Transaction Management using Causal Snapshot Isolation in Partially Replicated Databases

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

***Abstract*—We present here a transaction management protocol using causal 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, and its updates are propagated asynchronously to other sites. The protocol ensures that the snapshot observed by a transaction contains data versions that are causally consistent. The protocol requires propagating updates only to the sites replicating the updated items. 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. Through experimental evaluations, we demonstrate the scalability of this model and its performance benefits over full replication models.**

I. Introduction

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 [12], [9]. In asynchronous replication, the transaction is first committed locally and then its updates are asynchronously propagated later. This model provides lower latencies in transaction execution and high availability, but guarantees only eventual consistency [12] or causal consistency [33], [23].

We address here 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. 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. With partial replication, a partition may be replicated only in those geographic regions where it is likely to be accessed most frequently. Our goal is to provide Snapshot Isolation (SI) [8] based transaction support with 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. Due to these advantages, several systems [33], [23] have been developed recently with focus on supporting causal consistency. However, these systems are designed primarily for full-replication model and do not support partial replication. Supporting causal consistency under partial replication raises unique issues. In this paper, we address these issues and provide an efficient transaction model, called *Partitioned Causal Snapshot Isolation (PCSI)*.

A transaction may be executed at any site and may access items in any of the partitions. A transaction is committed locally at a site, and then, at some later time, its updates are propagated asynchronously to other sites and applied there. Transactions are ordered according to a causal ordering. The snapshot observed by a transaction may not always reflect the latest versions of the accessed items, but it is guaranteed to be consistent as described below.

The PCSI model provides the following guarantees for transaction execution under partial replication.

- Snapshot Isolation: As in the case of traditional snapshot isolation, the PCSI model guarantees that when two or more concurrent transactions update a common data item, only one of them is allowed to commit.
- Transaction Ordering: The PCSI model provides ordering guarantees for transactions based on three properties: *causal ordering*, *per-item global update ordering*, and *per-partition site-based ordering*. We define the ordering relationship ($\prec$) which provides a partial ordering over a set of transactions. Two non-concurrent transactions T_i and T_j are ordered as follows.
  - *causal ordering*: If T_j reads any of the updates made by T_i, then transaction T_i causally precedes transaction T_j ($T_i \prec T_j$).
  - *per-item global update ordering*: $T_i \prec T_j$ if T_j creates a newer version for any of the items modified by T_i, i.e. T_i commits before T_j.
  - *per-partition site-based update ordering*: $T_i \prec T_j$ if both T_i and T_j execute at the same site, both update a common partition, and T_i obtains its commit timestamp before T_j.

The ordering relationship is transitive, i.e. if $T_i \prec T_j$

and $T_j \prec T_k$, then $T_i \prec T_k$. Given a set of transactions, a site applies only those transactions which update any of the site's local partitions. In the PCSI model, these transactions are applied to the partition's replica at a site according to the ordering relationship ($\prec$).

- Globally Consistent Snapshot: In partial replication model, a transaction may access multiple partitions that may be stored at different sites. The PCSI model guarantees that a transaction observes a *consistent snapshot* spanning multiple partitions. A consistent snapshot has the following properties of *atomicity* and *causality*:
  - *Atomicity*: In a consistent snapshot either all or none of the updates of a transaction are visible.
  - *Causality*: If a snapshot contains updates of transaction T_i, then updates of all transactions causally preceding T_i are also contained in it.

We present the PCSI protocol for transaction management in partially replicated databases. We implemented a prototype system for evaluating the PCSI model. We evaluated this system using a custom benchmark on a local cluster and the Amazon EC2 environment. The main contributions of this paper are as follows. First, we identify the unique issues in supporting causal consistency in partitioned databases with partial replication. Second, we present a transaction model based on snapshot isolation in partially replicated databases. This model ensures causal consistency of data. Third, we develop a transaction execution protocol that uses asynchronous update propagation model and requires communicating updates only to the sites replicating the updated items. Finally, through experimental evaluations, we demonstrate the scalability benefits of partial replication using the PCSI model over full replication-based models.

The rest of the paper is organized as follows. In the next section we discuss the related work. Section III highlights the issues in supporting snapshot based transactions with causal consistency in partially replicated databases. Section IV provides conceptual overview of the PCSI model. In Section V we present the details of the PCSI protocol. Evaluations of the proposed model and its 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*. The issues with scalability in data replication with strong consistency requirements are discussed in [16]. 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) [8] and causal consistency.

Replication using snapshot isolation (SI) has been studied widely [22], [14], [35], [21]. SI-based database replication using lazy replication in the primary-backup model is investigated in [11]. Compared to the primary-backup model, the symmetric execution model is more flexible but requires coordination among replicas. Many of the systems for SI-based database replication [22], [14], [18] use eager replication with atomic broadcast to ensure that the replicas observe a total ordering of transactions. The notion of *1-copy snapshot isolation* [22] means that the schedule of transaction executions at different replicas under the read-one-write-all (ROWA) model is equivalent to an execution schedule 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 [12], [9], [7], [23], [33]. Dynamo [12] uses asynchronous replication with eventual consistency but does not support transactions. PNUTS [9] also does not provide transactions, but provides a stronger consistency level than eventually consistency, called as *eventual timeline consistency*. Megastore [7] provides transactions over a group of entities using synchronous replication. COPS [23] provides causal consistency, but does not provide transaction functionality, except for snapshot-based read-only transactions. Eiger [24] provides both read-only and update transactions with causal consistency but requires maintaining causal dependencies on per object level. PSI [33] provides transaction functionality for geo-replicated data using asynchronous replication, guaranteeing causal consistency.

Another approach for achieving higher scalability is to use partial replication instead of replicating the entire database on all sites [17], [32], [15], [34], [29], [30]. The approach presented in [17] guarantees serializability. It uses epidemic communication that ensures causal ordering of messages using a vector clock scheme where each site knows how current is a remote site's view of the events at all other sites. Other approaches [32], [29], [4], [28] are based on the database state machine model [27], utilizing atomic multicast protocols. These approaches support *1-copy serializability*. In contrast, the approach presented in [30] is based on the snapshot isolation model, providing the guarantee of *1-copy snapshot isolation*. This model is applied to WAN environments in [31] but relies on a single site for conflict detection in the validation phase. The notion of *genuine partial replication* [28] requires that the messages related to a transaction should only be exchanged between sites storing the items accessed by the transaction. The system presented in [4] uses the notion of generalized snapshot isolation (GSI) [14], where a transaction can observe a consistent but old snapshot of the database.

The Non-Monotonic Snapshot Isolation (NMSI) protocol presented in [3] is the closest to our work. Our work differs from NMSI in following ways. NMSI protocol requires maintaining dependency information either on per object version level or on per partition level with the restriction that updates within a partition should be serialized. For detecting concurrent update conflicts on a partition, it uses an atomic multicast protocol involving all replicas of all objects in a transaction's write-set. In contrast, PCSI uses vector clocks effectively to avoid maintaining any dependency information on per object level, and it does not require that updates on a partition be serialized. Only the updates on a given object are serialized which is a result of the no write-write conflict requirement,

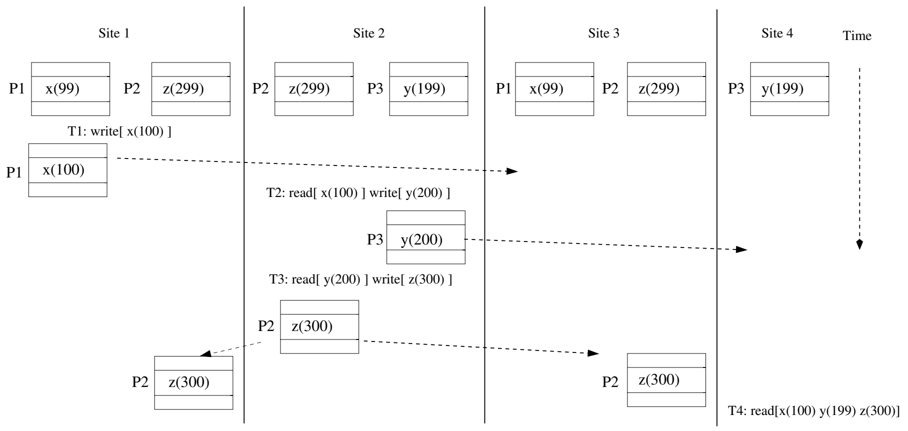

Fig. 1. Issues in supporting causality under partial replication

and the updates from a site to a partition are serialized using a local sequencer, without requiring any remote coordination. PCSI also minimizes causal dependencies by considering only the read-write sets of a transaction [26]. For obtaining the start snapshot, NMSI supports the notion of *forward freshness*, whereas in PCSI it is obtained in the beginning and it is "frozen". We believe that this advantage of NMSI is minor considering that typically abort rates due to update conflicts tend to be small, whereas its model of serializing all updates on a partition is a major drawback.

Orbe [13] supports partitioned and replicated databases supporting causal consistency, but it does not support multi-key update transactions. PCSI uses two-dimensional vector clocks similar to Orbe's dependency matrix and time-tables in [36]. Bolt-on causal consistency [6] provides mechanisms for causal consistency in systems with eventual consistency.

Spanner [10] provides strong consistency with serializable transactions under global-scale replication. However, it relies on special purpose hardware such as GPS or atomic clocks to minimize clock uncertainty. The work presented in [20] provides a new type of consistency scheme called as *red-blue consistency* which uses operation commutativity to relax certain ordering guarantees for better performance. The work in [19] presents a transaction commit protocol for wide-area replication that is more efficient than 2PC or Paxos.

III. Issues in supporting causal consistency under partial replication

Several issues arise in supporting causal consistency in partially replicated database systems where transaction updates are propagated asynchronously and only to the sites containing replicas of modified partitions. Ensuring causality guarantees requires that for applying a given transaction's updates to a partition's replica at a site, all its causally preceding events, including the transitive dependencies, must be captured in the state of the local partitions of that site. We illustrate this problem using the example 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

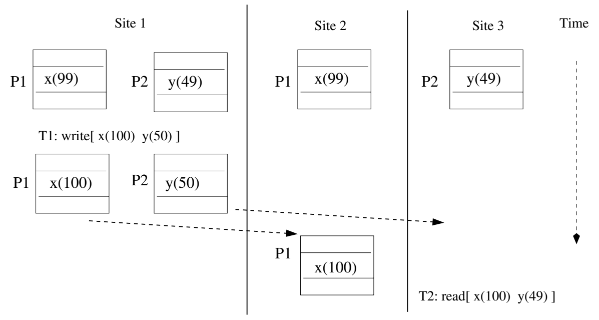

Fig. 2. Issues in obtaining atomically consistent snapshot

3. Partition $P3$ containing y is replicated at sites 2 and 4. The latest version number of an item is shown in parentheses.

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 $x(100)$. $T3$'s update is propagated asynchronously to sites 1 and 3. Suppose $T3$'s update for $z(300)$ arrives at site 3 before the update of transaction $T1$ for $x(100)$. In case of full replication based model such as PSI [33], 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 an 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 [23], or (2) communicating every update to all the sites in the system so that each site is cognizant of all causal dependencies. 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 [5].

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 $P2$. The snapshot observed by $T4$ is causally inconsistent because it contains $z(300)$ but not the causally preceding version $y(200)$.

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 2. 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 2. 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 PCSI model to address such issues in ensuring causal consistency.

IV. Overview of the PCSI Model

We consider partial replication of a database that consists of a 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 site is identified by a unique $siteId$. Each partition is replicated across one or more sites. Each site has a local database which supports multi-version data management.

Before executing read/write operations, a transaction must obtain a globally consistent snapshot satisfying the atomicity and causality properties. All read operations on a partition are performed according to the snapshot obtained for that partition. If the partition to be read is stored at the local site, it executes read operation on the local database, otherwise the transaction performs a read from a remote site. The writes are buffered till the commit time. When a transaction is ready to commit, it checks for update conflicts with concurrently committed transactions. In case of no conflicts, the transaction is committed and applied at the local site and its updates are asynchronously propagated to other sites that store the partitions updated by the transaction. For ensuring causal consistency, the causal dependencies of the transaction are computed and this information is communicated with the update propagation message. A remote site applies the updates only if it has applied updates of all the causally preceding transactions.

A. Timestamps, Vector Clocks and Partition Dependency View

Our goal in designing the PCSI model is to avoid the need of propagating updates to all sites. This requires distinguishing between transactions based on the partitions modified by them. Our solution to this problem is based on assigning transaction sequence numbers and maintaining causal dependency information on per-partition-replica basis. We refer to the partitions stored at a site as the *local partitions* of that site. In the PCSI model, a site 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 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. A transaction obtains, during its commit phase, a timestamp for each partition it is modifying. A timestamp is a pair $<siteId,\ seq>$, where seq is a local sequence number assigned to the transaction, for that partition, 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. For example, the commit timestamp vector $\mathcal{C}_t$ of transaction t modifying partitions p, q, and r is a set of timestamps $\{\mathcal{C}_t^p,\ \mathcal{C}_t^q,\ \mathcal{C}_t^r\}$. For an item modified by transaction t in partition q, the version number of the item is commit timestamp $\mathcal{C}_t^q$.

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 copy of p 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 value as well as all the causally preceding transactions that updated any of the partitions at the site.

For capturing causality and atomicity dependencies, a site also maintains, for each local partition p, a *partition dependency view ($\mathcal{D}_p$)*, which is a set of vector clocks. 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 current 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 the vector clock $\mathcal{D}_p^p$ is same as $\mathcal{V}_p$.

B. 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 a set of vector clocks $\{\mathcal{S}_t^1, \mathcal{S}_t^2, \cdots, \mathcal{S}_t^q, \cdots, \mathcal{S}_t^n\}$. An element $\mathcal{S}_t^p$ corresponds to the snapshot vector clock obtained for partition p. This snapshot should satisfy the global consistency requirement as defined below. For this, we define the *visible* relationship as follows. A transaction is visible in a snapshot of a partition at a site if the transaction's updates have been applied at that site. We formally define below the *visible* relationship for a transaction τ and snapshot S^q for a partition q as follows.

$$visible(\tau, \mathcal{S}^q) : \mathcal{C}_\tau^q.seq \leq \mathcal{S}^q[\mathcal{C}_\tau^q.siteId] \quad (1)$$

Globally consistent snapshot: A snapshot $\mathcal{S}$ is globally consistent, if for all pairs of partitions p, q in the snapshot $\mathcal{S}$, the following two requirements hold.

1) Causality: If a transaction t_i is *visible* in the snapshot $\mathcal{S}^p$ for partition p, i.e. $visible(t_i, \mathcal{S}^p)$, and there exists transaction t_j, $t_j \prec t_i$, such that t_j has modified partition q, then t_j should be *visible* in snapshot for q, i.e. $visible(t_j, \mathcal{S}^q)$.
2) Atomicity: If transaction t_i has modified partitions p and q, then $visible(t_i, \mathcal{S}^p)$ if and only if $visible(t_i, \mathcal{S}^q)$

When obtaining a snapshot, we use the partition dependency views to check if a snapshot is globally consistent for a given set of partition replicas to be accessed. For each pair of partitions p, q to be accessed by the transaction, we can consider $\mathcal{V}_p$ and $\mathcal{V}_q$ as snapshots for p and q provided the following conditions holds: $\mathcal{D}_p^q \leq \mathcal{V}^q \wedge \mathcal{D}_q^p \leq \mathcal{V}^p$.

C. Transaction Validation

Like PSI [33], for each data item, there is a designated *conflict resolver site* which is responsible for checking for update conflicts for that item. A conflict resolver site maintains an ordered list of the commit timestamps ($<siteId, seq>$) of all the committed versions of the corresponding item. The transaction coordinates with conflict resolver sites responsible for the items in its write-set to check if some other transaction has created an item version newer than the latest version visible in the transaction's snapshot. Such a situation indicates an update conflict. This coordination is done using a two-phase-commit (2PC) protocol. The transaction commits only if none of the items in its write-set have an update conflict.

D. Update Propagation

For ensuring causal consistency, a transaction's updates are applied at remote sites only after the updates of all the causally preceding transactions have been applied. For this purpose, the causal dependencies of the transaction are captured in form of a set of vector clocks, called *transaction dependency view* ($\mathcal{TD}$). A vector clock in this set corresponds to a partition and identifies all the causally preceding transactions pertaining to that partition. The atomicity dependencies are captured by the commit timestamp $\mathcal{C}_t$ value. The $\mathcal{TD}$ and $\mathcal{C}_t$ values are communicated with the update propagation message. The transaction's updates are delayed at a remote site until all its local partitions are advanced enough according to corresponding vectors in $\mathcal{TD}$. Furthermore, when applying the transaction, the partition dependency views of the modified partitions are updated using $\mathcal{TD}$ and $\mathcal{C}_t$ values to indicate the partition's causal and atomic dependencies on other partitions.

V. PCSI Protocol Description

We present here a formal description of the PCSI protocol. For simplicity of the presentation, we first describe the execution of a *local transaction*, i.e. a transaction executed at a site which stores all the partitions accessed by the transaction. Later, we describe how a transaction that requires accessing partitions at remote sites is executed.

A. Execution of Local Transactions

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 views of those partitions. The partition views of the local partitions at a site always form a consistent snapshot. 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. When the snapshot is obtained, the $\mathcal{D}$ vector values of the accessed partitions are also recorded, which are later used in Algorithm 4. We call these as snapshot-$\mathcal{D}$ vectors. The steps enclosed in 'begin' and 'end atomic region' are performed as a critical section at the execution site.

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

Transaction Validation: When the transaction reaches its commit point, it performs update conflict checking if it has modified any items. Algorithm 3 shows the protocol for conflict

Algorithm 1 Obtaining start snapshot time
```
function GETSNAPSHOT
    $\mathcal{P} \leftarrow$ partitions accessed by the transaction
    [ begin atomic region
    for all $p \in \mathcal{P}$ do
        $\mathcal{S}_t^p \leftarrow \mathcal{V}_p$
        snapshot-$\mathcal{D}_p \leftarrow \mathcal{D}_p$
    end atomic region ]
```

Algorithm 2 Performing read operations
```
function READ($item\ x$)
    $p \leftarrow$ partition containing item $x$
    /* performed in reverse temporal order of versions */
    for all $version\ v \in version\ log$ of $x$ do
        if $\mathcal{S}_t^p[v.siteId] \geq v.seq$ then return $v.data$
```

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 all $s \in sites$ do
        $itemList \leftarrow \{x | x \in writeset \wedge resolver(x) = s\}$;
        invoke RecvPrepare($itemList$, $\mathcal{S}$) at $s$
    if all votes are 'yes' then
        perform Commit function as shown in Algorithm 5;
        for all $s \in sites$ do
            invoke RecvCommit($itemList$, $\mathcal{C}_t$) at $s$
    else
        invoke RecvAbort($itemList$) at each $s \in sites$;
        abort transaction
```

```
/* Functions executed by the conflict resolver site */
function RECVPREPARE($itemList$, $\mathcal{S}_t$)
    for all $x \in itemList$ do
        $p \leftarrow$ partition containing item $x$
        $v \leftarrow$ latest version of item $x$
        if $\mathcal{S}_t^p[v.siteId] \geq v.seq \wedge x$ is unlocked then
            lock $x$
        else
            return response with 'no' vote
    if all $x \in itemList$ are locked then send response with
    'yes' vote

function RECVCOMMIT($itemList$, $\mathcal{C}_t$)
    for all $x \in itemList$ do
        $p \leftarrow$ partition containing item $x$
        record version timestamp $\mathcal{C}_t^p$ in version log
        release lock on $x$

function RECVABORT($itemList$)
    release locks on all $x \in itemList$
```

checking. The update conflict checking is performed using a two-phase commit (2PC) protocol with the conflict resolver sites responsible for the items contained in the transaction's write-set. If the local site is the conflict resolver for all write-set items then the conflict checking is performed locally without any coordination with any remote sites. In the first phase of 2PC, the transaction sends a prepare message to the conflict resolver sites, containing its write-set items and its start snapshot timestamp. A conflict resolver site checks, for each item it is responsible for, whether the latest version of that item is visible in transaction's start snapshot and that item is not locked by any other transaction. If this check fails, it sends a 'no' vote, else it sends 'yes' vote. If the transaction receives 'yes' votes from all conflict resolvers, it proceeds to executing commit phase.

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:

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.
    for all $p \in \mathcal{P}$ do
        $\mathcal{TD}_t^p \leftarrow \mathcal{S}_t^p$
    [ begin atomic region
    for all $p \in \mathcal{P}$ do
        for all $\mathcal{D}_p^q \in$ snapshot-$\mathcal{D}_p$ 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 region ]

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

$$\mathcal{TD}_t = \{\mathcal{TD}_t^1, \mathcal{TD}_t^2, \cdots, \mathcal{TD}_t^q, \cdots, \mathcal{TD}_t^n\}$$

An element $\mathcal{TD}_t^q$ identifies the transactions performed on partition q which causally precede t. Algorithm 4 shows the pseudocode for computing $\mathcal{TD}_t$. We first compute $\mathcal{TD}_t$ using the start snapshot timestamp of the transaction. Since the transaction observes only the events visible in its snapshot, the snapshot vector captures all the causally preceding events pertaining to the partitions accessed by the transaction. The start snapshot may contain events, or in other words item versions, that are not read by the transaction. In order to eliminate the false dependencies on such events, we can consider only the item versions read by t in computing $\mathcal{TD}_t$, as described in our earlier work on CSI [26]. However, for simplicity of discussion we omit this part in our protocol description. We then capture the causal dependencies on other partitions by including in $\mathcal{TD}_t$ the snapshot-$\mathcal{D}_p$ vectors of each partition p accessed by t. If any two partitions $p1$ and $p2$ accessed by t, each have in their dependency view 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.

Commit phase: Algorithm 5 shows the commit protocol for a 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 items updated by the transaction are written as new versions in the local database, and transaction dependency view set $\mathcal{TD}_t$ is computed. The updates to different partitions are applied concurrently without needing any synchronization, except when getting sequence number, and advancing vector clocks and dependency views. 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

Algorithm 5 Commit Protocol for transaction at site j

```
function COMMIT($writeset$)
    $\mathcal{P} \leftarrow$ partitions pertaining to $writeset$.
    [ begin atomic region
    for all $p \in \mathcal{P}$ do
        $ctr_p \leftarrow$ local sequence counter for partition $p$
        $\mathcal{C}_t^p.seq \leftarrow ctr_p{+}{+}$
    end atomic region ]
    ApplyUpdates($writeset$, $\mathcal{C}_t$)
    // compute dependencies as shown in Algorithm 4
    $\mathcal{TD}_t \leftarrow$ ComputeTransactionDependency()
    // advance local vector clocks
    AdvanceVectorClocks($\mathcal{TD}_t$, $\mathcal{C}_t$)
    /* propagate updates */
    propagate to every site that stores any partition $p \in \mathcal{P}$
    ($\mathcal{TD}_t$, $writeset$, $\mathcal{C}_t$)

function APPLYUPDATES($writeset$, $\mathcal{C}_t$)
    $\mathcal{P} \leftarrow$ partitions pertaining to $writeset$.
    for all $p \in \mathcal{P}$ do
        for all $item\ x$ in $writeset$ pertaining to $p$ do
            write the new version of $x$ to the local database.
            record version timestamp $\mathcal{C}_t^p$ in version log

/* Function to update vector clocks for partitions */
function ADVANCEVECTORCLOCKS($\mathcal{TD}_t$, $\mathcal{C}_t$)
    $\mathcal{P} \leftarrow$ partitions pertaining to $writeset$
    [ begin atomic region
    for all $p \in \mathcal{P}$ do
        for all $\mathcal{TD}_t^q \in \mathcal{TD}_t$ s.t. $q \neq p$ do
            $\mathcal{D}_p^q \leftarrow$ super($\mathcal{TD}_t^q$, $\mathcal{D}_p^q$)
        /* Advance $\mathcal{D}$ using $\mathcal{C}_t$ to capture the $t$'s
        update events in other partitions */
        for all $\mathcal{C}_t^q \in \mathcal{C}_t$ s.t. $q \neq p$ do
            $\mathcal{D}_p^q[\mathcal{C}_t^q.siteId] \leftarrow \mathcal{C}_t^q.seq$
        $\mathcal{V}^p[\mathcal{C}_t^p.siteId] \leftarrow \mathcal{C}_t^p.seq$
    end atomic region ]
```

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 asynchronously propagated to every site that stores any of the partitions modified by 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 $\mathcal{V}_p$ 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 $\mathcal{V}_p$

Algorithm 6 Applying updates at a remote site k

```
function RECVUPDATEPROPAGATION($\mathcal{TD}_t$, $writeset$, $\mathcal{C}_t$)
    // check if the site is up to date with respect to $\mathcal{TD}_t$
    for all $\mathcal{TD}_t^p \in \mathcal{TD}_t$ do
        if ($p$ is local partition) $\wedge$ $\mathcal{V}^p < \mathcal{TD}_t^p$ then
            buffer the updates locally
            synchronize phase: delay applying updates
            till the vector clock advances enough.
    for all $\mathcal{C}_t^p \in \mathcal{C}_t$ do
        if ($p$ is local partition) $\wedge$ $\mathcal{V}^p[\mathcal{C}_t^p.siteId] < \mathcal{C}_t^p.seq{-}1$
        then synchronize phase as shown above
    // apply updates to local partitions at site $k$
    ApplyUpdates($writeset$, $\mathcal{C}_t$)
    // advance vector clocks of site $k$
    AdvanceVectorClocks($\mathcal{TD}_t$, $\mathcal{C}_t$)
```

of the replica contains all the events preceding the sequence number value present in $\mathcal{C}_t^p$. If this check fails the site delays the 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.

B. 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). An important requirement for a multi-site transaction is to ensure that it observes a consistent global snapshot. We describe how the start snapshot vector is determined. 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. We can thus 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 event 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 to ensure that the condition $\mathcal{D}_p^q \leq \mathcal{V}^q \wedge \mathcal{D}_q^p \leq \mathcal{V}^p$ holds for all pairs of partitions p and q.

After obtaining the start snapshot time, transaction t performs local reads as shown in Algorithm 2. For a remote

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

```
function GetSnapshot
    $\mathcal{L} \leftarrow$ local partitions accessed by $t$
    $\mathcal{R} \leftarrow$ non-local partitions accessed by $t$
    for all $p \in \mathcal{L}$ do
        $\mathcal{S}_t^p \leftarrow \mathcal{V}_p$
    for all $q \in \mathcal{R}$ do
        for all  local partition $p$ do
            if $\mathcal{D}_p^q \in \mathcal{D}_p$ then
                $\mathcal{S}_t^q \leftarrow$ super($\mathcal{D}_p^q$, $\mathcal{S}_t^q$)
    for all $q \in \mathcal{R}$ such that $\mathcal{S}_t^q \notin \mathcal{S}_t$ do
        $\mathcal{S}_t^q \leftarrow$ GetRemoteSnapshot($\mathcal{S}_t$, $q$)
        if $\mathcal{S}_t^q$ 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($\mathcal{S}_t$, partition $q$)
    for all $r$ such that $\mathcal{D}_q^r \in \mathcal{D}_q \wedge \mathcal{S}_t^r \in \mathcal{S}_t$  do
        if $r \neq q \wedge \mathcal{S}_t^r < \mathcal{D}_q^r$ then
            return null; // indicates failure to get snapshot
        if $r = q \wedge \mathcal{S}_t^r > \mathcal{V}_q$ then
            return null; // indicates failure to get snapshot
    return ($\mathcal{V}_q$, $\mathcal{D}_q$);
```

read, site i contacts the remote site j which then performs a 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 where the transaction did not contact the remote site for obtaining its start snapshot.

If a transaction involves updating any remote partition, the execution site creates a local *ghost replica* for that partition. A ghost replica does not store any data but provides following functions. The main function is to assign local commit timestamps, using a local sequencer, to transactions updating this partition. It also stores the snapshot timestamp vector and the corresponding dependency vectors obtained from the remote site. The PCSI protocol does not propagate updates to the ghost replicas of any 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 ensured because the $\mathcal{D}$ vector set of the local partitions would force the use of the updated view of the remote partition.

Due to space limitation, the correctness proof of the protocol is omitted here. It is presented in [25] and establishes how the protocol guarantees the properties of snapshot isolation, transaction ordering, and consistent snapshots.

VI. Evaluations

We present below the results of our evaluation of the PCSI model. For these evaluations, we implemented a prototype system implementing the PCSI protocol. In our prototype implementation, we experimented with both in-memory database as well as HBase [2] as the persistent storage backend. In these evaluations, we were interested in evaluating the following aspects: (a) scalability of the PCSI model, (b) advantages of partial replication using PCSI over full replication, (c) impact of remote partition access in transaction execution, (d) the update application delays due to causal dependencies, and (e) average delay in a transaction's updates becoming visible at remote sites, which we refer to as *visibility latency*.

A. Experiment Setup

System Environments: We performed the evaluations using two types of system environments. The first environment is a local cluster of nodes using the resources provided by Minnesota Supercomputing Institute (MSI). In this cluster, each node 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 database site in our experiments. The other system environment we used is the Amazon EC2 cloud service [1]. We used 8 geographically distributed datacenters. At each site we used single 'Extra-Large' VM instance type, which had 8 cores with 1.2 GHz CPU capacity and 15 GB main memory. The average RTT between any two sites was found to be 214 ms.

Benchmarks and Database Configuration: We developed a custom benchmark to evaluate the impact of various parameters such as the degree of partial replication, number of partitions accessed by a transaction, percentage of read/write ratio, and the percentage of transactions accessing remote partitions, etc. The workload consists 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 our benchmark workload, the percentage of non-local transactions is configurable. In our evaluations, all transactions involve updating some items. Each transaction read from 2 partitions and modified 1 partition. For each accessed partition, the transaction read 4 items and modified 2 items, which were selected randomly with uniform distribution.

We performed experiments for different numbers of sites. The number of partitions was set equal to the number of sites. We emulated an environment where a geo-scale system is distributed over multiple regions, and each region has several sites. About half of the replicas of a partition are present in one region, and the other replicas are located at sites in different regions. In our experiments, all sites had the same number of partitions, and all regions had the same number of sites, which was set to 4 in our experiments. Each partition contained 100,000 items of 100 bytes each. In our experiments on the MSI cluster we conducted evaluations using replication degrees of 3, 4, 5, and full. In case of experiments using Amazon EC2, due to relatively small number of geographic

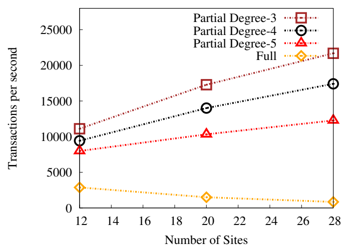

Fig. 3. Transaction Throughput Scalability for In-Memory Database

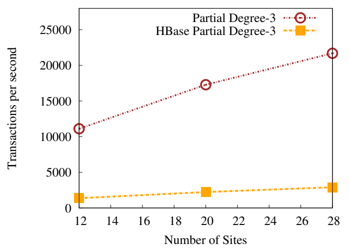

Fig. 4. Transaction Throughput – HBase and In-memory database

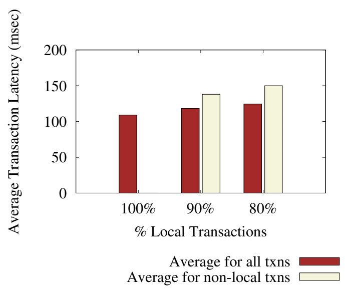

Fig. 5. Impact of Non-Local Transactions

sites, we set the replication degree to 2. For a partition, we designated one of its replica sites as the conflict resolver for the items in that partition.

B. Scalability of PCSI

We evaluated the scalability of the PCSI model for various number of sites. In our experiments on the MSI clusters we set the update propagation period to 1 second, which also accounts for wide-area network latencies, which are typically around 100-200 msec. For each system size, we measured the maximum transaction throughput, average transaction response time, time required for validation with a remote conflict resolver site, and delays in applying updates at remote sites due to causal dependencies.

Figure 3 shows the maximum transaction throughput for the custom benchmark, with all local transactions. This evaluation was performed using the MSI cluster. We observe that PCSI provides near-linear scalability; the maximum throughput achieved scales almost linearly with the number of sites. In contrast to partial replication, full replication offers poor throughput scalability. For example, as we increase the number of sites from 12 to 28, in case of full replication throughput decreased by a factor of roughly 3.4. In contrast, for partial replication with degree 3, throughput increased by 1.95. In these experiments the transaction response times were around 100 milliseconds, and the 2PC validation time was in 70-90 milliseconds range. Even when a transaction accesses all local partitions, it may still involve some remote sites in the 2PC validation phase. Figure 4 shows maximum throughput with replication degree of 3 with HBase as the storage backend, and also shows in-memory throughput for the same configurations.

C. 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. In this evaluation we varied the percentage of non-local transactions to 0, 10, and 20. Figure 5 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 non-local transactions, however, these latencies can be higher in wide-area environments. The reason for small difference in the performance of local and non-local transactions is that a local transaction can involve communication with some remote conflict resolver sites during the validation phase.

D. Impact of Propagation Period on Update Delays

A transaction's updates are applied at a site only after all of its causally preceding transactions have been applied. We refer to such delays as *causal delay*. Furthermore, updates also incur delays due to queueing at remote sites. We refer to the total amount of delay at a remote site as *update delay*. A transaction update also incurs queueing delays at the sending site, which we refer to as *propagation delay*. These delays depend on the propagation period, i.e. the periodic interval of transmitting updates to remove sites.

The visibility latency of a transaction is the sum of propagation delay, network delay, and update delay. Increasing the propagation period increases the latency in applying a transaction's updates at remote sites, thereby delaying updates of other causally dependent transactions. We conducted experiments to determine how these delays are impacted by the propagation period. Here we used a system with 28 sites and replication degree of 5, under a moderate load (60% of the max throughput).

TABLE I
IMPACT OF PROPAGATION FREQUENCY

Propagation Period (sec)	Propagation Delay (msec)	Causal Delay (msec)	Update Delay (msec)	Commit Rate %
1	968	41	41.5	99.3
2	1929	83.3	84	98.8
4	3801	275	276	97.8

Table I shows the impact of the propagation period on propagation delay, causal delay, and update delay. In this experiment, the read/write set size was set to 8 reads and 2 writes. Both the propagation delay and causal delay increase with increase in propagation period.

E. Evaluations on EC2

Table II shows the throughput scalability data for evaluations performed on Amazon EC2. In case of partial replication, the response times remain roughly constant with increase in the number of sites. The throughput with partial replication is higher than the throughput with full replication. The partial

TABLE II
SYSTEM PERFORMANCE ON AMAZON EC2

Num. of Sites	Max Throughput (txns/sec)	Avg. Response Time	Avg. Visibility Latency
Partial Replication			
4	1730	208 ms	10.82 sec
8	3012	231 ms	10.16 sec
Full Replication			
4	738	324 ms	18.76 sec
8	910	391 ms	31.1 sec

replication configuration provides lower response times compared to full replication. Moreover, in case of full replication, the response times typically increase with increase in the number of sites. The higher visibility latencies observed here were mainly due to wide-area communication latencies and the EC2 instances were much less powerful than the MSI nodes.

VII. CONCLUSION

We have presented here the Partitioned Causal Snapshot Isolation (PCSI) model for transaction management in partially replicated databases with asynchronous update propagation. The PCSI model is based on a weaker form of snapshot isolation providing causal consistency. We have elaborated here the unique issues that are raised due to partial replication in supporting causal consistency with the snapshot isolation model. The PCSI model addresses these issues and provides a transaction management protocol which ensures causal consistency and requires sending update propagation messages only to the sites storing the modified partitions. Our evaluations show that partial replication using the PCSI model provides significantly better scalability compared to full-replication.

Acknowledgments: This work was supported by NSF Award 131933, Minnesota Supercomputing Institute, and Grant-in-Aid award by University of Minnesota.

REFERENCES

[1] Amazon, "Amazon EC2, http://aws.amazon.com/ec2/."
[2] Apache, "Hbase, http://hbase.apache.org/." [Online]. Available: http://hbase.apache.org/
[3] M. S. Ardekani, P. Sutra, and M. Shapiro, "Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems," in *Proc. of SRDS '13*, 2013, pp. 163–172.
[4] 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 *Proc. of the ACM SAC*, 2008, pp. 2181–2185.
[5] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "The potential dangers of causal consistency and an explicit solution," in *Proc. of the Third ACM Symp. on Cloud Computing*, 2012, pp. 22:1–22:7.
[6] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proc. of ACM*, ser. SIGMOD '13, 2013, pp. 761–772.
[7] 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.
[8] 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.
[9] 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.
[10] J. C. Corbett and et al, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, Aug. 2013.
[11] K. Daudjee and K. Salem, "Lazy database replication with snapshot isolation," in *Proc. of the VLDB*, 2006, pp. 715–726.
[12] 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.
[13] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13, 2013.
[14] 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.
[15] U. Fritzke, Jr. and P. Ingels, "Transactions on partially replicated data based on reliable and atomic multicasts," in *Proc. of the Intl. Conference on Distributed Computing Systems (ICDCS'01)*, 2001, pp. 284–291.
[16] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proc. of ACM SIGMOD'96*, 1996, pp. 173–182.
[17] J. Holliday, D. Agrawal, and A. El Abbadi, "Partial database replication using epidemic communication," in *Proc. of the Intl. Conference on Distributed Computing Systems (ICDCS'02)*, 2002.
[18] H. Jung, H. Han, A. Fekete, and U. Roehm, "Serializable Snapshot Isolation for Replicated Databases in High-Update Scenarios," in *VLDB*, 2011.
[19] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "MDCC: Multi-data Center Consistency," in *Proc. of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13, 2013, pp. 113–126.
[20] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *Proc. of USENIX OSDI'12*, 2012, pp. 265–278.
[21] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris, "Enhancing edge computing with database replication," in *Proc. of the IEEE Symposium on Reliable Distributed Systems*, 2007, pp. 45 –54.
[22] 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.
[23] 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.
[24] ——, "Stronger semantics for low-latency geo-replicated storage," in *Proc. of USENIX NSDI'13*, 2013, pp. 313–328.
[25] V. Padhye, "Transaction and data consistency models for cloud applications," Ph.D. dissertation, University of Minnesota, Minneapolis, November 2013.
[26] V. Padhye and A. Tripathi, "Causally Coordinated Snapshot Isolation for Geographically Replicated Data," in *Proc. of IEEE SRDS*, 2012, pp. 261–266.
[27] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distributed Parallel Databases*, vol. 14, no. 1, Jul. 2003.
[28] N. Schiper, R. Schmidt, and F. Pedone, "Optimistic Algorithms for Partial Database Replication," in *Proc. of OPODIS*, 2006, pp. 81–93.
[29] N. Schiper, P. Sutra, and F. Pedone, "P-store: Genuine partial replication in wide area networks," in *IEEE SRDS*, 2010, pp. 214–224.
[30] D. Serrano, M. Patiño Martínez, R. Jiménez-Peris, and B. Kemme, "Boosting database replication scalability through partial replication and 1-copy snapshot isolation," in *Proceedings of PRDC.07*, 2007.
[31] ——, "An autonomic approach for replication of Internet-based services," in *Proc. of the IEEE SRDS*, 2008, pp. 127–136.
[32] A. Sousa, F. Pedone, R. Oliveira, and F. Moura, "Partial replication in the database state machine," in *Proc. of NCA'01*, 2001.
[33] 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.
[34] P. Sutra and M. Shapiro, "Fault-tolerant partial replication in large-scale database systems," in *Proc. of the Euro-Par Conference on Parallel Processing*, 2008, pp. 404–413.
[35] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Understanding replication in databases and distributed systems," in *Proc. of IEEE ICDCS*, 2000, pp. 464–474.
[36] 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.