_i) = \emptyset$.

Since GSI does not allow holes in the snapshot, it follows that the effects of all transactions that have both executed in any site and committed before the time of T_i's snapshot should also be included in T_i's snapshot. That is, the writesets of all update transactions that have committed before $snapshot(T_i)$ must be available at site S. This does not imply that transactions executing under PCSI will see only up-to-date snapshots of the database (and pay the corresponding implementation cost). When transaction T_i starts, its site S has all updates resulting from local committed transactions, but it may not have the updates of remote transactions that committed after S's last transaction committed. In that case T_i does not receive the latest snapshot of the database.

6. IMPLEMENTATION OF PCSI

We present two implementations of prefix-consistent snapshot isolation: the first uses centralized certification and the second uses distributed certification. Both of them implement only PCSI. If serializability is needed, then condition C1 or C2 can be applied. In both implementations, we assume that each database site has a full database replica, and may fail independently by crashing. However, each database eventually recovers after a crash, and there is a time after which the database never crashes. We use version numbers to approximate global time. In other words, 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 version 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.

6.1 Centralized Certification

The system consists of a master database (central certifier) and a number of database replicas. The master certifies the commits of update transactions, and it contains the latest version of data. Replicas execute transactions on behalf of users, and may not have the most up-to-date version of data. Replicas communicate only with the master and do not communicate among each other.

6.1.1 Data structures

The master maintains an ordered sequence, denoted SEQ, of $\{version\ number, writeset\}$ records. Each record 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}.

Each replica maintains its own version number $V_{replica}$, which indicates the version number of its current database version, and which may be different from the current version at the central site or at other replicas. For each transaction T_i that is active at a replica, the replica maintains the version number of its starting snapshot $snapshotVer(T_i)$.

6.1.2 Transaction execution

At the start of a transaction T_i, the replica provides the transaction with a snapshot equal to its current database version, and assigns $V_{replica}$ to $snapshotVer(T_i)$. Reads and writes execute locally against this snapshot, without any communication. T_i's snapshot can be updated only by T_i's writes such that T_i observes its own updates. When a read-only transaction completes, nothing further needs to be done. When an update transaction completes, it needs to be certified before it can be allowed to commit.

6.1.3 Certification of update transactions

The replica sends a message to the master, containing the current value of $V_{replica}$, and both $snapshotVer(T_i)$ and $writeset(T_i)$ for the transaction T_i to be certified. Using SEQ, the master determines if $writeset(T_i)$ intersects with any of the writesets of transactions with version numbers higher than $snapshotVer(T_i)$. If so, the transaction needs to be aborted. The master communicates this outcome to the replica, and the replica aborts the transaction. If on the contrary all intersections are empty, the transaction can be committed. The master increments V_{master}, applies the $writeset(T_i)$ to its database, and appends a new record for this transaction to the tail of SEQ. It sends a message to the replica, including the commit decision, the value of V_{master}, and an ordered sequence of the writesets of any transactions with version numbers between $V_{replica}$ (as received in the message from the replica) and V_{master}.

Upon receipt of this message, the replica applies (in order) the writesets it receives in the message, commits the transaction (applying its writeset as well), and sets $V_{replica}$ to V_{master}.

6.1.4 Proof of correctness

See section C.1 in the Appendix.

6.1.5 Refinements

The data in the SEQ data structure may already be contained in the replica's redo log. Therefore, the replica may implement a local certification step, checking if the transaction's writesets intersects with the writesets of previous transactions that it knows as a result of responses to earlier certification requests. If so, the replicas need to maintain a SEQ data structure similar to the master's.

To prevent replicas from falling too far behind, especially replicas that execute only or mostly read-only transactions, the writesets of committed transactions may be communicated asynchronously from the master to a replica. For instance, a replica can periodically send its $V_{replica}$ and request all missing writesets.

The SEQ data structure can be garbage collected by deleting old entries. When certifying a transaction, if its starting snapshot version is smaller than version number of the head of SEQ, 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.

Although the master is a central point of failure, it can use standard techniques such as primary-backup replication to improve its availability and scalability. In addition, it can use distributed certification, as discussed next.

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.

6.2 Distributed Certification

The system consists of a fixed number of replicas. Each replica can both execute transactions and certify update transactions. There is no master database (central certifier). We assume the existence of an atomic broadcast facility [14] to deliver the writesets for certification to all replicas. We use the state machine approach [21, 19] where certification contains only deterministic operations.

The key difference to the centralized implementation is that all replicas execute certification. Therefore, all replicas now maintain the data structures that were maintained by the master in the centralized implementation. Transaction start, reads and writes, and read-only transactions proceed as before.

6.2.1 Certification of update transactions

After executing an update transaction, the replica the sends a certification request, which contains $snapshotVer(T_i)$ and $writeset(T_i)$, by atomic broadcast to all replicas. When a replica receives the certification request, it applies the commit rule P2 and checks $writeset(T_i)$ against the writesets of T_i's overlapping transactions. That is, using SEQ, the replica determines if $writeset(T_i)$ intersects with any of the writesets of transactions with version numbers higher than $snapshotVer(T_i)$. If so, T_i is aborted; otherwise, all intersections are empty and T_i is committed: $V_{replica}$ is incremented, $\{V_{replica}, writeset(T_i)\}$ is appended to SEQ, and $writeset(T_i)$ is installed into the local data.

6.2.2 Proof of correctness

See section C.2 in the Appendix.

6.2.3 Refinements

Many of the same refinements can be used as in the centralized implementation. It is, however, no longer necessary to asynchronously send out writesets, because all replicas see all writesets as part of the certification process. Care has to be taken that the garbage collection of SEQ on different replicas does not make the algorithm nondeterministic. This can be done, for instance, by making sure that all replicas pick the same value of version number, below which all records in SEQ are going to be deleted.

7. PERFORMANCE ANALYSIS OF PCSI

This section assesses analytically the relative performance of prefix-consistent snapshot isolation to conventional snapshot isolation. 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 for update transactions. This model is used to predict the probability of waits and deadlocks in centralized [13, pp. 428] and replicated databases [12]. 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 generalized snapshot isolation to conventional snapshot isolation. Then, we consider a replicated database over multiple sites to compare the abort rate of PCSI to 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 generalized snapshot isolation under the following assumptions. The database has a fixed set of data items. The total number of these data items is $DBSize$. We consider only update transactions, and 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 G2, any transaction T_j will force T_i to abort if $commit(T_j) \in CW(T_i)$ and T_j writes a data item that is also written by T_i.

In this model, access to the data items is uniform without any hotspot. Furthermore, the model assumes the following: $DBSize >> (TPS_{update} * L * W)$, 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 generalized snapshot isolation 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 commits 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 G2. Since T_i has W of these updates, the probability that T_i has to abort is $W*$ (probability of a single conflict) $= (W^2 * TPS_{update} * CW)/DBSize$.

Metric	prefix-consistent snapshot isolation	conventional snapshot isolation	ratio of PCSI/CSI
System abort rate of update transactions	$((N * TPS_{update} * W)^2/DBsize)*$ $(D + L + 0.5 * RR)$, 126.72 transactions per second (i.e., %1.06 of all update transactions)	$((N * TPS_{update} * W)^2/DBsize)*$ $(L + RR)$, 57.6 transactions per second (i.e., %0.48 of all update transactions)	$(D + L + 0.5 * RR)/$ $(L + RR)$, 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. Numeric values: D = 400 ms, $DBSize$ = 10,000,000, L = 50 ms, N = 8 nodes, RR = 200 ms, TPS = 10,000 transaction/sec (update transactions ratio %15, TPS_{update} = 1500), and W=4.

Second 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 conventional snapshot isolation, 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 *(artificial)* generalized snapshot isolation instead of conventional snapshot isolation $= (1 + (D/L))$. This is a function of D, the snapshot age. The older the snapshot, the higher the relative increase in abort rate.

7.3 Replicated Database Abort Rate

In this subsection we consider a replicated database over N sites, and estimate the abort rate for update transaction under prefix--consistent and conventional snapshot isolation. We assume that there are N database sites where each has a full database replica and originates TPS_{update} update transactions per second. We use a simple distributed implementation based on a master database with centralized certification for both prefix-consistent and conventional snapshot isolation. Hence, our performance evaluation is valid only for this implementation. The request-reply delay is RR. It is the time necessary for a replica to send a message to the master and for the replica to receive the response. RR includes the network round-trip delay, time necessary to transfer the data, and message processing delay. In a wide-area network, the round-trip delay constitutes the 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 with the number of replicas as it is a quadratic function of N. It remains to estimate the length of the conflict window, CW.

Under prefix-consistent snapshot isolation, each update transaction observes the most recent snapshot available locally at the replica 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 system-wide conflict among the overlapping writesets. The length of the conflict window $CW = (D + L + 0.5 * RR)$.

Under conventional snapshot isolation, each update transaction must observe the latest snapshot in the system as it arrives at a database node. 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 the reply 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 there is no conflict among the concurrent transitions in the system. The length of the conflict window $CW = (0.5 * RR + L + 0.5 * RR)$.

The relative increase in node abort rate due to using prefix-consistent snapshot isolation instead of conventional snapshot isolation $= (D + L + 0.5 * RR)/(L + RR)$.

7.4 Replicated Database Response Time

In this subsection, we consider the time it takes to execute update and read-only transactions. We define the response time of a transaction to be time taken from receiving the transaction at a database site until knowing the commit or abort status at the same database site. For prefix-consistent snapshot isolation in Figure 1, the response times of update and read-only transactions are $(L + RR)$ and (L), respectively. Similarly for conventional snapshot isolation 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 prefix-consistent snapshot isolation are lower, at the cost of **(1)** letting read-only transactions to observe less recent data and **(2)** 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 [23] and request-response delay of 200 ms over the Internet [5]. We used these numbers because we intend to use PCSI in a database replicated over the internet. TPC-W is the industry standard benchmark for e-commerce systems [8, 24]. However, we vary the ratio between some of these parameters in Figures 3 and 4.

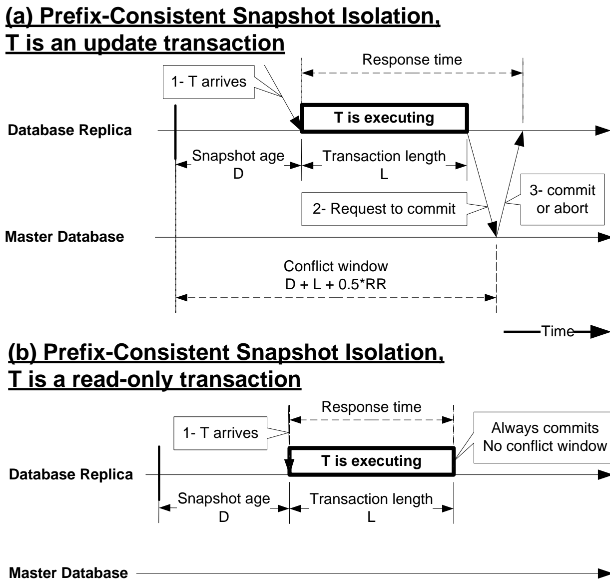

Figure 1: **Transaction execution in a replicated database using prefix-consistent snapshot isolation with master database and centralized certification.**

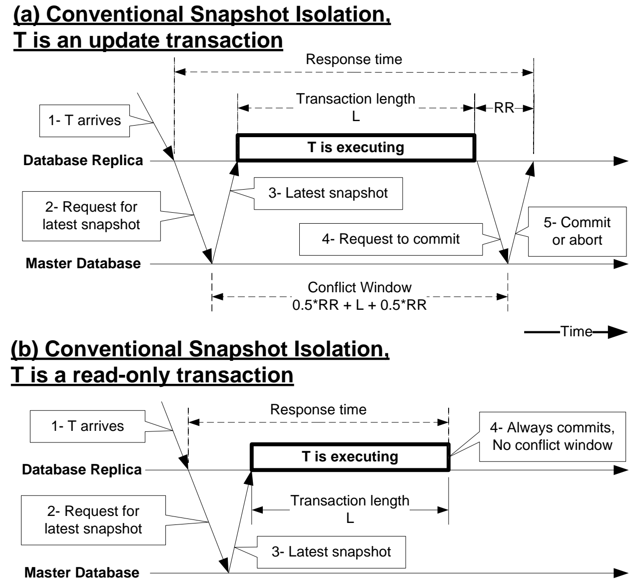

Figure 2: **Transaction execution in a replicated database using conventional snapshot isolation with master database and centralized certification.**

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. Considering the above two metrics, update transactions perform nearly equally under PCSI and CSI. 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 abort rate in CSI. The graph contains several curves that correspond to different values of D/L. Initially, PCSI aborts more transactions than CSI. However, 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. (This ratio is not a function of D). There are three curves corresponding to read-only transactions, update transactions, and finally "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.

7.6 Summary

For centralized databases, the abort rate under GSI (when enforced artificially) is higher than that of CSI and 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 conventional snapshot isolation, relatively few papers have discussed its properties. Conventional snapshot isolation was first introduced in 1995 [6, 15]. 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 [10]. Schenkel et al. [20] have discussed using CSI in federated transaction 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. [2] 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. [22] 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.

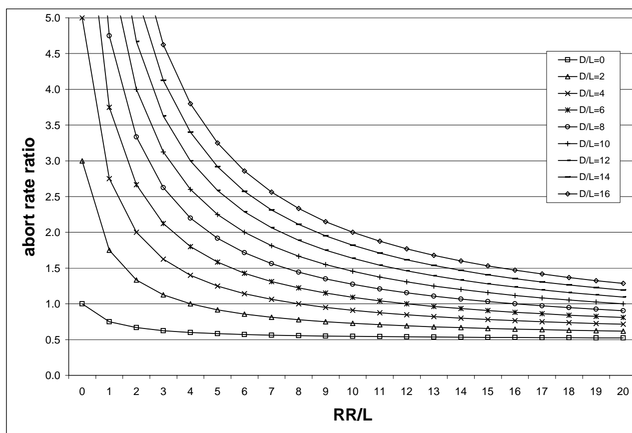

Figure 3: Relative update transactions abort ratio of PCSI to CSI on a replicated database with centralized certification, parameterized by D/L, (X-axis is RR/L).

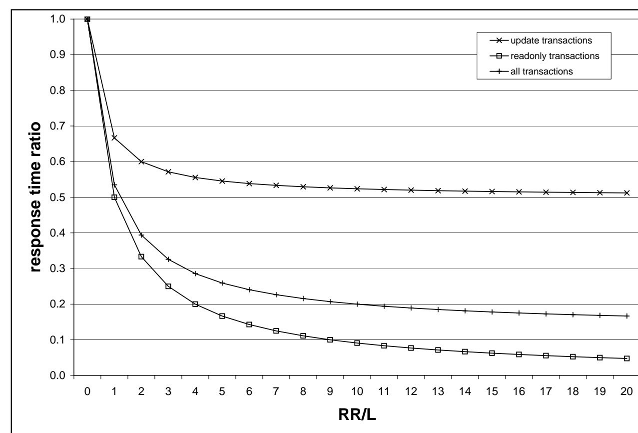

Figure 4: Relative response time ratio of PCSI to CSI on a replicated database with centralized certification, (X-axis is RR/L).

Kemme et al. [17] discussed how to implement different isolation levels (including serializability) in replicated databases using group communication primitives, in addition they implemented Postgres-R [16] and their experimental evaluation shows the feasibility of using eager replication protocols.

Fekete et al. [9, 10] have studied the conditions under which transactions executing on a centralized database with conventional snapshot isolation produce serializable histories. They developed a syntactic condition similar to condition C1 and it is sufficient but not necessary. They also show how database applications can be tested against the syntactic condition by building an extended Sibling--Conflict graph [22]. 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 C1 is an extension to their syntactic condition, and it applies to generalized snapshot isolation in centralized and distributed environments. Also condition C2, allows more transaction to commit in a serializable history than those two conditions.

Dahlin et al. [11] discussed application specific data replication techniques for edge services over the Web, with the objective of reducing response time, scaling up system performance, and enhancing the availability of Web services. 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 transparently rewriting the transactions in the application to satisfy condition C1, or by enforcing condition C2 at run-time.

9. CONCLUSIONS AND FUTURE WORK

This research presents of a new database concurrency control algorithm, *generalized snapshot isolation (GSI)*, which is an extension to conventional snapshot isolation (CSI). GSI does not delay transactions, which may observe an old snapshot of the database. We discussed the serializability properties of GSI, and showed two conditions that make transactions running under GSI produce serializable histories.

We presented *prefix-consistent snapshot isolation (PCSI)*, a special case of GSI that is particularly 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 the database site. The most important benefit of using PCSI is that it does not delay read-only transactions and executions can be made serializable if needed.

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, 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 generate dynamic Web content using a geographically distributed network of proxies. This workload is especially suitable for PCSI: It has many more read-only transactions than update transactions, and update transactions are short.

10. REFERENCES

[1] Data Concurrency and Consistency, Oracle8 Concepts, Release 8.0: Chapter 23. Technical report, Oracle Corporation, 1997.

[2] 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.

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

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

[10] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, June 1996.

[11] 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.

[12] 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.

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

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

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

[16] 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.*

[17] 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.*

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

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

[20] 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.

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

[22] 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.

[23] 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`.

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

APPENDIX

A. PROOF OF PROPOSITION 2

PROPOSITION 2. *Any update history h that satisfies GSI's rules G1 and G2 and condition C1 also satisfies CSI.*

PROOF: For a contradiction, assume it does not. Then h violates either (a) CSI's read rule or (b) CSI's commit rule.

(a) From CSI's read rule, there must exist some transaction T_i in h such that (a.1) T_i reads X from T_j and $start(T_i) \leq commit(T_j)$ or (a.2) there exists a transaction T_k that updates X such that $start(T_i) \leq commit(T_k) \leq commit(T_j)$. Case (a.1) cannot happen since transactions do not read uncommitted data under GSI. Consider case (a.2). From GSI's commit rule, it has to be that $commit(T_j) < snapshot(T_k)$, otherwise T_k would abort. From GSI's read rule, we have $snapshot(T_i) < commit(T_k)$, otherwise T_i would read from T_k; moreover, since T_i reads X from T_j, $commit(T_j) < snapshot(T_i)$. Therefore, T_i and T_k overlap, and by GSI's serializability condition it cannot be that $T_k \in h$ since $writeset(T_k) \cap readset(T_i) \neq \emptyset$.

(b) From CSI's commit rule, there are transactions T_i and T_j in h, such that (b.1) $start(T_i) < commit(T_j) < commit(T_i)$ and (b.2) $writeset(T_j) \cap writeset(T_i) \neq \emptyset$. Condition (b.1) implies that $snapshot(T_i) < commit(T_j) < commit(T_i)$, and so T_i and T_j overlap. Thus, since their writesets intersect, from GSI's commit rule, T_i should abort, a contradiction that concludes the proof. □

B. PROOF OF PROPOSITION 3

PROPOSITION 3. *Generalized snapshot isolation is serializable under C2.*

This proof is very similar to the proof of Proposition 1. Here we show only the different cases.

Since T_i and T_j satisfy condition C2, two cases should be considered:

Case 1. $readset(T_i) \cap writeset(T_j) = \emptyset$ and $(readset(T_j) \cap writeset(T_i) = \emptyset$ or $commit(T_i) < commit(T_j))$. Since T_i reads X and T_j writes X, it follows that the next case should hold.

Case 2. $readset(T_j) \cap writeset(T_i) = \emptyset$ and $(readset(T_i) \cap writeset(T_j) = \emptyset$ or $commit(T_j) < commit(T_i))$. Since $readset(T_i) \cap writeset(T_j) \neq \emptyset$, it follows that $commit(T_j) < commit(T_i)$, a contradiction that concludes the proof for update transactions. □

C. PROOF OF CORRECTNESS FOR PCSI IMPLEMENTATIONS

C.1 Centralized Certification

The rules of PCSI are followed in this algorithm. As each transaction reads the most recent snapshot that is available at the database replica and sees its own updates, the read rule P1 is satisfied. The replicas and the master database enforce the commit rule P2, where an update transaction commits only if no overlapping transaction has written into its writeset through the certification process.

To prove that this algorithm implements prefix-consistent snapshot isolation, it remains to show that if T_i overlaps with T_j in global time, then T_i overlaps with T_j when using versions. Assume that T_i overlaps with T_j in global time and T_j commits before T_i commits. Therefore, we have $snapshot(T_i) < commit(T_j) < commit(T_i)$. It is required to prove that $snapshotVer(T_i) < commitVer(T_j) < commitVer(T_i)$

We have $snapshotVer(T_i) < commitVer(T_i)$. Because T_j commits first at the master, we have $commitVer(T_j) < commitVer(T_i)$. Now regarding $snapshotVer(T_i)$, there are only two cases: First, $snapshotVer(T_i) < commitVer(T_j) < commitVer(T_i)$. Second, $commitVer(T_j) < snapshotVer(T_i) < commitVer(T_i)$. The second case is impossible and we show this by a contradiction: Assume $commitVer(T_j) < snapshotVer(T_i)$. Then, there exists $commit(T_k)$ such that $commit(T_k) < snapshot(T_i)$ and $commitVer(T_k) = snapshotVer(T_i)$. It must be true that $commitVer(T_j) < commitVer(T_k)$. Therefore, $commit(T_j) < commit(T_k)$. This leads to $commit(T_j) < commit(T_k) < snapshot(T_i)$ which is a contradiction that shows the second case is impossible. Therefore, since $snapshotVer(T_i) < commitVer(T_j) < commitVer(T_i)$, T_i must overlap with T_j when using versions. □

C.2 Distributed Implementation

To prove that this implementation is deterministic and obeys PCSI's rules, we need to show two properties. The first property is that at the certification of T_i, all replicas have the same SEQ and $V_{replica}$. Consequently, every replica reaches the same decision on the certification of T_i. Atomic broadcast is used to deliver the certification request, which contains $snapshotVer(T_i)$ and $writeset(T_i)$, to all replicas. It guarantees two properties: agreement (if a replica delivers message m, then every replica delivers m) and order (no two replicas deliver any two messages in different orders). We use mathematical induction on the length of SEQ to prove that SEQ is the same at all replicas. Induction base: all databases start from identical initial states (.i.e, same SEQ and $V_{replica}$. Induction hypothesis: assume that SEQ and $V_{replica}$ are the same at all replicas. Induction step: all replicas reach the same abort decisions on all certification requests containing $snapshotVer(T_k)$ and $writeset(T_k)$, until such T_k that is ready to commit. Hence, each replica increments $V_{replica}$ and appends $\{V_{replica}, writeset(T_k))$ to SEQ, such that all replicas get new identical SEQ and $V_{replica}$.

The second property is that if T_i overlaps with T_j in global time, then T_i must overlap with T_j when using versions. The proof of this property is identical to the proof of the corresponding property for centralized certification in the pervious subsection.