One-Copy Serializability with Snapshot Isolation under the Hood

Mihaela A. Bornea [1], Orion Hodson [2], Sameh Elnikety [2], Alan Fekete [3]

[1]*Athens U. of Econ and Business,* [2]*Microsoft Research,* [3]*University of Sydney*

***Abstract*—This paper presents a method that allows a replicated database system to provide a global isolation level stronger than the isolation level provided on each individual database replica. We propose a new multi-version concurrency control algorithm called, serializable generalized snapshot isolation (SGSI), that targets middleware replicated database systems. Each replica runs snapshot isolation locally and the replication middleware guarantees global one-copy serializability. We introduce novel techniques to provide a stronger global isolation level, namely readset extraction and enhanced certification that prevents read-write and write-write conflicts in a replicated setting. We prove the correctness of the proposed algorithm, and build a prototype replicated database system to evaluate SGSI performance experimentally. Extensive experiments with an 8 replica database system under the TPC-W workload mixes demonstrate the practicality and low overhead of the algorithm.**

I. Introduction

In many server systems replication is used to achieve higher performance and availability than a centralized server. Replication in database systems is, however, particularly challenging because the transactional semantics have to be maintained. The effects of an update transaction at one replica have to be efficiently propagated and synchronized at all other replicas, while maintaining consistency for all update and read-only transactions. This challenge has long been recognized [18], leading to several replication protocols that explicitly trade-off consistency to achieve higher performance: A replicated database system may provide a lower isolation level than a centralized database system.

We show, contrary to common belief, that a replicated database system can efficiently provide a global isolation level stronger than the isolation level provided by the constituent replicas. We focus here on snapshot isolated database systems and introduce a concurrency control algorithm that guarantees global one-copy serializability (1SR), while each replica guarantees snapshot isolation (SI), which is weaker than serializability. We support this claim by proposing an algorithm, proving its correctness, implementing it, and building a prototype replicated database to evaluate it experimentally.

Database engines such as PostgreSQL and Oracle support SI as it provides attractive performance for an important class of transactional workloads that have certain properties, e.g., dominance of read-only transactions, short updates and absence of write hot-spots. To take advantage of this performance gain, some database engines support SI in addition to traditional locking schemes. For example, Microsoft SQL Server supports both SI and 2PL. The performance gain of using SI comes at a correctness cost: SI is not serializable.

With careful design and engineering, SI engines can be replicated to get even higher performance with a replicated form of SI such as Generalized Snapshot Isolation (GSI) [13], which is also not serializable. The objective of this paper is to provide a concurrency control algorithm for replicated SI databases that achieves almost the same performance as GSI while providing global one-copy-serializability (1SR).

To guarantee serializability, concurrency control algorithms typically require access to data read and written in update transactions. Accessing this data is especially difficult in a replicated database system since built-in facilities inside a centralized DBMS, like the lock manager, are not available at a global level. Moreover, replicated systems face the additional challenge of how this data is extracted, represented and communicated among the replicas. Notice that serializability algorithms for a centralized system, such as two-phase locking (2PL) and serializable snapshot isolation (SSI) [7], are not sufficient in replicated database systems: while every replica enforces serializability locally for local transactions, the replicated database system does not provide 1SR.

We propose a new concurrency control algorithm, called Serializable Generalized Snapshot Isolation (SGSI), for middleware replicated database systems. Each replica uses snapshot isolation, while the replication middleware applies SGSI to ensure one-copy serializability. SGSI validates both the writeset and readset of update transactions.

Typically, the readset is much larger than the writeset and it is challenging to identify and represent. Our work is the first to address this challenge in a replicated setting. We introduce a technique for extracting the readset by applying automatic query transformation for each `SQL` statement in an update transaction. We describe how readsets are certified and prove certification correctness. Surprisingly, we find that readset certification supersedes writeset certification: Writeset certification is no longer needed. Contrary to prior work, writesets are extracted for update propagation and durability, *not* for preventing global write-write conflicts through writeset certification.

We show how to implement SGSI in the replication middleware, providing several practical advantages. Changing the replication middleware is easier than changing the database engine, which is more complex consisting of millions source code lines and could be closed-source such as Microsoft SQL Server and Oracle. The middleware implementation is easier to deploy; It can be used with engines from different vendors, and does not have to be changed with engine upgrades.

The main contributions of this work are the following:

- We propose SGSI, a novel concurrency control algorithm for replicated databases with SI replicas. SGSI ensures 1SR, an isolation level stronger than the level of individual components. We formally prove SGSI correctness;
- We are the first to address the problem of readset management in a replicated setting. We introduce a framework which provides techniques to extract, represent and communicate the readset among replicas in the system;
- We show how to implement SGSI and build a prototype middleware replicated system. We provide concrete solutions for extracting the readset of update transaction and for its certification in the relational model;
- We conduct extensive experiments showing SGSI is practical guaranteeing correctness at a low performance cost.

The paper is structured as follows. Section II provides background information on snapshot isolation and its anomalies. We define SGSI in Section III while in Section IV we show how SGSI is used in a replicated system. We present the readset certification framework for a relational DBMS in Section V and readset certification in Section VI. We discuss the prototype implementation and experimental evaluation in Section VII. The paper ends with related work in Section VIII and conclusions in Section IX.

II. Background

In this section we present a brief introduction to SI, GSI, 1SR and provide examples showing non-serializable executions. SGSI prevents all such anomalies.

A. Concurrency Control

Snapshot Isolation (SI). Snapshot isolation (SI) [5] provides each transaction with a snapshot of the database at the time of transaction start. Snapshot isolation is available in several database engines, such as Oracle, PostgreSQL, and Microsoft SQL Server. Moreover, in some systems that do not implement two-phase locking (2PL) schemes, including Oracle and PostgreSQL, SI is the strongest available isolation level. Snapshot isolation has attractive performance properties when compared to two-phase locking (2PL), particularly for read dominated workloads: Under SI, read-only transactions can neither block, nor abort, and they do not block concurrent update transactions. However, SI allows non-serializable behavior and may introduce inconsistencies.

Generalized Snapshot Isolation (GSI). Generalized Snapshot Isolation (GSI) [13] extends SI to replicated databases. GSI allows transactions to use local snapshots of the database on each replica and provides the same desirable non-blocking and non-aborting properties as SI for read-only transactions. GSI has the same serializability anomalies as SI.

One-Copy Serializability (1SR). The main correctness criterion for replicated databases is One-Copy Serializability (1SR) [6]. The effect is that transactions performed on the database replicas have an ordering which is equivalent to an ordering obtained when the transactions are performed sequentially in a single centralized database.

B. Serialization Anomalies Under SI

Since SI is weaker than serializability, it allows certain anomalies. These anomalies are also allowed by GSI since SI is a special case of GSI. SGSI prevents all serialization anomalies.

Write Skew Anomaly. The following scenario can introduce the write skew anomaly [5]. Assume two transactions T_1 and T_2 that withdraw money from two bank accounts X and Y. The bank enforces the constraint that the sum of X and Y is positive. Transactions T_1 and T_2 are executed concurrently on a database where each account, X and Y, contains an initial balance of 50. The following history is not serializable, but can be generated under SI:
$h_1 = R_1(X_0, 50),\ R_1(Y_0, 50),\ R_2(X_0, 50),\ R_2(Y_0, 50),$
$W_1(X_1, -40), W_2(Y_1, -40)$. At the end of this history the constraint imposed by the bank is violated.

Phantom Anomaly. Phantoms [14] are caused by data item insertions or deletions. The effect of this type of anomaly is shown in the following non-serializable history:
$h_2 = R_1(X_0, 50),\ R_1(Y_0, 50),\ R_2(X_0, 50),\ R_2(Y_0, 50),$
$W_2(Z_0, -20), W_1(X_2, -40)$. X and Y are accounts belonging to the same group on which the bank should enforce a positive balance. While transaction T_1 withdraws 90 from account X, transaction T_2 creates a new account Z with a fee of 20. These operations result in a negative group balance.

Read-Only Transaction Anomaly. The read-only transaction anomalies [17] result in inconsistent output of read-only transactions.
$h_3 = R_2(X_0, 0), R_2(Y_0, 0), R_1(Y_0, 0), W_1(Y_1, 20), C_1,$
$R_3(X_0, 0), R_3(Y_1, 20), C_3, W_2(X_2, -11), C_2$. Transaction T_1 deposits 20 in a savings account Y, transaction T_2 withdraws 10 from checking account and pays 1 as overdraft penalty. Transaction T_3 just outputs the balance for the client. The anomaly in this example is that the user sees values 0 and 20 while the final values are -11 and 20. Such values would not be reported in a serializable history.

III. SGSI Concurrency Control Model

Here we define SGSI and show that it guarantees serializability. We first introduce a multi-version concurrency control model to formally define SGSI and then we prove its correctness.

A. Database and Transaction 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. We denote T_i's write 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 on item Y by $R_i(Y_j)$, which means that T_i reads the version of item Y produced by transaction T_j. T_i's commit or abort is denoted

by C_i or A_i, respectively. To simplify the presentation, we assume that transactions do not contain redundant 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 old and 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 [6]. To simplify definitions, we assign a distinct time to each database operation, resulting in a total order consistent with the partial order $\prec$.

In SGSI, each transaction T_i observes a snapshot of the database that is taken at some time, denoted $snapshot(T_i)$. 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 for transaction T_i:

- $snapshot(T_i)$: the time when T_i's snapshot is taken.
- $commit(T_i)$: the time of C_i, if T_i commits.

We define the relation *impacts* for update transactions.

- T_j *write-impacts* T_i **iff**
  $writeset(T_i) \cap writeset(T_j) \neq \emptyset$, **and**
  $snapshot(T_i) < commit(T_j) < commit(T_i)$.
- T_j *read-impacts* T_i **iff**
  $readset(T_i) \cap writeset(T_j) \neq \emptyset$, **and**
  $snapshot(T_i) < commit(T_j) < commit(T_i)$.

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" to mean that if T_i were to commit now, then T_j would impact T_i.

B. SGSI Definition

SGSI has three rules: R1 regulates read operations, while R2 and R3 regulate commit operations. For any history h created by SGSI, the following properties hold (indices i, j, and k are different):

- **R1. (SGSI Read Rule)**
  $\forall T_i, X_j$ 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** $snapshot(T_i) < commit(T_k)]$.
- **R2. (SGSI No Write Impact Rule)**
  $\forall T_i, T_j$ such that $C_i, C_j \in h$:
  4- $\neg(T_j$ write-impacts $T_i)$.
- **R3. (SGSI No Read Impact Rule)**
  $\forall T_i, T_j$ such that $C_i, C_j \in h$:
  5- $\neg(T_j$ read-impacts $T_i)$.

The read rule R1 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.

Rules R2 and R3 limit which transactions can commit. This process is called certification and we will see in the next section that they require communicating readsets and writesets. The no-write-impact rule R2 prevents any update transaction T_i from committing if it is write-impacted by another committed update transaction. Rule R3 is similar, preventing T_i from committing if it is read-impacted.

C. SGSI Correctness

Theorem 1: (Serializability.) R1, R2 and R3 ensure one-copy serializability.

Proof: Let h be a history that satisfies $R1$, $R2$, and $R3$. We show that whenever there is a dependency $p \to q$ in h, where p is an operation in T_i and q is an operation in T_j on the same data item, and where T_i and T_j both commit, then $commit(T_i)$ precedes $commit(T_j)$. This implies that h is view serializable, with the serialization order given by the order of commit events [31] (see also Theorem 3.13 of [32]). Consider the possible dependency types.

1) wr **true-dependency**. Assume for some item X, $p = W_i(X_i)$ and $q = R_j(X_i)$. $R1$ gives directly that $commit(T_i) < snapshot(T_j)$ and $snapshot(T_j)$ is before $commit(T_j)$. Thus the whole execution interval of T_i comes before the execution interval of T_j, and in particular $commit(T_i) < commit(T_j)$.
2) ww **output-dependency**. Assume for some item X, $p = W_i(X_i)$, $q = W_j(X_j)$ and X_i precedes X_j in the version order on X. Since $X \in writeset(T_i) \cap writeset(T_j)$, the intersection of writesets is non-empty. Since $R2$ says that $\neg(T_j$ write-impacts $T_i)$, we see that either $commit(T_j) < snapshot(T_i)$ or else $commit(T_i) < commit(T_j)$. In the former case we have that the whole execution interval of T_j comes before the execution interval of T_i, which contradicts the version order placing X_i before X_j. Thus we are left with the latter case, that is, $commit(T_i) < commit(T_j)$.
3) rw **anti-dependency**. Assume for some item X, $p = R_i(X_k)$, $q = W_j(X_j)$ and X_k precedes X_j in the version order on X. Since $X \in readset(T_i) \cap writeset(T_j)$, the intersection of $readset(T_i)$ with $writeset(T_j)$ is non-empty. Since $R3$ says that $\neg(T_j$ read-impacts $T_i)$, we see that $commit(T_j) < snapshot(T_i)$ or $commit(T_i) < commit(T_j)$. In the former case we have (by rule $R1$ and the fact that

[1]This assumption is not restrictive: These redundant operations can be added to the model, but they complicate the presentation.

$R_i(X_k) \in h$) that $commit(T_j) < commit(T_k)$ which contradicts the version ordering placing X_k before X_j. Thus we are left with the latter case, that is, $commit(T_i) < commit(T_j)$.

■

IV. SGSI in a Distributed System

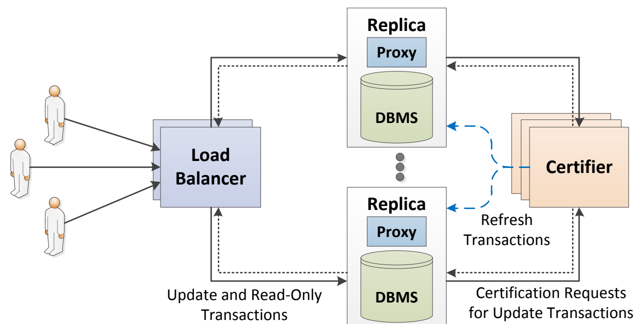

Fig. 1. SGSI Replicated Database System Architecture.

SGSI is defined in the previous section. Here we focus on how to use SGSI in distributed system. We introduce a distributed system model and present the replicated database architecture. We propose a certification algorithm to enforce SGSI rules R2 and R3. We defer the implementation aspects for relational database engines to the next section.

A. Distributed System Model

We assume an asynchronous distributed system composed of a set of database sites $Replica_1 \ldots Replica_n$ which communicate with a reliable message passing protocol. No assumptions are made regarding the time taken for messages to be transmitted or subsequently processed. Each site has a full copy of the database.

B. Replicated System Architecture

The system architecture (Figure 1) is comprised of three component types: load balancer, replica and certifier. The design is consistent with the state-of-the-art for middleware-based replicated database systems [9], [11], [20], [21], [24].

Load Balancer. The load balancer receives transactions from client applications and passes them to the replicas using a load balancing strategy such as round robin or least number of connections. It also relays responses from replicas to clients.

Certifier. The certifier performs the following tasks: (a) detects and prevents system-wide conflicts, and assigns a total order to update transactions that commit, (b) ensures the durability of its decisions and committed transactions, and (c) forwards the writeset of every committed update transaction to the replicas in form the of refresh transactions.

Replica. Each replica consists of a proxy and a standalone DBMS employing snapshot isolation. The proxy receives *client transactions* from the load balancer and *refresh transactions* from the certifier. Refresh transactions are those that have been executed at other replicas and have been certified. These are applied directly to the database. For client transactions, the proxy applies the SQL statement inside each transaction to the database, and sends the response to the client via the load balancer. For update statements, the proxy extracts the partial writeset of the transaction for early certification [11]. It checks whether this partial writeset conflicts with any pending writeset of refresh transactions to prevent the hidden deadlock problem [33]. In the case of conflict, the client's update transaction is aborted. When the client requests to *commit* a transaction, the proxy commits immediately if the transaction is read-only. For update transactions, the proxy sends a certification request to the certifier and awaits a decision. When certifier's decision is received, the proxy commits (or aborts) the transaction to the database and sends the outcome to the client.

Fault Tolerance. The system assumes the standard crash-recovery failure model [2]. In this failure model, a host may crash independently and subsequently recover. The certifier is lightweight and deterministic, and may be replicated for availability [11] using the state machine approach [27]. The load balancer is lightweight because it maintains only a small amount of soft-state and a failover (standby load balancer) may be used for availability. The hard (persistent) state in the system is maintained by database replicas, and this state is orders of magnitude larger than the load balancer's state. After a failure, the failed component can recover using standard approaches as discussed in prior work [11]. This design has no single point of failure as each component is replicated.

In our implementation, the database replicas are replicated for higher performance. We use a single load balancer and a certifier, which both can also be replicated for higher availability. As shown in prior work [10], [11], [12], a single load balancer and a single certifier are sufficient to match a replicated database system of up to 16 replicas.

C. SGSI Certification Algorithm

We present the SGSI certification algorithm for the asynchronous distributed model. The objective of the algorithm is to commit transactions when they satisfy SGSI rules R2 and R3. In order to detect and prevent conflicts, the certifier manages the writesets produced by the committed transactions together with the commit order. Roughly speaking, to commit an update transaction, the replica sends a certification request containing the transaction readset and writeset. The certifier ensures that there is no committed update transaction which is read-impacting (i.e., the readset is still valid) or write-impacting (i.e., there is no write-write conflict).

Database Versioning. When an update transaction commits, it creates a new version of the database, identified by a version number assigned by the certifier forming the global commit order. Initially, the database starts at version 0. A replica evolves from version V_i to version V_{i+1} by applying the writeset of update transaction T that commits at version $i+1$.

Replacing timestamps. For transaction T, both $snapshot(T)$ and $commit(T)$ have been defined in the transactional model

Algorithm 1 Certification Algorithm.

1- When $Replica_j$ sends $(V_{Rep_j}, writeset(T), readset(T))$,
  $Certifier$ receives and executes:
  if validate($V_{Rep_j}, writeset(T), readset(T)$) == false **then**
    send (abort,-,-) to $Replica_j$
  else
    $V_{master} \leftarrow V_{master} + 1$
    $h \leftarrow T$; manage $writeset(T)$
    send (commit, V_{master}) to $Replica_j$
    send refresh($V_{master}, writeset(T)$) to $Replica_i$, $i \neq j$.
2- When ready to commit T, $Replica_i$ executes:
  send $(V_{Rep_i}, writeset(T), readset(T))$ to $Certifier$
  wait until receive $(result, V_{new})$ from $Certifier$
  if $result$ == commit **then**
    db-apply-writesets($writeset(T)$)
    $V_i \leftarrow V_{new}$
    db-commit(T)
  else
    db-abort(T).
3- When $Certifier$ sends refresh($V_{master}, writeset(T_j)$),
  $Replica_k$ receives and executes:
  db-apply-writesets($writeset(T_j)$)
  $V_{Rep_k} \leftarrow V_{master}$.

in terms of timestamps. We cannot use timestamps in the asynchronous distributed model because it requires access to *global time*. Instead, we use versions, as follows:

- $snapshot(T)$: the version of database that T observes.
- $commit(T)$: the commit version of T.

Algorithm Description. As presented in Algorithm 1, the certifier receives a certification request containing the writeset, the readset and the version of replica snapshot. Given a certification request from $Replica_j$ for a transaction T with $snapshot(T) = V_{Rep_j}$, the certifier accesses all committed writesets with a version greater than $snapshot(T)$. Writesets with version number smaller or equal to $snapshot(T)$ belong to transactions that committed before the snapshot of T was taken and transaction T sees their effects. For each accessed writeset item X, the certifier checks if $X \in writeset(T)$ or $X \in readset(T)$, in which case T is aborted since its commit would introduce an impacted transaction in the history and an abort message is sent to $Replica_j$. If certification is successful, the transaction commits and the certifier assigns a global order to the transaction which becomes the version of the database. A commit response message is sent to $Replica_j$, and a refresh message containing the newly added writeset is sent to all other replicas in the system.

The second part of Algorithm 1 describes the replica's actions when it receives a request to commit an update transaction T. The replica snapshot version, the writeset and the readset are sent to the certifier. The replica waits for the certification response to commit or abort T. In the third part of Algorithm 1, each replica applies the writeset received in a refresh message to its local database.

We state the following theorem for algorithm correctness.

Theorem 2: (Certification with versions.) The certification algorithm (which uses versions) satisfies SGSI rules R2 and R3 (which are expressed in timestamps).

We omit the proof of Theorem 2, but show the proof of an equivalent theorem, Theorem 3: Certification in the relational model, as it is more relevant to our implementation in the relational model.

V. Certification Framework for Relational DBMS

A. Transaction Model and Query Language

We employ a relational transaction model in which each transaction starts with a BEGIN statement and ends with a COMMIT or ABORT. Read and write operations are expressed through SQL queries. We support a large subset of SQL queries in a transaction. The notations used in the SQL transaction model are summarized in the Table I.

SQL Queries inside a transaction

A. SELECT $expr_list$ FROM R_i WHERE $pred(R_i)$
B. INSERT INTO R_i VALUES $(values)$
C. UPDATE R_i SET $attr_j = value_j$ WHERE $pred(R_i)$
D. DELETE FROM R_i WHERE $pred(R_i)$
E. SELECT AGG($attr$) FROM R_i WHERE $pred(R_i)$
  GROUP BY $group_attr$
  HAVING $pred$(AGG($attr$))
F. SELECT $attr_list$
  FROM $R_1 ... R_i ... R_n$
  WHERE $pred(R_1)$ LOP ... LOP $pred(R_i)$
    LOP ... LOP $pred(R_n)$
    LOP $join_pred(R_i, R_j)$

Symbol	Description
R_i	Relation belonging to the database schema
$expr_list$	List of projected attributes and expressions as used in SELECT SQL statement
$values$	List of attribute values as used in the INSERT SQL statement
$attr$	attribute of relation R
$pred(R_i)$	SQL selection predicate on attributes of relation R_i
$join_pred(R_i, R_j)$	SQL join predicate on the join attributes of R_i and R_j
pk	Primary key
$@pk$	Primary key value
AGG($attr$)	Aggregate(SUM, AVG, MIN, MAX, COUNT, TOP K) applied on attribute $attr$ of relation R
$group_attr$	Attribute of relation R included in the GROUP BY clause
$pred$(AGG($attr$))	a general predicate on the value of the aggregate included in the HAVING clause
LOP	a logical operator: OR, AND, NOT

TABLE I
Symbols Used in SQL Query Subset.

Our model includes basic SQL statements accessing one relation: SELECT, UPDATE, DELETE, and INSERT. Grouping and aggregates (AGG i.e. MIN, MAX, AVG, SUM) are part of our model. We also support queries involving more than one relation. Finally, our query language also incorporates subqueries as well as UNION, INTERSECT, EXCEPT, ORDER BY and DISTINCT, the details of which are omitted due to the lack of space.

B. Writesets

Writesets are used for certification and for update propagation. We assume that each tuple in the database is identified

by its primary key value. Tuples in the writeset can be introduced by UPDATE, INSERT or DELETE SQL statements in a transaction.

A writeset is a list of tuples, including the old and new values for each attribute in the tuple.

If certification is successful, the certifier adds both the old and new values of the writeset to its database, and sends the new values of the writeset to replicas for update propagation. The writeset of a transaction is not certified. As we show in Section VI-B it is sufficient to certify the readset.

There are several approaches to extract writesets, including triggers, log sniffing or direct support from the database engine (e.g., in Oracle) [11], [21]. In our prototype the writeset is extracted by applying the predicates of the update statements on the replica database since we parse SQL statements to extract the readset. We show in the next section that readset certification subsume writeset certification.

C. Readsets

Readset identification is challenging and, as far as we know, this work is the first to consider this problem in a replicated setting. In contrast to writesets, identifying readsets using the tuples that are read by a transaction based on the primary keys is a poor choice: First, the readset of a transaction is typically much larger than the writeset, and it is, therefore, expensive to send the set of rows read in a transaction from the replica to the certifier. Second, without capturing predicates in the readset, phantoms might be introduced, obviating 1SR.

Our approach is based on the observation that the readset of a SQL query is defined by its predicate. The readset is a list of predicates expressed as SQL queries. We explain how to extract the readset in Section VI.

D. Writeset Management at Certifier

The certifier manages two data structures: a persistent log (hard-state) and a main memory database (soft-state). The writesets of all committed update transactions are stored in the log. After a crash, certifier uses the log to recover its state.

The content of recent committed writesets is maintained in an in-memory database named *certification database* (CertDB). The CertDB is used to certify update transactions by validating their writesets and readsets, and is not durable as its contents can be recovered from the log file. CertDB has a schema similar to that of the replicated database, augmented with a version attribute in each relation. After a successful commit, each tuple in the transaction's writeset is extended with the commit version of the transaction and is inserted in the corresponding relation in CertDB. CertDB contains both the old and the new values of an updated tuple in order to support predicate-based certification.

Certification must be executed quickly since it is required for each update transaction. This is achieved by keeping CertDB small and in-memory. The amount of space needed by the CertDB is small because it is proportional to the sum over all active update transactions multiplied by the number of database elements that the transaction updates. CertDB size is orders of magnitude smaller than the size of the database at each replica. The certifier periodically garbage collects its data structures, maintaining a small size for CertDB. Old writesets are removed from the log file to limit its size. When the certifier receives a certification request for a transaction with a snapshot version that is older than CertDB (i.e., when older versions have been purged from CertDB), the transaction is aborted. This happens rarely since CertDB maintains enough versions to make this event unlikely.

VI. Readset Certification

Readset certification is one of the key contributions in this work. The readset is extracted by applying an automatic query transformation to each SQL statement inside an update transaction. The transformation creates the *certification queries* which are evaluated during the certification process. First we introduce basic certification and next the enhanced certification. We present certification correctness proof for each type of transformation. In addition, we show that readset certification subsumes writeset certification.

We certify the readsets for update transactions. Read-only transactions do not need certification. Roughly speaking, to certify the readset of a transaction we want to ensure that if the transaction executes on the latest version it would read the same values; that is, no concurrent update transaction committed writes into the readset.

We observe that the INSERT, UPDATE and DELETE statements have readsets since each SQL statement in our model has a predicate that defines the readset.

In order to perform the query transformation and extract the readset, the replica identifies the predicate of each statement. The certification query resulted from the transformation includes the predicate of the original query as well as a a version predicate which restricts certification to concurrent transactions. The certifier executes the certification queries of a given transaction T on CertDB. The result of the certification queries forms the conflict set, denoted $CS(T)$, of transaction T. If $CS(T) \neq \phi$, then transaction T aborts; otherwise, there is no impacting transaction and T commits.

Theorem 3: (Certification in the relational model.) When certifying transaction T with conflict set $CS(T)$, if there is a read-impacting or write-impacting transaction in the history, then $CS(T) \neq \phi$.

We prove the theorem for each statement type and present the certification queries.

A. SELECT Queries

The readset of a SELECT statement includes any tuple that matches the selection predicate $pred(R_i)$ on the attributes of relation R_i. The replica constructs the certification query using the original SQL statement by combining the following components: (a) SELECT * FROM; (b) the target table of the SQL query; (c) the content of WHERE clause, and (d) the version predicate.

Certification Queries: SELECT

A. SELECT * FROM R_i WHERE $pred(R_i)$ AND

$version > snapshot(T)$

`ORDER BY` and projections are ignored because they do not influence the predicate operation.

Proof of Theorem 3: Let T_i denote an update transaction to be certified, with its snapshot version $snapshot(T_i)$ and $readset(T_i)$ captured in the above certification query. Let T_j be a read-impacting transaction that created database version V_j. We show that $CS(T_i) \neq \phi$, which causes T_i to abort.

Since T_j read-impacts T_i, T_j writes a tuple that matches the predicate of the `SELECT` statement if T_i would be executed on version V_j. Let $t \in R$ be a this tuple. In this case $t \in CertDB$ and $V_j > snapshot(T)$. When the certifier executes the certification queries of T_i, t matches both the selection predicate and the version predicate. Thus $t \in CS(T_i)$. ■

B. Update Statements

In this section we show how to extract and certify the readset of update statements. One important result of this section is that certifying the readset of update statements also detects ww conflicts and it is, therefore, not necessary to certify the writeset. This claim is supported by the proof of Theorem 3.

The readset of an `UPDATE SQL` statement includes any tuple that matches the predicate on the target table. Similarly, the readset of a `DELETE` statement contains any tuple that matches the deletion predicate. The readset of an `INSERT` statement is identified by the primary key of the new inserted tuples, based on the fact that the database checks the uniqueness of the primary key. These conditions are captured by the following certification queries:

Certification Queries: Update Queries

B. `SELECT * FROM` R_i `WHERE` $pk = @pk$ `AND` $version > snapshot(T)$

C. `SELECT * FROM` R_i `WHERE` $pred(R_i)$ `AND` $version > snapshot(T)$

D. `SELECT * FROM` R_i `WHERE` $pred(R_i)$ `AND` $version > snapshot(T)$

Proof of Theorem 3: We show that when certifying T_i, if T_j write-impacts T_i, then $CS(T_i) \neq \phi$. Consider tuple $t \in R$ such that $t \in writeset(T_i)$ and $t \in writeset(T_j)$. We consider all cases that cause $t \in writeset(T_i)$:

- `UPDATE` statement. In this case transaction T_i modifies tuple t which satisfies the update predicate $pred(R)$.
  Since $t \in writeset(T_j)$, T_j modifies tuple t. The value of t in the snapshot of T_j also satisfies the update predicate $pred(R)$. (Otherwise there is another write-impacting transaction that modifies t from a value that satisfies $pred(R)$ to a value that does not. This transaction commits after the snapshot of T_i was taken and before the snapshot of T_j was taken). Thus, tuple t is in CertDB and the value of its version attribute is $V_j > snapshot(T_i)$. Moreover, after the certification queries of T_i are executed at the certifier, $t \in CS(T_i)$ and T_i aborts.
- `DELETE` statement. In this case transaction T_i modifies tuple t which satisfies the delete predicate $pred(R)$.
  Since $t \in writeset(T_j)$, T_j modifies tuple t. Using the same reasoning as in the previous case, we state that the value of t in the snapshot of T_j also satisfies the delete predicate $pred(R)$. Thus, tuple t is in CertDB and the value of its version attribute is $V_j > snapshot(T_i)$. Moreover, after the certification queries of T_i are executed at the certifier, $t \in CS(T_i)$ and T_i aborts.
- `INSERT` statement. In this case transaction T_i tries to insert tuple t in the database. The certification query for T_i contains a predicate on the primary key of t. Let T_j be a write-impacting transaction that also inserted (or modified) t in the database at version V_j. Since T_j committed, t exists in CertDB and its version attribute is $V_j > snapshot(T_i)$. t triggers the certification queries and $t \in CS(T_i)$ and T_i aborts.

■

C. Groups and Aggregates

Aggregate queries have predicates specified by the `WHERE` clause. Such predicates determine the readset. When present in an aggregate, the `HAVING` clause restricts the aggregation groups that appear in the query results. We remove it from the certification query in order to validate the results for all groups. Aggregate queries are transformed as follows:

Aggregates Certification

E. `SELECT * FROM` R_i `WHERE` $pred(R_i)$ `AND` $version > snapshot(t)$

The outcome of aggregates like `AVG` and `SUM` depends on the value of each tuple contained in the set over which they are evaluated. However, the outcome of `MIN` and `MAX` is determined by the value of one tuple; we introduce an optimization for this case. Consider the `MAX` aggregate where the `GROUP BY` statement is missing. In order to determine the maximum value, the replica DBMS reads all values of the relation. If concurrent (remote) transaction modifies any of the relation tuples, it causes the aggregate to abort during readset certification even if a transaction modifies tuples that do not influence the outcome of `MAX`. At the time when the certification queries are built the value of the `MAX` aggregate over $attr$ is already known and we assume it is equal to max_val. Moreover, the result of the aggregate changes if concurrent transactions write a tuples of R_i with $attr \geq max_val$. Based on the previous observation, the certification query can be rewritten as:

Aggregates Certification

E. `SELECT * FROM` R_i `WHERE` $pred(R_i)$ `AND` $attr \geq max_val$ `AND` $version > snapshot(T)$

D. Joins

Joins have `SELECT` queries involving several target relations $R_i, i = 1..n$. There are two types of predicates, combined by conjunctive and/or disjunctive logical operators, that define the readset. First, the selection predicates involve only attributes of a single relation R_i and are denoted by $pred(R_i)$. Second, the join predicates involve attributes that

define the join between two pairs of relations R_i and R_j and are denoted by $join_pred(R_i, R_j)$.

So far, the certifier maintains CertDB, which contains the writesets of recently committed transactions. CertDB is not enough to evaluate all predicates and to certify the readset. We present an approximation here and in the next subsection we show how to enhance the certifier to perform refined certification for joins.

Certifying a superset of the readset. The certifier can check for updates in each individual relation that participates in the join using the following certification queries:

Join Certification
for each relation R_i:

F. SELECT * FROM R_i WHERE $version > snapshot(T)$

It is easy to prove that these certification queries guarantee correctness since they match any committed writesets belonging to any of the R_i target relations.

E. Extended Certification

Data Managed at Certifier. Certification should ideally satisfy two properties: soundness (correctness, i.e., when a transaction is certified successfully, there is no read-impacting or write-impacting committed transaction), and completeness (accuracy, i.e., if there is no read-impacting or write impacting transaction, the certification request always succeeds).

The certifier maintains soundness at all times, but completeness is a function of the amount of data maintained at the certifier. At one end of the spectrum, maintaining the recent writesets at the certifier, may require using a superset of the query readset. This case can lead to an unnecessary abort. At the other end of the spectrum, maintaining the full database at the certifier achieves completeness but has the maximum overhead.

Unnecessary aborts may happen when the certifier does not maintain enough information to evaluate the readset join predicates. One alternative would be to have the replica send this information to the certifier; this approach is, however, not practical as it is likely too expensive to ship the needed data from the replica to the certifier.

Another alternative is to store and maintain the data required to evaluate the join predicates at the certifier. In essence, this is similar to maintaining a materialized view in a database system. Here the certifier manages a copy of the relations referenced in the query and we denote these relation by $R_{1_C} \ldots R_{n_C}$. Relations used in transactions that are aborted frequently are good candidates. The main cost of maintaining a relation in CertDB is the memory needed to store the relation as well as an increase in the processing of certification queries.

To maintain a relation at the certifier, we extend it with start and end versions (V_{Start} and V_{End}) to determine the visibility of each tuple, similar to the way versions are managed in a multi-version row storage engine, for example as in Postgres [29]. The latest version of a tuple has $V_{End} = -1$.

To certify transaction T containing a query with an arbitrary predicate $query_pred$ over the target relations $R_1 ... R_n$, we introduce two additional predicates. In order to detect tuples that are updated after the snapshot of T was taken, the certifier checks that $V_{Start} > snapshot(T)$ OR $V_{End} > snapshot(T)$. We call this the update predicate, denoted $upd(R_{i_C})$, and we employ it when accessing any relation R_{i_C} at the certifier. In order to detect tuples that are visible to transaction T, the certifier checks that NOT($V_{End} < snapshot(T)$ AND $V_{End} \neq -1$). We call this the visibility predicate, denoted $vis(R_{i_C})$, and we use it to access to any relation R_{i_C} at the certifier.

All SQL queries can be certified on the certifier copy of the database in order to reduce the superset of the readset.

Join Certification. We show below the extended certification queries for joins. Certification is successful if there is no tuple returned, showing the absence of any impacting transaction.

Certification Query: Joins
F. SELECT * FROM $R_{1_C} ... R_{i_C} ... R_{n_C}$
WHERE ($query_pred$)
AND ($upd(R_{1_C})$ ... OR $upd(R_{i_C})$ ... OR $upd(R_{n_C})$)
AND ($vis(R_{1_C})$ ... AND $vis(R_{i_C})$ ... AND $vis(R_{n_C})$)

Proof of Certification Completeness:
Let T_i be a transaction that needs to be certified and its readset is identified by the previous certification query. We show that if there is no impacting transaction, the certification request always succeeds.

Assume this transaction is aborted while there is no impacting transaction. If the transaction is aborted, the certification query returns at least one tuple. Consider the set of instances $t_{R_i} \in R_i$ that satisfy $query_pred$ and produce this tuple in the join result. This implies that there is at least one tuple instance $t_R \in R$ in this set that matches the version predicate $upd(R)$. Thus, we have $V_{Start} > snapshot(T_i)$ or $V_{End} > snapshot(T_i)$. This further implies that there is a transaction that committed after the snapshot of T_i was taken and it modified tuple t_R. Since all the components t_{R_i} that satisfy the $query_pred$ are visible to T_i, our initial assumption that there is no impacting transaction is contradicted. ∎

Proof of Certification Soundness:
Let T_i be a transaction that needs to be certified and its readset is identified by the previous certification query. Let transaction T_j be an impacting transaction that modifies tuple $t_R \in R$ and it creates version V_j.

We show that the certification queries for T_i report at least one result and transaction T_i aborts.

Since the update of tuple t_R in transaction T_j causes the modification of an item read by transaction T_i, there is a set of tuples $t_{R_i} \in R_i$, $i = 1..n$ and t_R influences the ability of this set to satisfy the predicate $query_pred$. This further implies that all tuples t_{R_i} are visible to the transaction T_i. Moreover, t_R can influence the outcome of the $query_pred$ evaluation if: (a) it was part of a tuple set that matched the predicate and it was modified by T_j, or (b) it is a new instance created by T_j that completes a set of tuples such that they satisfy the predicate. The same set of tuples mentioned above satisfies the $query_pred$ and the visibility predicates $vis(R_i)$, $i = 1..n$ of the certification queries when they are evaluated at the certifier copy of the database. However, in case (a),

tuple t_R has $V_{End} = V_j$ while in case (b), $V_{Start} = V_j$. $V_j > snapshot(T_i)$ and in both cases the update predicate $upd(R)$ is also satisfied. Thus, the certification query has at least one response and T_i aborts. ■

VII. Prototype Implementation and Evaluation

We build a replicated system for experimental evaluation. System architecture is described in Section IV. We extend the prototype with a web server front-end.

A. Implementation

Each replica runs a multithreaded C# application as the proxy and runs a local instance of Microsoft SQL Server 2008 as the database. SQL Server is configured to run transactions at snapshot isolation level with row-level locking. Replicas receive client transactions via the web front-end and extract the readset and writeset of each transaction. The readset is automatically identified by the replica from the text of `SQL` queries. We parse this text and locate the target relations as well as the predicates of the query. Then, we build the certification queries as explained in Sections VI, including extended certification in Subsection VI-E. The writeset is extracted from the `UPDATE`, `DELETE` and `INSERT` statements. We extract the predicate of `UPDATE` and `DELETE` statements and generate queries to capture the writesets. Notice that in the case of `UPDATE`, we capture both the new and old values.

The certifier is implemented as a C# application. It uses an in-memory SQLite.NET database for conflict detection. SQLite.NET constrains the implementation to a single connection to the in-memory database, serializing conflict detection.

The certifier batches commits to the in-memory database to mitigate the commit costs. Experimentally we found committing after batches of 32 certifications provides better throughput and scaling. The certifier also periodically cleans old state from the in-memory database to constrain memory usage and improve performance. In the measurements presented the cleaning period was every 5000 certifications and records older than 1000 versions are expunged.

The prototype implements session consistency [8] where each client sees increasing versions of the database system, and after a client commits an update transaction, the next client transaction observes the effects of this update.

The web front-end and load balancer component is a C# ASP.NET application running under IIS 7.0. The load generator emulates clients that drive the front end. Each emulated client is bound to a session and generates requests as it is driven through the client model. The requests are forwarded to replicas with round-robin load balancing policy.

The load generator implements the browser emulator described in the TPC-W specification [30] and is unmodified from prior research projects [11], [13].

B. Hardware Components

The machines used for the certifier and database replicas are Dell Optiplex 755 running 64-bit Windows Server 2008. Each has an Intel Core 2 Duo Processor running at 2.2GHz with 2GB DDR2 memory, and a 7200RPM 250GB SATA drive.

The load generator and the web front-end (and load balancer) run on Dell T3500 with Intel Xeon E5520 processors and 4GB of memory running Windows Server 2008. The load generator, web front-end and load balancer are well provisioned so that they do not become the bottleneck in any experiment. All machines have a single gigabit ethernet interface and are connected to one gigabit ethernet switch.

C. TPC-W Benchmark

TPC-W models an e-commerce site, in the form of an online bookstore. The TPC-W specification requires 14 different interactions, each of which must be invoked with a particular frequency. Of these interactions, eight initiate update transactions, whereas the other six generate read-only transactions. Each interaction may also involve requests for multiple embedded images for items in the inventory.

Mixes. TPC-W has three workload mixes that vary in the fraction of update transactions. The *browsing mix* workload has 5% updates, the *shopping mix* workload has 20% updates and the *ordering mix* workload has 50% updates. The *shopping mix* is the main mix but we have explored all mixes in our experiments. During the update propagation, the average writeset size is about 300 Bytes.

Database Size. In our experiments, the TPC-W database scaling parameters are 200 EBS (emulated browsers) and 10000 items. Each experiment begins with the same 850 MB initial database. The database fits within the memory available in each replica (2GB). This choice maximizes the demands on processor resources at the replica and at the certifier. A larger database would have a greater I/O component and pose less of a performance challenge.

Measurement Interval. Each experiment ran with a 3-minute warm up period followed by a 5 minute measurement interval.

D. SmallBank Benchmark

The SmallBank benchmark [3] models a simple banking database. It consists from three tables: Account (name, customerID), Saving (customerID, balance) and Checking (customerID, balance). It contains five transaction types for balance (Bal), deposit-checking (DC), withdraw-from-checking (WC), transfer-to-savings (TS) and amalgamate (Amg) which are assigned uniformly to clients. The Bal transaction is read-only while DC, WC, TS and Amg are update transactions. SmallBank is not serializable under SI and GSI.

We use a database containing 1000 randomly generated customers and their checking and savings accounts(as employed in prior work[7]). The test driver runs the five possible transactions with uniform distribution, creating a workload with 80% update transactions. The driver uses one thread per client. The running time for each experiment is 300 seconds. Each experiment is repeated 10 times and the error bars in Figure 8 show the standard deviation.

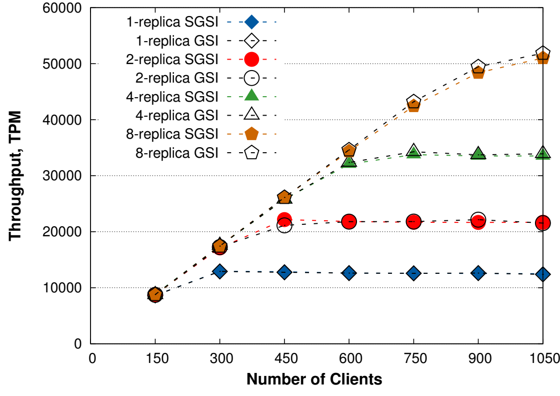

Fig. 2. Throughput of TPC-W Shopping Mix (20% updates).

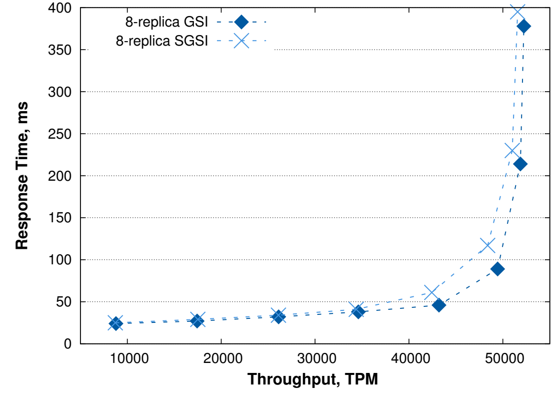

Fig. 4. Scalability of TPC-W Shopping Mix (20% updates).

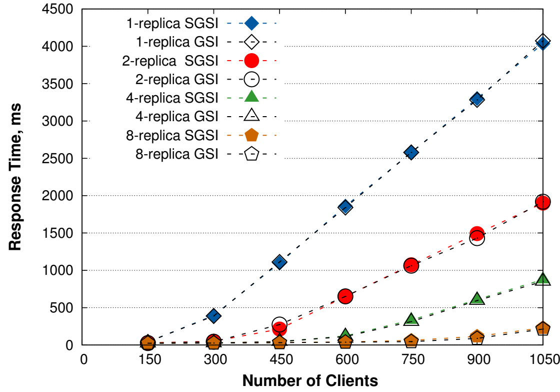

Fig. 3. Resp. Time of TPC-W Shopping Mix (20% updates).

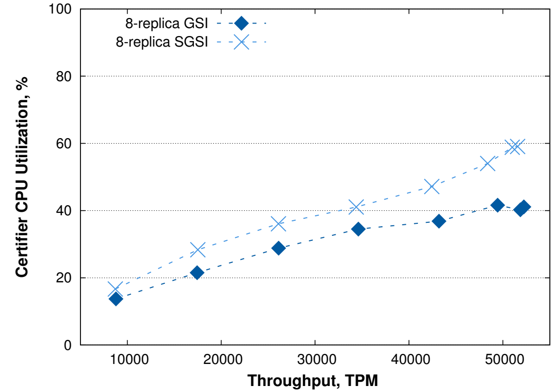

Fig. 5. Certifier CPU util., TPC-W Shopping(20% updates).

E. TPC-W Performance Evaluation

Our objective is to show that SGSI is a practical replication algorithm with competitive performance. We investigate its scalability and overhead. We use the TPC-W benchmark to assess SGSI performance as it is widely used to evaluate replicated database systems [11], [22], [33], [20].

*1) **Scaling of SGSI with Replication Degree**:* We use the main workload of TPC-W, the Shopping Mix (20% updates) to assess system scalability under SGSI in Figures 2 and 3. We vary the number of clients from 150 to 1050 along the X-axis, and we plot four curves for 1, 2, 4 and 8 replica SGSI systems (and we discuss GSI curves in next subsection). The Y-axis is either throughput in TPM (transactions per minute) or average response time in ms (milliseconds). Replication increases the peak throughput, in particular the 8-replica system reaches 51000 TPM, which is 4X the single replica performance. Replication is also used to reduce average response time; at the 1050 clients, response time is reduced from 4000 ms at 1 replica to 230 ms at 8 replicas. We conclude that the SGSI system provides good scalability with the number of replicas to increase throughput or reduce average response time.

*2) **Comparing SGSI to GSI**:* The performance of GSI represents an upper bound on the performance for SGSI. SGSI certifies readsets and writesets to provide serializability, whilst GSI only certifies writesets. Figure 4 shows the throughput against response time for an 8-replica system using the TPC-W Shopping Mix for both SGSI and GSI certification.

The SGSI curve is slightly below GSI, showing that they have almost the same performance. To explain the difference between SGSI and GSI curves, we study the overhead of SGSI certification, in terms of certifier response time and abort rate. The CPU is the bottleneck resource at the certifier. Figure 5 shows that certifier CPU utilization is under 60% and 43% for SGSI and GSI respectively. Certifier response time for SGSI varies from 7-20 ms and for GSI varies from 4-6 ms as the load at the certifier varies from 8000-51000 TPM.

The abort rates we observe for SGSI in our TPC-W experiments are low: the Shopping Mix has a peak of 8% of transactions aborting. For all mixes, we see the number of aborts increase with the rate of certifications. A more detailed study of abort rates is presented in Section VII-F.

In Figures 2 and 3, the performance of GSI and SGSI are shown for different degrees of replication. The performance of SGSI and GSI are almost identical in throughput and response time for all degrees of replication. We conclude that SGSI introduces a small overhead when compared to GSI due to extra processing needed to guarantee one-copy serializability.

*3) **Sensitivity to Update Transaction Ratio**:* We next turn to studying how SGSI scales under different ratios of update transactions. If the workload is read-only, replication should yield linear scalability. For update dominated workloads the

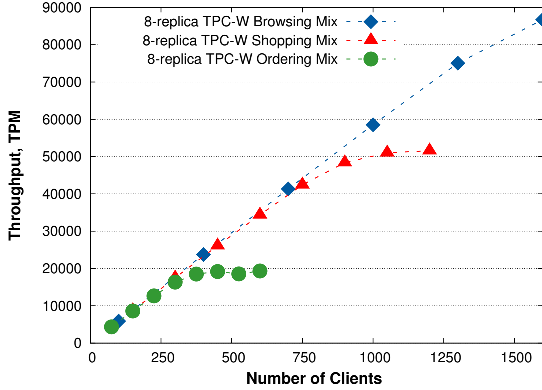

Fig. 6. SGSI Throughput of TPC-W Mixes.

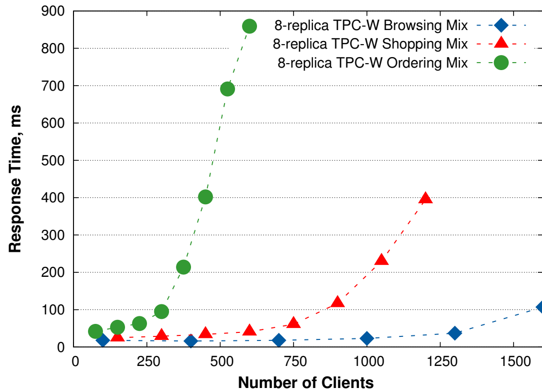

Fig. 7. SGSI Response Time of TPC-W Mixes.

replication may give little performance improvement, or even a degradation, as the number of replicas increases, since the updates need to be certified and applied at every replica.

We use all three TPC-W mixes to characterize the performance of SGSI from read dominated workloads to update intensive workloads. Figures 6 and 7 show throughput and average response times for an 8-replica SGSI system as the number of TPC-W clients varies. The Browsing Mix (5% updates) is read dominated and the figures show that its throughput scales almost linearly up to 88000 TPM, which is 7X the single replica performance. Under the Shopping Mix (20% updates) throughput reaches a peak of 51000 TPM: 4X the single replica performance. The Ordering Mix (50% updates) reaches a throughput of 19500 TPM at 8 replicas, which is 2X the throughput of a single replica. As expected, the benefits of replication depends on the ratio of updates.

F. Abort Analysis via SmallBank

We analyze the impact SGSI on abort rates under varying levels of multiprogramming. The abort rates reported for TPC-W in Section VII-E2 are low and others have reported similar findings [10], [33]. We use the SmallBank benchmark as its workload consists of 80% updates and it supports a variable number of clients. SmallBank was used to evaluate making SI serializable both within, and outside of, SI engines [3], [7].

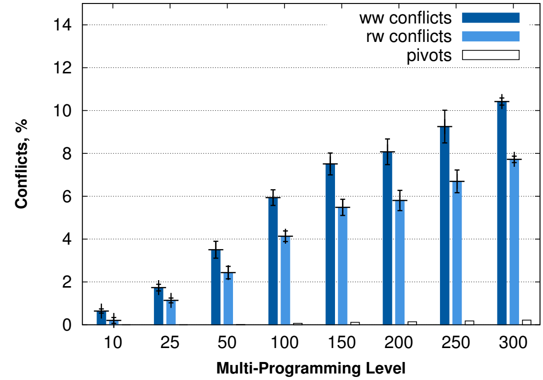

Fig. 8. Conflicts with SmallBank (80% updates).

Figure 8 shows the increase in abort rates as the multiprogramming level increases from 10 to 300. SGSI and GSI both abort ww conflicts, whereas rw conflicts are only aborted under SGSI to guarantee one-copy serializability. The additional aborts from rw conflicts under SGSI are strictly lower than baseline ww conflict aborts. This suggests that SGSI has a slight effect on the observed abort rates. This is important since workloads of SI engines have low abort rates.

When examining the rw conflicts, the issue arises as to the number leading to serialization errors. Testing for serializability is a well-known NP-complete problem, and all practical serializable concurrency control techniques abort transactions that could be serialized. Figure 8 shows the number of pivots as an upper bound on potential serialization errors. A pivot has an incoming and an outgoing rw-dependency in the serializability graph [7] indicating a dangerous structure that could be unserializable [16]. Our results show up to 2.5% of rw conflicts are pivots and suggest that the majority of rw conflicts would be serializable were we to build the multi-version serializability graph and check for cycle absence.

VIII. Related Work

The related work on Snapshot Isolation (SI) and Generalized Snapshot Isolation (GSI) has been discussed in Section II-A. Here we present related work on serializability and replication.

Serializability. Serializability [6], [23] is the most common database correctness criteria. Researchers have investigated guaranteeing serializability in centralized systems both when using weaker isolation levels and when using application specific knowledge. Adya et al. [1] have provided a theoretical foundation to formally specify practical isolation levels for commercial databases. Atluri et al. [4] have studied the serializability of weaker isolation levels such as the ANSI SQL isolation levels for centralized databases. Shasha et al. [28] have presented the conditions that allow a transaction to be divided into smaller sub-transactions that release locks earlier than the original transaction under traditional locking policies.

Serializability under SI. Several researchers [15], [16], [13] have recently demonstrated that, under certain conditions on the workload, transactions executing on a database with SI

produce serializable histories. Serializable Snapshot Isolation (SSI) [7] is a concurrency control mechanism which enforces serializability by correcting anomalies allowed by SI. SSI is implemented inside the database engine and it detects and prevents the anomalies at run time.

Replicated Database Systems. Gray et al. [18] have classified database replication into two schemes: eager and lazy. Eager replication provides consistency at the cost of limited scalability. Lazy replication increases performance by allowing replicas to diverge, exposing clients to inconsistent states of the database. The prototype we use has features from both lazy (e.g., in update propagation) and eager (e.g., no reconciliation is ever needed) systems.

SI Replicated Database Systems. Kemme et al. implement Postgres-R [19], and integrate the replica management with the database concurrency control [21], [33] in Postgres-R(SI). Plattner et al. [24] present Ganymed, a master-slave scheme for replicating snapshot isolated databases in clusters of machines. Tashkent [11], [12] is a replicated database system that provides GSI and Pangea [22] uses eager update propagation. None of these systems guarantees serializability.

SI Federated Database Systems. Schenkel et al. [26] discuss using SI in federated databases where global transactions access data distributed across multiple sites. Their protocols guarantee SI at the federation level rather than serializability. In later work [25], a ticketing algorithm ensures serializability in federated SI databases. Global update transactions are executed sequentially in ticket order. In contrast, SGSI addresses the problem of replication rather than federation, and update transactions execute concurrently on replicas.

IX. Conclusions

This research investigates providing a global isolation level in replicated database systems stronger than the level provided locally by individual replicas. We introduce SGSI, a concurrency control technique that ensures one-copy serializability in replicated systems in which each replica is snapshot isolated. We employ novel techniques to extract transaction readsets and perform enhanced certification. We implement SGSI and build a prototype replicated system and evaluate its performance to show that SGSI is practical. The performance results under the TPC-W workload mixes show that the replicated system performance scales with the number of replicas.

Acknowledgements

We are grateful to Konstantinos Krikellas for his contributions to our experimental system, Tim Harris for his insightful comments, Zografoula Vagena who participated in the early stages of this work, and M. Tamer Özsu for his suggestions. Mohammad Alomari provided us with the small bank benchmark. Fernando Pedone and Willy Zwaenepoel suggested the possibility of building a GSI system that provides 1SR.

References

[1] A. Adya, B. Liskov, and P. E. O'Neil. Generalized Isolation Level Definitions. In *ICDE 2000*.

[2] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proc. 12th Intl. Symposium on Distributed Computing*, 1998.

[3] M. Alomari, M. Cahill, A. Fekete, and U. Rohm. The cost of serializability on platforms that use snapshot isolation. *ICDE 2008*.

[4] V. Atluri, E. Bertino, and S. Jajodia. A Theoretical Formulation for Degrees of Isolation in Databases.

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *SIGMOD*, pages 1–10, New York, NY, USA, 1995. ACM.

[6] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison–Wesley, 1987.

[7] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable Isolation for Snapshot Databases. In *SIGMOD'08*.

[8] K. Daudjee and K. Salem. Lazy Database Replication with Ordering Guarantees. In *ICDE '04*, pages 424–436, 2004.

[9] K. Daudjee and K. Salem. Lazy Database Replication with Snapshot Isolation. In *VLDB '06*, pages 715–726, 2006.

[10] S. Elnikety, S. Dropsho, E. Cecchet, and W. Zwaenepoel. Predicting Replicated Database Scalability from Standalone Database Profiling. In *EuroSys 2009*.

[11] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: Uniting Durability with Transaction Ordering for High-Performance Scalable Database Replication. In *EuroSys 2006*.

[12] S. Elnikety, S. Dropsho, and W. Zwaenepoel. Tashkent+: Memory-Aware Load Balancing and Update Filtering in Replicated Databases. In *EuroSys 2007*.

[13] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database Replication using Generalized Snapshot Isolation. *SRDS 2005*.

[14] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Comm of the ACM 1976*.

[15] A. Fekete. Allocating Isolation Levels to Transactions. In *PODS 2005*.

[16] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making Snapshot Isolation Serializable. *ACM Trans. Db. Sys.*, 30(2), 2005.

[17] A. Fekete, E. O'Neil, and P. O'Neil. A read-only transaction anomaly under snapshot isolation. *SIGMOD Rec.*, 33(3):12–14, 2004.

[18] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *SIGMOD 1996*.

[19] B. Kemme and G. Alonso. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *VLDB 2000*.

[20] K. Krikellas, S. Elnikety, Z. Vagena, and O. Hodson. Strongly Consistent Replication for a Bargain. *ICDE 2010*.

[21] Y. Lin, B. Kemme, M. Patino-Martínez, and R. Jiménez-Peris. Middleware Based Data Replication providing Snapshot Isolation. *SIGMOD 2005*.

[22] T. Mishima and H. Nakamura. Pangea: An eager database replication middleware guaranteeing snapshot isolation without modification of database servers. In *VLDB 2009*.

[23] C. H. Papadimitriou. *The Theory of Database Concurrency Control*. W. H. Freeman & Co., New York, NY, USA, 1986.

[24] C. Plattner and G. Alonso. Ganymed: Scalable Replication for Transactional Web Applications. *Middleware 2004*.

[25] R. Schenkel and G. Weikum. Integrating Snapshot Isolation into Transactional Federation. In *CooplS 2002*.

[26] R. Schenkel, G. Weikum, N. Weienberg, and X. Wu. Federated transaction management with snapshot isolation. In *8th Int. Workshop on Foundations of Models and Languages for Data and Objects - Transactions and Database Dynamics*, 1999.

[27] F. B. Schneider. *Replication Management using the State-Machine Approach*. ACM Press/Addison-Wesley Publishing Co., 1993.

[28] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction Chopping: Algorithms and Performance Studies. *ACM Trans. Db. Sys.*, 20(3).

[29] M. Stonebraker. The design of the postgres storage system. *VLDB 1987*.

[30] The Transaction Processing Council (TPC). The TPC-W Benchmark. http://www.tpc.org/tpcw.

[31] W. Weihl. Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types. *ACM Trans. Prog. Lang. Syst.*, 11(2), 1989.

[32] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[33] S. Wu and B. Kemme. Postgres-R(SI): Combining Replica Control with Concurrency Control Based on Snapshot Isolation. In *ICDE 2005*.