Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 13-021

Transaction Management using Causal Snapshot Isolation in Partially Replicated Databases

Vinit Padhye and Anand Tripathi

July 24, 2013

Transaction Management using Causal Snapshot Isolation in Partially Replicated Databases

Vinit Padhye and Anand Tripathi
Department of Computer Science
University of Minnesota Minneapolis, 55455 Minnesota USA
Email: (padhye,tripathi)@cs.umn.edu

***Abstract*—We present here a transaction management protocol using snapshot isolation in partially replicated multi-version databases. We consider here replicated databases consisting of multiple disjoint data partitions. A partition is not required to be replicated at all database sites, and a site may contain replicas for any number of partitions. Transactions can execute at any site, and read or write data from any subset of the partitions. The updates in this model are performed using asynchronous update propagation. The protocol ensures that the snapshot observed by a transaction contains data versions that are causally consistent. In developing this protocol, we address the issues that are unique in supporting transactions with causal consistency together with the snapshot isolation model in partially replicated databases. Such a model is attractive in geo-scale systems because of its support for partial replication, use of causal consistency model which does not require a global sequencer, and asynchronous propagation of updates.**

I. Introduction

Providing transaction support is an important requirement of a database system, since transaction semantics are often needed in many applications for atomically executing a sequence of operations. In large-scale systems, the database is typically replicated across multiple sites/hosts, which may be distributed geographically. Database replication poses fundamental trade-offs between data consistency, scalability, and availability. Synchronous replication, in which the updates of a transaction are propagated synchronously to other sites before committing the transaction, provides strong consistency but incurs high latencies for transactions. Moreover, this may not be practical under wide-area settings [1], [2]. In asynchronous replication, the transaction is first committed locally and then its updates are asynchronously propagated at a later time. Asynchronously replicated systems provide lower latencies in transaction execution, high availability, but provide only eventual consistency [1] or causal consistency [3], [4], [5]. Furthermore, in this model because the data read by transactions may not be current, a validation phase is typically required during transaction commit.

In this paper, we address the problem of providing transaction support for partially replicated databases which use asynchronous replication. Partial replication is useful for scalability since the data items need not be replicated at all sites and thus the updates need to be propagated only to the sites replicating the updated data items. There can also be other reasons for partial replication such as replicating data only at the sites in the geographic regions where data is likely to be accessed most frequently. Our goal is to provide causal consistency of data under partial replication. Causal consistency provides more useful semantics than eventual consistency and can be supported under asynchronous replication and even under network partitions.

In our earlier work [4], we have presented a transaction model, called *Causally-coordinated Snapshot Isolation (CSI)*, which provides snapshot isolation (SI) [6] based transactions with causal consistency of data for asynchronously replicated databases. This model builds upon the approach presented in [3], but eliminates false causal dependencies. The CSI model does not enforce a total ordering of transactions across sites, since implementing it requires a global sequencer mechanism, which can become a scalability bottleneck and is not practical under wide-area settings. The CSI model provides total ordering of transactions only within a site, and across sites it provides causal ordering of transactions. We define the causal ordering ($\prec$) between two transactions as follows. Transaction T_i casually precedes transaction T_j ($T_i \prec T_j$), or in other words T_j is causally dependent on T_i, if T_i and T_j are not concurrent and if T_j reads any of the updates made by by T_i or creates a newer version for any of the items modified by T_i. Also, this relationship is transitive, i.e. if $T_i \prec T_j$ and $T_j \prec T_k$, then $T_i \prec T_k$. The causal ordering defines a partial ordering over transactions.

The CSI model guarantees that a transaction observes a *consistent snapshot* which has the properties of *atomicity* and *causality* as defined below.

- *Atomicity*: If the transaction sees an update made by a transaction T_i then all other updates of T_i are also visible in its snapshot.
- *Causality*: The database state observed by the transaction is consistent with causal ordering of transactions, i.e. if the transaction sees effects of transaction T_i, then effects of all the transactions causally preceding T_i are also visible in its snapshot.

Moreover, the CSI model ensures that when two or more concurrent transactions update a common data item, only one of them is allowed to commit. The snapshot observed by a transaction may not always reflect the latest versions of the accessed items, but it is guaranteed to be consistent.

The CSI model assumes database is fully replicated, i.e. data items are replicated at all sites. In this paper, we present a transaction model, referred to as the *Partitioned-CSI (P-CSI)*

which extends the CSI model for partially replicated database systems. We consider a replication model in which database is partitioned in multiple disjoint partitions and each partition is replicated at one or more sites, and a site may contain any number of partitions. A transaction may be executed at any site and may access items in any of the partitions. The P-CSI model provides for a partitioned database all the guarantees provided by the CSI model, as described above, These guarantees are preserved even when a transaction accesses items belonging to multiple partitions stored at different sites.

In this paper, we present a protocol which preserves the causal guarantees noted above and requires communicating updates only to the sites storing the updated data items. Towards this goal, we address a number of issues that arise due to partial replication with asynchronous update propagation. The first set of issues are related to ensuring that a transaction observes a *consistent snapshot* under the asynchronous update propagation model. The second set of issues that we address is related to executing a transaction that involves accessing partitions stored at different site. When executing such a transaction, we must ensure that the snapshots used for accessing different partitions together form a globally consistent snapshot with respect to the *atomicity* and *causality* properties described above. The P-CSI model ensures these properties.

We elaborate the above issues and describe the P-CSI model and its transaction management protocol. We also discuss the correctness of the protocol in ensuring the properties noted above. We implemented a prototype system for evaluating the P-CSI model. Our evaluations demonstrate the scalability of the P-CSI model and its performance benefits over the full replication model.

The rest of the paper is organized as follows. In the next section we discuss the related work. In Section III, we give a brief overview of the CSI model. Section IV highlights the problems arising in supporting transactions with causal consistency in partial replication. In SectionV, we describe the P-CSI model. Evaluations of the proposed models and mechanisms are presented in Section VI. Conclusions are presented in the last section.

II. Related Work

The problem of transaction management in replicated database systems has been studied widely in the past. Initial work on this topic focused on supporting transactions with *1-copy serializability* [7]. Transaction execution models in replicated systems broadly fall into two categories: symmetric execution model where transactions can be executed at any site containing a replica, or asymmetric model where the transaction is executed only on some designated sites. In cluster-based environments, approaches for data replication management have mainly centered around the use of the state machine model [8], [9] using atomic broadcast protocols. Examples of such approaches include [10], [11], [12], [13], [14], [15], [16], [17].

The issues with scalability in data replication with strong consistency requirements are discussed in [18]. Such issues can become critical factors for data replication in large-scale systems and geographically replicated databases. This has motivated use of other models such as snapshot isolation (SI) [6] and causal consistency. The snapshot isolation model is a multi-version scheme in which a transaction reads only committed version of data. Thus read-only transactions never abort. An update transaction is committed only if no concurrent committed transaction has modified any of the items in its write-set. This requires execution of a validation phase at the commit time. The validation phase is required to be executed in a sequential order by concurrent transactions.

Replication using snapshot isolation (SI) has been studied widely in the past [19], [14], [20], [10], [21], [22], [23], [16]. SI-based database replication using lazy replication approach is investigated in [22], however that approach used the primary-backup model. Compared to primary-backup model, the symmetric execution model is more flexible but requires some coordination among replicas to avoid update conflicts. Many of the systems for SI-based database replication [14], [19], [20], [24] based on this model use eager replication with atomic broadcast to ensure that the replicas observe a total ordering of transactions. The notion of *1-copy snapshot isolation* in replicated database systems is introduced in [14], which means that the schedule of transaction execution at different replicas under the read-one-write-all (ROWA) model are equivalent to an execution of the transactions using the SI model in a system with only one copy.

Recently, many data management systems for cloud datacenters distributed across wide-area have been proposed [1], [2], [25], [5], [3]. Dynamo [1] uses asynchronous replication with eventual consistency and do not provide transactions. PNUTS [2] also does not provide transactions, but provides a stronger consistency level than eventually consistency, called as *eventual timeline consistency*. Megastore [25] provides transactions over a group of entities using synchronous replication. COPS [5] provides causal consistency, but does not provide transaction functionality, except for snapshot-based read-only transactions. PSI [3] provides transaction functionality with asynchronous replication and causal consistency. The CSI model, which builds upon the PSI model, provides a more efficient protocol for ensuring causal consistency by eliminating false causal dependencies.

Another approach for achieving higher scalability is to use partial replication instead of replicating the entire database on all sites [26], [27], [28], [29], [30], [31]. The approach presented in [26] guarantees serializability. It uses epidemic communication that ensures causal ordering of messages using the vector clock scheme of [32], where each site knows how current is a remote site's view of the events at all other sites. Other approaches [27], [30], [33], [34] are based on the database state machine model [8], utilizing atomic multicast protocols. The notion of *genuine partial replication* introduced in [34] requires that the messages related to a transaction should only be exchanged between sites storing the items accessed by the transaction. These approaches support 1-copy-serializability. In contrast, the approach presented in [31] is

based on the snapshot isolation model, providing the guarantee of 1-copy-SI. This model is applied to WAN environments in [35] but relies on a single site for conflict detection in the validation phase. The system presented in [33] uses the notion of generalized snapshot isolation (GSI) [20], where a transaction can observe a consistent but old snapshot of the database.

In contrast to the above approaches, the P-CSI model presented here provides a weaker but useful model of snapshot isolation, based on causal consistency. Because it does not require any total ordering of transactions during validation, it provides greater concurrency towards scalability. The P-CSI model presented here avoids sending update messages to the sites that do not contain any of the accessed and modified partitions and thus it eliminates some of the issues raised in regard to the use of causal consistency [36] model in replicated systems.

III. Overview of the CSI Model

We present here a brief overview of the Causally-coordinated SI model. The details of this model are presented in [4]. This model forms the basis for the development of the P-CSI protocol.

A. System Model

The system consists of multiple database sites and each site is identified by a unique $siteId$. Each site has a local database that supports multi-version data management and transactions. Data items are replicated at all the sites. For each data item, there is a designated *conflict resolver site* which is responsible for checking for update conflicts for that item. Transactions can execute at any site. Read-only transactions can be executed locally without needing any coordination with remote sites. Update transactions need to coordinate with conflict resolver sites for update conflict checking for the items in their write-sets.

B. CSI Model

As noted earlier, the total ordering on transactions is not enforced by the CSI model. This eliminates the need of a global sequencer. Instead, a transaction is assigned a commit sequence number seq from a monotonically increasing local sequence counter maintained by its execution site. Thus, the commit timestamp for a transaction is a pair $<$*siteId,seq*$>$. Similarly, a data item version is identified by a pair $<$*siteId, seq*$>$. The local sequence number is assigned only when the transaction is guaranteed to commit, i.e. only when there are no update conflicts. Thus, there are no gaps in the sequence numbers of the committed transactions. A transaction first commits locally and then its updates are propagated to other sites asynchronously. A remote site, upon receiving a remote transaction's updates, applies the updates provided that it has also applied updates of all the causally preceding transactions. The updates of the transactions from a particular site are always applied in the order of their sequence numbers, i.e. a transaction with sequence number n from site i is applied only when all the preceding transactions from site i with sequence number up to n-1 are applied. All the updates of a transaction are applied to the local database as an atomic operation, which also includes updating a local vector clock.

Each site maintains a vector clock [37], [38], which we denote by $\mathcal{V}$, indicating the updates of the transactions from other sites that it has applied to the local database. Thus, a site i maintains a vector clock $\mathcal{V}$, where $\mathcal{V}[j]$ indicates that site i has applied the updates of all transactions from site j up to this timestamp, moreover, site i has also applied all the other updates that causally precede these transactions. In the vector clock, $\mathcal{V}[i]$ is set to the sequence number of the latest transaction committed at site i.

Snapshot-based access: A transaction t executing at site i is assigned, when it begins execution, a *start snapshot timestamp* $\mathcal{S}_t$, which is set equal to the current vector clock $\mathcal{V}$ value of site i. When t performs a read operation for item x, we determine the latest version of x that is visible to the transaction according to its start snapshot timestamp. Note that, because there is no total ordering, the versions can not be compared based on the timestamps assigned to the versions. Instead the versions are ordered based on the order in which they are applied. Thus, for a data item, the most recently applied version indicates the latest version of that item. Recall that the causal consistency property described above ensures that a data item version is applied only when all the preceding versions are applied. For each data item x, we maintain a version log which indicates the order of the versions. When t performs a read operation on x, we check for every version $< j,n>$, starting from the version that is applied most recently, if the version is visible in the transaction's snapshot or not, i.e. if $\mathcal{S}_t[j] \geq n$. We then select the latest version that is visible in t's snapshot. When t performs a write operation, writes are kept in a local buffer until the commit time.

Commit protocol: If t has modified one or more items, then it performs update conflicts checking using a two-phase commit (2PC) protocol with the conflict resolver sites responsible for those items. In the prepare message to each site, t sends $\mathcal{S}_t$ and the list of items it has modified for which that site is the conflict resolver. Each site checks, if the latest versions of those items are visible in t's snapshot and that none of the items is locked by any other transaction in its conflict detection step. If this check fails, then the resolver sends a 'no' vote. Otherwise, it locks the corresponding items and sends a 'yes' vote. If t receives 'yes' votes from all conflict resolvers, t is assigned a monotonically increasing local sequence number by t's local site. First, t commits locally, applying the updates to the local database. The local site's vector clock is advanced appropriately. It now sends a commit message, containing the sequence number, to all the conflict resolvers. Otherwise, in case of any 'no' vote, t is aborted and an abort message is sent to all the conflict resolvers. Upon receiving a commit or abort message, a conflict resolver releases the locks, and in case of commit it records the new version number as a 2-tuple: ($<siteId, seq>$). After performing these operations, the local site asynchronously propagates t's updates to all the

other sites. If all the items that t has modified have local site as the conflict resolver then t's validation can be performed entirely locally.

Update propagation: For ensuring causal consistency, t's updates are applied at remote sites only after the updates of all the causally preceding transactions have been applied. For update propagation, we define the *effective causal snapshot*, which indicates, for each site, the latest event from that site which is 'seen' by the transaction based on the items it read or modified. In contrast to the PSI model [3] the approach taken in the CSI model avoids false causal dependencies. In other words, we capture causal dependencies with respect to a transaction rather than a site. The effective causal snapshot for a transaction t, executed at a site i is defined as a vector timestamp denoted by $\mathcal{E}_t$, and is determined as follows. $\mathcal{E}_t[i]$ is set equal to n-1 where n is t's sequence number. This indicates that t can be applied only when the transaction immediately preceding t at site i is applied. The other elements of $\mathcal{E}_t$, i.e. those corresponding to the remote sites, are determined as follows:

$$\forall j; j \neq i : \mathcal{E}_t[j] = max\{seq \mid \forall\, x\ s.t. (x \in readset(t) \vee x \in prevwrites(t)) \wedge (version(x) =< j, seq >)\}$$

Here, $prevwrites(t)$ is the set of latest versions visible at that site for the items in the write-set of t. If this information about the write-set is not included, then it may happen that for a particular data item x modified by t, a remote site may store the version created by t without having all the preceding versions of x. We call it the *missing version* problem. This can violate the basic multi-version semantics of snapshot-based access in cases such as time-travel queries, which read from a specific older snapshot. It also complicates the version ordering logic described above. It should also be noted that the *effective causal snapshot* vector for a transaction is determined at the end of the transaction execution, and therefore the information about the read/write sets is needed only after the transaction execution has finished.

The update propagation protocol uses the $\mathcal{E}_t$ value of transactions while propagating their updates. Upon receiving updates, the remote site compares its vector clock with $\mathcal{E}_t$ vector to ensure that an update is applied at that site only when it has seen all the causally preceding updates. On applying the updates, the vector clock of the site is advanced appropriately.

IV. Issues in supporting causal consistency under partial replication

We now discuss the issues in extending the CSI model to provide causal consistency under partial replication. In a partial replication scheme, data items are not replicated at all sites. Ideally, one of the advantage of partial replication scheme is that the updates of a transaction need to be propagated only to the sites that store the data items accessed by the transaction. This reduces the communication cost compared to the full replication scheme as the updates are not required to be propagated to all sites. However, propagating a transaction's updates only to the sites containing the items modified by the transaction raises issues with respect to supporting causal consistency.

Ensuring causality guarantees requires that for applying a given transaction's updates to a partition replica at a site, all its causally preceding events, including the transitive dependencies, must be captured in the views of the local partitions of that site. We illustrate this problem using the example scenario shown in Figure 1. In this example, partition $P1$ containing item x is replicated at sites 1 and 3. Partition $P2$ containing item z is replicated at sites 1, 2, and 3. Partition $P3$ containing y is replicated at sites 2 and 4. Transaction $T1$ executed at site 1 updates item x and creates version $x(100)$. This update is asynchronously propagated to site 3, shown by a dashed arrow in the figure. Transaction $T2$ executed at site 2 reads $x(100)$ from partition $P1$ at site 1 and updates y to create version $y(200)$. Later transaction $T3$ is executed at site 2, which reads $y(200)$ and modifies z to create version $z(300)$. Note that version $z(300)$ causally depends on $x(100)$. The update of $T3$ is propagated asynchronously to sites 1 and 3. Suppose the update of $T3$ for $z(300)$ arrives at site 3 before the update of transaction $T1$ for $x(100)$. In case of full replication using the CSI model, all transactions' updates are sent to all sites and the update of $T3$ in the above scenario would only be applied after the updates of transaction $T1$ and $T2$ are applied. However, with partial replication shown in Figure 1, the updates of $T2$ would never be sent to site 3. Therefore, we need update synchronization mechanism that selectively waits for the updates of transaction $T1$ but not $T2$. Applying the update $z(300)$ before applying the update of $x(100)$ will result in causally inconsistent state of partitions $P1$ and $P2$ at site 3.

A straightforward solution for supporting causal consistency requires either (1) maintaining the entire causal dependencies graph for every item version [5], or (2) communicating every update to all the sites in the system so that each site is cognizant of all causal dependencies for a data item version. The first solution is not feasible since the causal dependency graph can potentially become very large. The second solution nullifies the advantage of partial replication, since it requires communicating the updates to all sites [36]. The proposed P-CSI model presents a more efficient protocol which ensures causal consistency and requires propagating the transaction's updates only to the sites storing the items modified by the transaction.

Next we illustrate the issues that arise when executing a transaction that needs to access some partitions stored at a remote site. Suppose, in the example shown in Figure 1, at site 4 transaction $T4$ is executed which reads items x, y, and z. This transaction reads $y(199)$ from the local partition $P3$ and reads $x(100)$ and $z(300)$ from site 1 since site 4 does not contain partitions $P1$ and $P2$. In Figure 2, we show the causal relationships between the versions of items x, y, z. We also show the snapshot observed by $T4$, which is causally inconsistent because it contains $z(300)$ but not the causally preceding version $y(200)$.

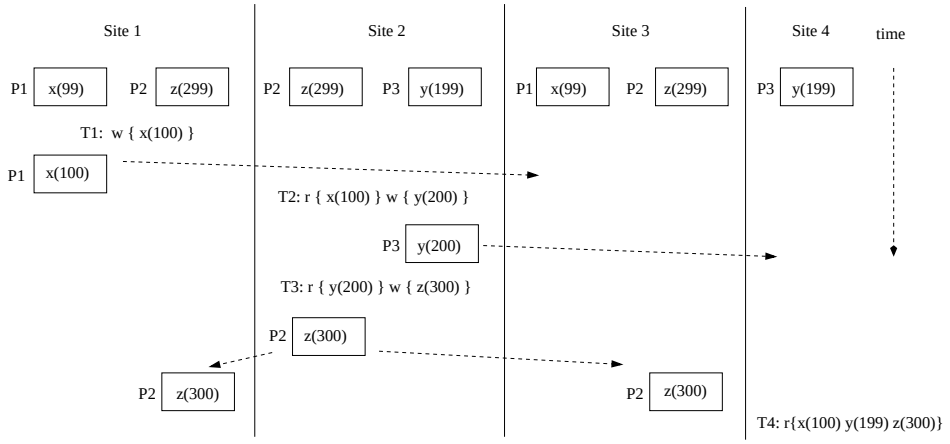

Fig. 1. Issues in supporting causality under partial replication

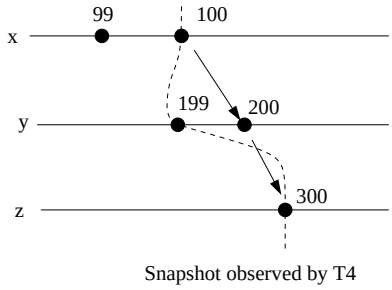

Fig. 2. Causally inconsistent snapshot due to remote reads

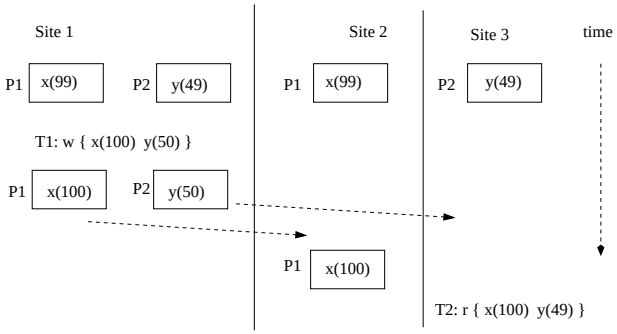

Fig. 3. Issues in obtaining atomically consistent snapshot

Another issue that arises when reading data from remote partitions under asynchronous propagation is related to the atomicity property of consistent snapshots. We illustrate this with an example shown in Figure 3. Here partition $P1$ containing item x is replicated at sites 1 and 2, and partition $P2$ containing y is replicated at sites 1 and 3. Transaction $T1$ executed at site 1 updates x and y, creating versions $x(100)$ and $y(50)$. The updates of this transaction are propagated asynchronously to sites 2 and 3. Suppose that site 3 executes transaction $T2$ which reads item x and y. $T2$ is executed before site 3 applies the update of $T1$ for version $y(50)$. $T2$ reads $y(49)$ from its local partition and reads $x(100)$ from site 1. This reflects an atomically inconsistent snapshot of partitions $P1$ and $P2$ with respect to items x and y. In the next section, we present the P-CSI model to address such issues in ensuring causal consistency.

V. Partitioned-CSI Model

A. System Model

We consider partial replication of database that consists of a finite set of data items partitioned into multiple disjoint data partitions. The system consists of multiple sites, and each site contains one or more partitions. Each partition is replicated across one or more sites. Each site supports multi-version data management and SI-based transactions.

As in the case of CSI, for each data item, there is a designated *conflict resolver site* which is responsible for checking for update conflicts for that item. Transactions can execute at any site. A transaction may access items in multiple partitions. For partially replicated databases, the P-CSI model ensures all the guarantees provided by CSI. These properties are ensured even when a transaction accesses items from multiple sites.

B. Vector Clocks and Partition Dependency View

Our goal in designing the P-CSI model is to avoid the need of propagating updates to all sites. Our solution to this problem is based on maintaining vector clocks on per-partition basis. In the following discussion, we refer to the partitions stored at a site as the *local partitions* of that site. In the P-CSI model, each site maintains a vector clock for each local partition, referred to as the *partition view ($\mathcal{V}_p$)*. The *partition view* $\mathcal{V}_p$ for a local partition p maintained by site j indicates the sequence numbers of transactions from all sites that have updated any of the items in partition p and have been applied to the local partition at site j. Thus, the value $\mathcal{V}_p[k]$ indicates that site j has applied all the transactions pertaining to partition p from site k up to this much timestamp as well as all the causally preceding transactions. The $\mathcal{V}_{\mathcal{P}_i}$ value at a particular time identifies the state of the partition p visible at site j at that time. A site also

maintains a sequence counter for each of its local partitions, which is used to assign sequence numbers to local transactions modifying items in that partition. A transaction executing at a site obtains, during its commit phase, a local sequence number for each partition it is modifying.

Events of interest in the system are the update events performed by a transaction. A transaction may update multiple partitions thus resulting in distinct update events in different partitions. We define an *atomic event set* as the set of all update events of a given transaction. In any consistent snapshot either all or none of the events of an atomic event set are present.

A site also maintains for each local partition p a *partition dependency view ($\mathcal{D}_p$)* which is a set of vector clocks. It represents a consistent global snapshot, capturing both atomicity and causality properties. For causality, it identifies for each of the other partitions the events that have occurred in that partition and that causally precede the partition p's state as identified by its current partition view. In other words, $\mathcal{D}_p$ indicates the state of other partitions on which the state of partition p is causally dependent. For atomicity, it captures the atomic event sets of all the transactions applied to partition p. The *partition dependency view* $\mathcal{D}_p$ consists of a vector clock for each other partition. Formally, $\mathcal{D}_p$ is a set of vector clocks $\{\mathcal{D}_p^1, \mathcal{D}_p^2, \cdots, \mathcal{D}_p^q, \cdots, \mathcal{D}_p^n\}$, in which an element $\mathcal{D}_p^q$ is a vector clock corresponding to partition q. Each element of the vector clock $\mathcal{D}_p^q$ identifies the transactions performed on partition q that causally precede the transactions performed on partition p identified by $\mathcal{V}_p$. Note that partition q may or may not be stored at site j. Also, note that vector clock $\mathcal{D}_p^p$ is the same as $\mathcal{V}_p$. A site maintains partition dependency views for each local partition. We describe below how the partition dependency views are maintained and how they are used to ensure causal consistency.

C. Transaction Execution Protocol

We now give a brief overview of the transaction execution model of P-CSI and then discuss the various issues involved in it. For simplicity of the discussion, we first describe the execution of a transaction at a site which stores all the partitions accessed by the transaction. Later, we discuss how a transaction that requires accessing partitions at remote sites is executed. A transaction t goes through a series of execution phases. In the first phase it obtains a *start snapshot time ($\mathcal{S}_t$)*, which is a set of vector clocks corresponding to the partitions to be accessed by the transaction. An element $\mathcal{S}_t^p$ in $\mathcal{S}_t$ indicates the start snapshot time for partition p. The transaction then performs read operations using the start snapshot time. P-CSI protocol ensures that the snapshot observed by a transaction is causally consistent.

When the transaction reaches its commit point it needs to coordinate with the conflict resolver sites of the items in its write-set, as in the CSI protocol, to check for update conflicts. Read-only transactions can be executed locally without needing any coordination with remote sites. After successful validation, the transaction executes its commit phase. In the commit phase, the transaction obtains a sequence number for each partition that it modified, from the local site. This sequence number is used to assign a timestamp to the transaction for that partition. A timestamp is defined as a pair $<siteId, seq>$, where seq is a local sequence number assigned to the transaction by the site identified by $siteId$. The *commit timestamp vector*($\mathcal{C}_t$) of transaction t is a set of timestamps assigned to the transaction corresponding to the partitions modified by the transaction. Thus, the commit timestamp vector $\mathcal{C}_t$ of transaction t modifying n partitions is a set of timestamps $\{\mathcal{C}_t^1, \mathcal{C}_t^2, \cdots, \mathcal{C}_t^q, \cdots, \mathcal{C}_t^n\}$. For an item modified by transaction t in partition q, the version number of the item is identified by commit timestamp $\mathcal{C}_t^q$.

A transaction's updates are applied to the local database and then asynchronously propagated to other sites which store any of the partitions modified by the transaction. The information transmitted to sites in the update propagation message includes a set of vector clocks, called *transaction dependency view* ($\mathcal{TD}$), containing a vector clock corresponding to each partition. The transaction dependency view for a transaction t identifies all the transactions that causally precede t. At the remote site, the updates of transaction t are applied only when all the events identified in $\mathcal{TD}_t$ have been applied. We describe the details of the update propagation later.

We describe below in details the various steps in the transaction execution protocol.

Obtaining start snapshot time: In the case when all partitions to be accessed by the transaction are local, the start snapshot time $\mathcal{S}_t$ for transaction t is obtained using the partition view values for those partitions. The pseudocode for obtaining start snapshot time is shown in Algorithm 1. Later in Algorithm 7 we generalize this for a transaction accessing partitions from remote sites.

Algorithm 1 Obtaining start snapshot time

```
function GETSNAPSHOT
    P ← partitions accessed by the transaction
    [ begin atomic action
    for each p ∈ P do
        S_t^p ← V_p
    end atomic action ]
```

Algorithm 2 Performing read operations

```
function READ(item x)
    p ← partition containing item x
    /* performed in reverse temporal order of versions */
    for each version v ∈ version log of x do
        if S_t^p[v.siteId] ≥ v.seqno then return v.data
```

Performing read operations: When a transaction reads a data item x from a partition p, we determine the latest version for x that must be visible to the transaction based on the transaction's start snapshot time $\mathcal{S}_t^p$ for that partition. The procedure to read a data item version from partition p according to transaction's snapshot $\mathcal{S}_t^p$ is the same as in the CSI

model. Algorithm 2 gives the pseudocode for performing read operations. In case of write operations, the writes are buffered locally until the commit time.

Algorithm 3 Update conflict checking performed by a transaction at site j

function CHECKCONFLICTS($writeset$)
  $sites \leftarrow$ conflict resolver sites for items $\in writeset$
  for each $s \in sites$ **do**
    $itemList \leftarrow \{x | x \in writeset \wedge resolver(x) = s\}$
    send prepare message to s : ($itemList$, $\mathcal{S}$)
  if all votes are 'yes' **then**
    perform *Commit* function as shown in Algorithm 5 to each $s \in sites$
      send commit message: ($itemList$, $\mathcal{C}_t$)
  else
    send abort message to each $s \in sites$
    abort transaction

/* Functions executed by the conflict resolver site */

/* upon receiving prepare message for transaction t */
function RECVPREPARE($itemList$, $\mathcal{S}_t$)
  for each $x \in itemList$ **do**
    $p \leftarrow$ partition containing item x
    $v \leftarrow$ latest version of item x
    if $\mathcal{S}_t^p[v.siteId] \geq v.seqno \wedge x$ is unlocked **then**
      lock x
    else
      return response with 'no' vote
  if all $x \in itemList$ are locked then send response with 'yes' vote

/* upon receiving commit message for transaction t */
function RECVCOMMIT($itemList$, $\mathcal{C}_t$)
  for each $x \in itemList$ **do**
    $p \leftarrow$ partition containing item x
    record version timestamp $\mathcal{C}_t^p$ in version log
    release lock on x

/* upon receiving abort message for transaction t */
function RECVABORT($itemList$)
  release locks on all $x \in itemList$

Update conflict checking: When the transactions reaches its commit point, it performs update conflict checking if it has modified any items. The procedure to check for update conflicts is same as described in Section III for the CSI model. Algorithm 3 shows the protocol for conflict checking.

Determining transaction dependencies: After successful validation, transaction t computes its *transaction dependency view* $\mathcal{TD}_t$. $\mathcal{TD}_t$ is a set of vector clocks $\{\mathcal{TD}_t^1, \mathcal{TD}_t^2, \cdots, \mathcal{TD}_t^q, \cdots, \mathcal{TD}_t^n\}$, in which an element $\mathcal{TD}_t^q$ identifies the transactions performed on partition q which causally precede t. Algorithm 4 shows the pseudocode for computing $\mathcal{TD}_t$. Transaction t computes its effective causal snapshot vector $\mathcal{E}_t$, which is a set containing vector clocks for each partition accessed by t. Element $\mathcal{E}_t^p$ in this set is the effective causal snapshot corresponding to partition p and is computed as discussed in Section III, considering the items read/written by t from partition p. $\mathcal{E}_t$ captures the causal dependencies for transaction t, solely based on items read or written by t. To capture transitive dependencies, we include in $\mathcal{TD}_t$ the *partition dependency view* vectors of each partition accessed by t. If any two partitions $p1$ and $p2$ accessed by t each have in their dependency views an element for some other partition q, i.e. $\exists q$ s.t. $\mathcal{D}_{p1}^q \in \mathcal{D}_{p1} \wedge \mathcal{D}_{p2}^q \in \mathcal{D}_{p2}$, then we take element-wise max value from $\mathcal{D}_{p1}^q$ and $\mathcal{D}_{p2}^q$ using the 'super' function shown in Algorithm 4.

Algorithm 4 Computing transaction dependency view for transaction t

function COMPUTETRANSACTIONDEPENDENCY
  $\mathcal{P} \leftarrow$ set of partitions on which t performed any read/write operation.
  $\mathcal{E}_t \leftarrow$ set of effective causal snapshot vectors corresponding to partitions in $\mathcal{P}$
  for each $p \in \mathcal{P}$ **do**
    $\mathcal{TD}_t^p \leftarrow \mathcal{E}_t^p$
  [begin atomic action
  for each $p \in \mathcal{P}$ **do**
    for each $\mathcal{D}_p^q \in \mathcal{D}_p$ **do**
      if $\mathcal{TD}_t$ does not contain element for q **then**
        $\mathcal{TD}_t^q \leftarrow \mathcal{D}_p^q$
      else
        $\mathcal{TD}_t^q \leftarrow$ super($\mathcal{D}_p^q$, $\mathcal{TD}_t^q$)
  end atomic action]

function SUPER($V1$, $V2$,$\cdots$,Vk) **returns** V
  $\forall i$, $V[i] = max(V1[i], V2[i], \cdots, Vk[i])$

Commit phase: Algorithm 5 shows the commit protocol for transaction. A commit timestamp vector $\mathcal{C}_t$ is assigned to the transaction by obtaining a sequence number of each partition modified by t. The updates made by transactions are written to the local database and transaction dependency view set $\mathcal{TD}_t$ is computed. The partition views $\mathcal{V}$ and dependency views $\mathcal{D}$ of all updated partitions are advanced using $\mathcal{TD}_t$ and $\mathcal{C}_t$, as shown in the function 'AdvanceVectorClocks'. If the committed transaction involves modifying items in multiple partitions, then the above procedure ensures that the partition dependency view $\mathcal{D}$ for each modified partition is updated using the $\mathcal{C}_t$ value to capture all events in the atomic set of the transaction. The above procedure is done as a single atomic action to ensure that the transaction's updates are made visible atomically. The updates, along with $\mathcal{TD}_t$ and $\mathcal{C}_t$ values, are propagated to every site that stores any of the partitions modified by t. The update propagation can be started asynchronously once the $\mathcal{TD}_t$ and $\mathcal{C}_t$ have been computed.

Algorithm 5 Commit Protocol for transaction at site j

```
/* [..] denotes an atomic region */
function COMMIT(writeset)
    P ← partitions pertaining to writeset.
    for each p ∈ P do
        ctr_p ← local sequence counter for partition p
        C_t^p.seq ← ctr_p++
    ApplyUpdates(writeset, C_t)
    // compute dependencies as shown in Algorithm 4
    TD_t ← ComputeTransactionDependency()
    // advance local vector clocks
    AdvanceVectorClocks(TD_t, C_t)
    /* propagate updates */
    propagate to every site that stores any partition p ∈ P
    (TD_t, writeset, C_t)

function APPLYUPDATES(writeset, C_t)
    P ← partitions pertaining to writeset.
    for each p ∈ P do
        for each item x in writeset pertaining to p do
            write the new version of x to the local database.
            record version timestamp C_t^p in version log

/* Function to update vector clocks for partitions */
function ADVANCEVECTORCLOCKS(TD_t, C_t)
    P ← partitions pertaining to writeset
    [ begin atomic region
    for each p ∈ P do
        for each TD_t^q ∈ TD_t s.t. q ≠ p do
            D_p^q ← super(TD_t^q, D_p^q)
        /* Advance D to capture the t's update events in
        other partitions */
        for each C_t^q ∈ C_t s.t. q ≠ p do
            D_p^q[C_t^q.siteId] ← C_t^q.seq
        V^p[C_t^p.siteId] ← C_t^p.seq
    end atomic region ]
```

Algorithm 6 Applying updates at a remote site k

```
function RECVUPDATEPROPAGATION(TD_t, writeset, C_t)
    /*check if the site is up to date with respect to TD_t */
    for each TD_t^p ∈ TD_t do
        if (p is local partition) ∧ V^p < TD_t^p then
            buffer the updates locally
            synchronize phase: either pull the
            required causally preceding updates or
            wait till the vector clock advances enough.
    for each C_t^p ∈ C_t do
        if (p is local partition) ∧ V^p[C_t^p.siteId] < C_t^p.seq−1
        then synchronize phase as shown above
    // apply updates to local partitions at site k
    ApplyLocalUpdates(writeset, C_t)
    // advance vector clocks of site k
    AdvanceVectorClocks(TD_t, C_t)
```

Applying updates at remote sites: When a remote site k receives update propagation for t, it checks if it has applied, to its local partitions, updates of all transactions that causally precede t and modified any of its local partitions. Thus, for every partition p specified in $\mathcal{TD}_t$, if p is stored at site k, then site checks if its partition view $\mathcal{V}_p$ for that partition is advanced up to $\mathcal{TD}_t^p$. Moreover, for each of the modified partitions p for which the remote site stores a replica of p, the site checks if $\mathcal{V}_p$ of the replica contains all the events preceding the sequence number value present in $\mathcal{C}_t^p$. If this check fails the site defers the updates locally and enters a synchronization phase which includes either pulling the required updates or delaying the application of updates until the vector clocks of the local partitions advance enough. If this check is successful, the site applies the updates to the corresponding local partitions. Updates of t corresponding to any non-local partitions are ignored. The partition views at site k are advanced as shown in procedure 'AdvanceVectorClock' in Algorithm 5.

D. Execution of Multi-site Transactions

It is possible that a site executes a transaction that accesses some partitions not stored at that site. This requires reading/writing items from remote site(s). One important requirement while performing a multi-site transaction is to ensure that the transaction observes a consistent global snapshot.

We describe how the start snapshot vector is determined. The Algorithm 7 shows the modified 'GetSnapshot' function. Note that at a given site the partition dependency view of any partition reflects a consistent global snapshot. One can thus simply take the $\mathcal{D}$ vector set of any one of the local partitions to be accessed by the transaction. However, it is possible that such a set may not contain a vector corresponding to some partition to be accessed by the transaction. We present below a procedure to obtain a snapshot for all partitions to be accessed by the transaction.

We can form a consistent global snapshot by combining the partition dependency views of all the local partitions. If two local partitions contain in their $\mathcal{D}$ sets a vector for some partition p, then we can take 'super' of these two vectors as the snapshot for p. We follow this rule for each partition to be accessed across all local partition dependency views to form a global snapshot. Such a snapshot is consistent because the causal and atomic set dependencies of all the local partitions are collectively captured in this snapshot.

It is still possible that this set may not have a snapshot vector for some partition to be accessed by the transaction. For each such partition q, we then need to follow a procedure to obtain a snapshot from some remote site containing that partition. We read the partition view $\mathcal{V}_q$ of the remote site and consider it as the snapshot for q provided that its causal dependencies as indicated by the $\mathcal{D}_q$ set at the remote site have been seen by the local site. The function 'GetRemoteSnapshot' performs this step.

After obtaining the start snapshot time, transaction t performs local reads as shown in Algorithm 2. For a remote read, site i contacts the remote site j which then performs a

Algorithm 7 Obtaining snapshot for a multi-site transaction t at site i

```
function GETSNAPSHOT
    L ← local partitions accessed by t
    R ← non-local partitions accessed by t
    for each p ∈ L do
        S_t^p ← V_p
    for each q ∈ R do
        for each p ∈ partitions(i) do
            if D_p^q ∈ D_p then
                S_t^q ← super(D_p^q, S_t^q)
    for each q ∈ R such that S_t^q ∉ S_t do
        S_t^q ← GetRemoteSnapshot(S_t)
        if S_t^q is null then repeat above step using some other
        replica site for partition q

/* Function executed at remote site j to
obtain snapshot for t for partition q */
function GETREMOTESNAPSHOT(S_t, partition q)
    for each r such that D_q^r ∈ D_q ∧ S_t^r ∈ S_t do
        if S_t^r < D_q^r then
            return null
    return V_q
```

local read and returns the version. Before performing the read operation, site j checks if it is advanced up to the transaction's snapshot for that partition. This check is needed only in the case when the transaction did not need to contact the remote site when it constructed its start snapshot time.

If a transaction involves updating any remote partition, it must make sure to obtain a snapshot vector for that partition at the start of the transaction, as described above. In the commit phase it contacts that site to obtain a sequence number for that partition. The rest of the commit protocol is performed as shown in Algorithm 5. The updates to local partitions are applied first and remote updates are sent to the remote site using the update propagation mechanism described above. Even though there is a delay in applying updates to the remote partition, the atomicity guarantee in obtaining a snapshot is still guaranteed because the $\mathcal{D}$ vector set of the local partitions would force the use of the updated view of the remote partition.

E. Discussion of Protocol Correctness

We now discuss the correctness of the P-CSI protocol. The property central to the correctness of the protocol is that the partition dependency view $\mathcal{D}$ of any partition at a site reflects a consistent global snapshot across all partitions. The partition dependency view is updated whenever a transaction is executed modifying that partition. Initially, $\mathcal{D}$ sets for all partitions are empty, and therefore this property holds trivially.

We show that this property holds by using induction on the length of causal sequence of transactions executed at a site. Assuming that this property holds for transaction sequences up to length n, we show that this property is preserved when a new causally succeeding transaction t is executed at the site extending the length of a causal sequence to $n+1$.

As shown in Algorithm 5, in the commit phase the transaction t updates some partitions and updates their $\mathcal{D}$ vector sets and partition views. The first step involves obtaining commit timestamps for each partition to be modified. It then inserts new versions of the modified items in these partitions. However, these versions are not visible to any transaction since the partition views are yet not advanced. Next, the commit protocol computes the transaction dependency view $\mathcal{TD}_t$. The procedure shown in Algorithm 4, constructs a vector clock for each partition r on which transaction t is causally dependent to capture all the causally preceding events in partition r. This is denoted by $\mathcal{TD}_t^r$. These causal dependencies arise because of partitions accessed by t which happen to be dependent on events in r. Suppose that t accesses partitions $\{P1, P2, \cdots, Pk\}$, then $\mathcal{TD}_t^r$ has the following property, where $\mathcal{E}_t^r$ is the effective causal snapshot of partition r if it is accessed by t.

$$\mathcal{TD}_t^r = super(\mathcal{D}_{P1}^r, \mathcal{D}_{P2}^r, \cdots, \mathcal{D}_{Pk}^r, \mathcal{E}_t^r) \quad (1)$$

The above expression means that $\mathcal{TD}_t^r$ includes all causal dependencies of transaction t on partition r.

Next the transaction updates the $\mathcal{D}$ vector sets of all the modified partitions using the $\mathcal{TD}_t$ vector. For each such partition p and each r in $\mathcal{TD}_t$, it sets $\mathcal{D}_p^r$ equal to $\mathcal{TD}_t^r$, thus updating the dependency view of a modified partition p to include its dependency on events in r. Moreover, for each modified partition p, $\mathcal{D}_p^q$ for each other modified partition q, $q \neq p$, is modified using the $\mathcal{C}_t^q$ vector so that $\mathcal{D}_p$ includes in its view the update event of t on partition q. This ensures that the partition dependency view of each modified partition captures all events in the atomic set of transaction t. The steps of modifying $\mathcal{D}$ vectors are performed as a single atomic action to ensure that any other concurrent transaction would always observes a consistent snapshot. The modified $\mathcal{D}$ vector sets ensure the consistency property mentioned above.

Propagation of transaction t's updates to remote sites includes $\mathcal{TD}_t$, $\mathcal{C}_t$ and the write-set. Before applying updates of transaction t, each remote site ensures that for each partition r stored at the remote site, it has applied all the causally preceding events implied by $\mathcal{TD}_t^r$. When the updates of t are applied, the procedure followed for updating $\mathcal{D}$ vector set and partition views of the partitions located at the remote site is the same as that described above for the execution site. Thus the modified $\mathcal{D}$ vector sets at the remote sites also ensure the consistency property mentioned above.

VI. EVALUATIONS

We present below the results of our evaluation of the P-CSI model. For these evaluations, we implemented a prototype system implementing the P-CSI protocol. In our prototype, we implemented an in-memory key-value store to serve as the local database for a site. Each site also maintains a 'commit-log' in secondary storage. During the commitment of an

update transaction, the updates are written to the 'commit log'. Committed update transactions are propagated by the execution site at periodic intervals, set to 1 second. During the update synchronization phase at the remote site (refer Algorithm 6), if the updates cannot be applied, then the site buffers the updates locally and delays their application until the corresponding vector clock values have been advanced enough.

A. Experiment Setup

We performed the evaluations on a cluster of 30 nodes using the resources provided by Minnesota Supercomputing Institute. Each node in the cluster had 8 CPU cores with 2.8 GHz Intel X5560 "Nehalem EP" processors, and 22 GB main memory. Each node in the cluster served as a single database site in our experiments.

Replication configuration: We performed experiments for different numbers of sites. The number of partitions was set equal to the number of sites. Thus, for 10 sites, the system database consisted of 10 partitions, whereas for 20 sites the database was scaled accordingly to contain 20 partitions. Each partition was replicated on three sites. Each partition contained 100,000 items of 100 bytes each. For a partition, we designated one of its replica sites as the conflict resolver for items in that partition.

We synthesized a transaction workload consisting of two types of transactions: *local* transactions which accessed only local partitions, and *non-local* transactions which accessed some remote partitions from a randomly selected site. In the transaction workload, each transaction read from 2 partitions and modified 1 partition. In case of local transactions. the partitions to read/modify were randomly selected from the local partitions. For each accessed partition, the transaction read/modified 5 randomly selected items. In subsection VI-E, we varied the number of modified partitions to evaluate its impact, as described later.

In these evaluations, we were interested in evaluating the following aspects: (1) advantages of partial replication using P-CSI over full replication, (2) scalability of the P-CSI model, and (3) impact of locality in transaction execution. The primary performance measures used were: (1) transaction throughput measured as committed transactions per second, (2) transaction latency, measured in milliseconds, and (3) cost of update propagation, measured as number of propagation messages sent per transaction.

B. Advantages over Full Replication

We first present the comparative evaluation of partial replication using the P-CSI model and full replication. This evaluation was performed using 20 sites, with all transactions accessing local partitions. For full replication scheme, the database contained only one partition, consisting of 2 million items, replicated on all sites. Note that this configuration corresponds to the basic CSI model. Table I shows the result of this evaluation. The 'max throughput' column in the table gives the maximum transaction throughput that we could achieve for that system configuration. As noted earlier, one of the advantages of the P-CSI model is that updates need to be propagated only to the sites storing the modified partitions. In contrast, full replication requires propagating updates to all sites, resulting in lower throughput. With partial replication, the throughput achieved is more than factor of two compared to full replication. The number of update propagation messages per transaction depends on the number of modified partitions. In our experiments, the number of update propagation messages per transaction in case of partial replication was found to be close to 2, whereas this number in case of full replication was close to 19.

TABLE I
COMPARATIVE EVALUATION OF PARTIAL AND FULL REPLICATION

	Max. Throughput	Transaction Latency (msec)	Visibility Latency (msec)
Partial Rep.	14937	65.3	4608
Full Rep.	6494	87.8	8714

Another important measure is the *visibility latency* of an update, which is the time since an update is committed till the time it is applied at all the replicas storing the updated item. This depends on three factors: network latency, delays in propagating updates at the execution site because of the lazy propagation model, and delays in applying updates at the remote site due to causal dependencies. In our experiments the update propagation interval was set to 1 second. Visibility latency indicates the inconsistency window i.e. the time for which the data copies at different replicas are inconsistent with respect to a given update. We show in Table I the average value of visibility latency for partial and full replication models. In case of full replication, due to the higher cost of update propagation the visibility latency is higher, indicating that replicas are out-of sync for a longer time compared to the partial replication.

C. Scalability

We evaluated the P-CSI model for various system sizes, i.e. number of sites, to demonstrate its scalability under the scale-out model. For each system size, we measured the maximum transaction throughput and the average transaction latency. Figures 4 and 5 show the maximum throughput and average latency of transactions, respectively. The transaction throughput increases almost linearly with the increase in system size, with only marginal increase in latencies.

D. Impact of Non-local Transactions

To evaluate the impact of locality in transaction execution, we induced non-local transactions, i.e. transactions with remote partition access. We varied the percentage of the non-local transactions to observe the impact on transaction latencies. Figure 6 shows the results of this evaluation. We show in this figure the average latencies for all transactions as well as average latencies for non-local transactions. We observe only a slight increase in the overall latencies due to

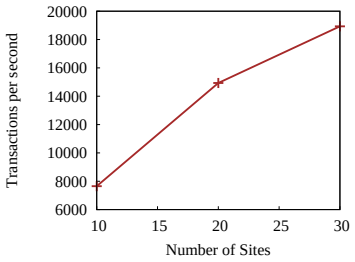

Fig. 4. Transaction Throughput Scalability

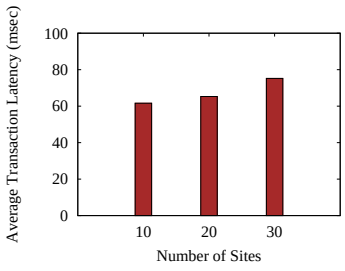

Fig. 5. Transaction Latency

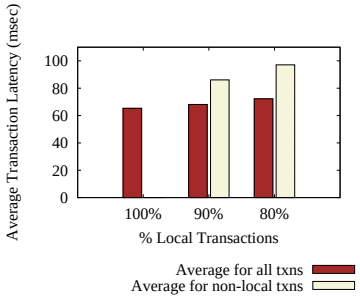

Fig. 6. Impact of Non-Local Transactions

TABLE II
IMPACT OF TRANSACTION SIZE

No. of Read Partitions	No. of Write Partitions	Transaction Latency	Avg. Prop. Messages
2	1	65.3	1.98
3	2	82.8	3.23
4	3	89.7	4.31

non-local transactions, however, these latencies can be higher in wide-area environments.

E. Impact of Transaction Size

As noted earlier, in partial replication, the cost of update propagation, i.e. the number of update propagation messages sent for a transaction, depends on the number of modified partitions m. Thus, if n is the largest degree of replication for a partition, then the number of update propagation messages for a transaction modifying m partitions is at most $m \cdot (n-1)$. In Table II we show the number of update propagation messages per transaction observed in our experiments for different values of m. We also show in this table the average transaction latencies. The average transaction latency increases with m mainly due to the increase in the latencies imposed by the 2PC protocol, since a transaction would need to coordinate with more number of sites with increase in m. This evaluation was conducted using 20 sites and 40 partitions.

VII. CONCLUSION

We have presented in this paper a model called Partitioned-CSI (P-CSI) for transaction management in partially replicated databases with asynchronous update propagation. In this model, transactions can be executed at any site and can access any subset of partitions. The P-CSI model provides a weaker form of snapshot isolation, which is based on causal consistency. The P-CSI model guarantees that a transaction always observes a causally consistent snapshot. We have elaborated in this paper the unique issues that are raised due to partial replication in supporting causal consistency with the snapshot isolation model. We address these issues in this paper and present a transaction management protocol which ensures causal consistency and requires sending update propagation messages only to the sites storing the modified partitions. There are several aspects of this model which make it attractive for large scale environments while providing a useful consistency model. These aspects include snapshot isolation with causal consistency that eliminates need for a global sequencer, asynchronous update propagation, and support for partial replication requiring messages to be communicated only with the sites containing the partitions accessed by a transaction. We have implemented a data replication management system using the P-CSI model. We evaluated this system on a cluster to demonstrate the scalability and the performance benefits of partial replication using P-CSI over the full replication scheme.

REFERENCES

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, October 2007.

[2] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, pp. 1277–1288, August 2008.

[3] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proc. of ACM SOSP*, 2011, pp. 385–400.

[4] V. Padhye and A. Tripathi, "Causally Coordinated Snapshot Isolation for Geographically Replicated Data," in *Proc. of IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2012, pp. 261–266.

[5] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS," in *Proc. of the 23rd ACM SOSP*, 2011, pp. 401–416.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," in *Proc. of ACM SIGMOD'95*. ACM, 1995, pp. 1–10.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[8] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distributed Parallel Databases*, vol. 14, no. 1, Jul. 2003.

[9] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299 – 319, December 1990.

[10] S. Elnikety, S. Dropsho, and F. Pedone, "Tashkent: uniting durability with transaction ordering for high-performance scalable database replication," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. ACM, 2006, pp. 117–130.

[11] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-r, a new way to implement database replication," in *Proceedings of the 26th International Conference on Very Large Data Bases*, ser. VLDB '00, 2000, pp. 134–143.

[12] ——, "A new approach to developing and implementing eager database replication protocols," *ACM Trans. Database Syst.*, vol. 25, pp. 333–379, September 2000.

[13] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *IEEE Trans. on Knowl. and Data Eng.*, vol. 15, no. 4, pp. 1018–1032, July 2003.

[14] Y. Lin, B. Kemme, M. Patiño Martínez, and R. Jiménez-Peris, "Middleware based data replication providing snapshot isolation," in *Proc. of the ACM SIGMOD*, 2005, pp. 419–430.

[15] M. Patiño Martinez, R. Jiménez-Peris, B. Kemme, and G. Alonso, "Middle-r: Consistent database replication at the middleware level," *ACM Trans. Comput. Syst.*, vol. 23, no. 4, Nov. 2005.

[16] C. Plattner and G. Alonso, "Ganymed: scalable replication for transactional web applications," in *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, 2004, pp. 155–174.

[17] I. Stanoi, D. Agrawal, and A. El Abbadi, "Using broadcast primitives in replicated databases," in *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'98)*, 1998, pp. 148–155.

[18] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. ACM, 1996, pp. 173–182.

[19] S. Wu and B. Kemme, "Postgres-R(SI): Combining Replica Control with Concurrency Control Based on Snapshot Isolation," in *Proc. of IEEE ICDE*, 2005, pp. 422–433.

[20] S. Elnikety, F. Pedone, and W. Zwaenepoel, "Database replication using generalized snapshot isolation," in *24th IEEE Symposium on Reliable Distributed Systems*, 2005, pp. 73 – 84.

[21] M. Wiesmann, F. Pedone, A. Schiper, K. B., and G. Alonso, "Understanding replication in databases and distributed systems," in *Proc. of the 20th International Conference on Distributed Computing Systems*, 2000, pp. 464–474.

[22] K. Daudjee and K. Salem, "Lazy database replication with snapshot isolation," in *Proc. of the VLDB*, 2006, pp. 715–726.

[23] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris, "Enhancing edge computing with database replication," in *26th IEEE International Symposium on Reliable Distributed Systems*, 2007, pp. 45 –54.

[24] H. Jung, H. Han, A. Fekete, and U. Roehm, "Serializable Snapshot Isolation for Replicated Databases in High-Update Scenarios," in *VLDB*, 2011.

[25] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *CIDR*, 2011, pp. 223–234.

[26] J. Holliday, D. Agrawal, and A. El Abbadi, "Partial database replication using epidemic communication," in *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, ser. ICDCS '02, 2002.

[27] A. Sousa, F. Pedone, R. Oliveira, and F. Moura, "Partial replication in the database state machine," in *In Proceedings of NCA'01*, 2001.

[28] U. F. Jr. and P. Ingels, "Transactions on partially replicated data based on reliable and atomic multicasta," in *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'01)*, 2001, pp. 284–291.

[29] P. Sutra and M. Shapiro, "Fault-tolerant partial replication in large-scale database systems," in *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, ser. Euro-Par '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 404–413.

[30] N. Schiper, P. Sutra, and F. Pedone, "P-store: Genuine partial replication in wide area networks," in *IEEE Symposium on Reliable Distributed Systems*, 2010, pp. 214–224.

[31] D. Serrano, M. Patio-martinez, R. Jimenez-peris, and B. Kemme, "Boosting database replication scalability through partial replication and 1-copy-snapshotisolation," in *In Proceedings of PRDC.07*, 2007.

[32] G. T. Wuu and A. J. Bernstein, "Efficient solutions to the replicated log and dictionary problems," in *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, 1984, pp. 233–242.

[33] J. E. Armendáriz-Iñigo, A. Mauch-Goya, J. R. G. de Mendívil, and F. D. Muñoz Escoí, "SIPRe: a partial database replication protocol with SI replicas," in *Proceedings of the 2008 ACM Symposium on Applied computing*, 2008, pp. 2181–2185.

[34] N. Schiper, R. Schmidt, and F. Pedone, "Optimistic Algorithms for Partial Database Replication," in *Proceedings of OPODIS*, 2006, pp. 81–93.

[35] D. Serrano, M. Patiño Martínez, R. Jiménez-Peris, and B. Kemme, "An autonomic approach for replication of internet-based services," in *Proceedings of the 2008 IEEE Symposium on Reliable Distributed Systems*, 2008, pp. 127–136.

[36] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "The potential dangers of causal consistency and an explicit solution," in *Proceedings of the Third ACM Symposium on Cloud Computing*. New York, NY, USA: ACM, 2012, pp. 22:1–22:7.

[37] C. Fidge, "Logical time in distributed computing systems," *Computer*, vol. 24, pp. 28–33, August 1991.

[38] F. Mattern, "Efficient algorithms for distributed snapshots and global virtual time approximation," *J. Parallel Distrib. Comput.*, vol. 18, pp. 423–434, August 1993.