Scalable Transaction Management for Partially Replicated Data in Cloud Computing Environments

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

***Abstract*—We present here a scalable protocol for transaction management in key-value based multi-version data storage systems supporting partial replication of data in cloud and cluster computing environments. We consider here systems in which the database is sharded into partitions, a partition is replicated only at a subset of the nodes in the system, and no node contains all partitions. The protocol presented here is based on the Partitioned Causal Snapshot Isolation (PCSI) model and it enhances the scalability of that model. The PCSI protocol is scalable for update transactions which involve updating of only local partitions. However, it faces scalability limitations when transactions update non-local partitions. This limitation stems from the scheme used for obtaining update timestamps for remote partitions, causing vector clocks to grow with the system configuration size. We present here a new protocol based on the notion of sequence number escrow and address the underlying technical problems. Our experimental evaluations show that this protocol scales out almost linearly when workloads involve transactions with remote partition updates. We present here the performance of this protocol for three different workloads with varying mix of transaction characteristics.**

I. Introduction

Replication management techniques in large-scale systems have to make certain fundamental trade-offs between data consistency, scalability and availability. There are several factors that can affect the scalability of replication management techniques. A high degree of replication can become a major obstacle in achieving scalability as noted in [10]. Full replication is undesirable in large-scale systems. This has motivated investigation of techniques for managing data with partial replication [9], [11], [17], [16], [18], [19], i.e. a data item is not required to be replicated at all nodes, and a node contains replicas of only a subset of the items in the database.

Another obstacle in regard to scalability is the requirement of strong consistency, as in serializable transactions, which can require distributed coordination such as the execution of a two-phase commit protocol involving multiple nodes. This has motivated designs of many large-scale systems to adopt weaker yet useful models of consistency [6], [4], which can be supported using asynchronous replication management techniques. The weak consistency based models for replication management in large-scale systems have ranged from eventual consistency [6], per-record timeline consistency [4], causal consistency [13], and snapshot isolation (SI) [3] with causal consistency [20].

The focus of this paper is on the development of a scalable protocol for providing transaction support in partially replicated databases in large-scale systems. We consider here systems in which a database is sharded into multiple disjoint partitions. Each partition stores a set of items using the key-value based multi-version data storage model. A partition is replicated at a subset of the nodes (sites) in the system, and a node may contain any number of partitions. A transaction may be executed at any node and it may read or modify items in any of the partitions, local or remote. The protocol which we present here is based on our previous work [15] on the PCSI (*Partitioned Causal Snapshot Isolation*) model.

The PCSI model is based on snapshot isolation (SI) for transaction execution, which is an optimistic concurrency control technique. A transaction obtains a snapshot time when it begins execution. It reads latest committed versions of the data items as per its snapshot time, and all updates are made to local copies of data in the transaction's buffer memory. The transaction then goes through a validation phase which allows it to commit only if there are no concurrent *write-write* conflicts. When a transaction is committed, it obtains an update sequence number from each of the partitions that it modifies. The commit timestamp of a transaction is a vector clock consisting of these sequence numbers. The transaction's updates are then applied to the local partitions, and then its updates are propagated asynchronously to other nodes which contain the replicas of the updated partitions. At a remote node, the transaction updates are applied according to their causal ordering. For this purpose a vector clock is maintained for each partition and the length of the vector clock is equal to the number of its replicas.

In this paper we address the scalability limitations of the PCSI protocol presented in [15]. It is scalable when transactions update only the items in the local partitions at their execution sites. It is also scalable for workloads in which a partition is updated by transactions executing only at some bounded number of remote nodes that do not have that partition. For workloads in which any partition may be updated by transactions executing at any of the sites in the system, the scalability of the PCSI protocol was found to be limited to about 20 sites. This scalability limitation of the PCSI protocol motivated us to re-examine its mechanism for supporting transactions updating non-local partitions. It

2159-6190/16 $31.00 © 2016 IEEE
DOI 10.1109/CLOUD.2016.41

creates at the transaction's execution site a *ghost replica* of the remote partition, which does not store any data and no updates are propagated to it. Its purpose is to locally assign update sequence numbers. We found that this mechanism causes the vector clock of a partition to become as big as the system size, having an element for each of the ghost replicas, in addition to those for the real replicas. This becomes the major factor in limiting the scalability of the protocol, as the vector clock related computations start affecting the performance.

We developed a new protocol for supporting non-local partition updates in the PCSI model to make it scalable when partitions may be updated from any of the nodes in the system. Our experimental evaluations show its performance scalability on a cluster of 100 nodes. We conducted these evaluations using three different workloads, each with five benchmarks of varying mix of transaction characteristics. The approach taken in this protocol is based on obtaining the sequence number for a transaction's updates to a remote partition from one of its replicas. This approach raises several technical issues which we elaborate upon and address in this paper. One is to ensure that the local transactions at the replica's site which grants a sequence number to a remote transaction are not stalled due to the causal ordering of transaction updates. To address this issue, we develop a scheme based on the notion of *escrow-based sequence numbering*. When granting a sequence number to a remote transaction, a replica reserves as escrow a range of sequence numbers for local transactions and gives out to the remote transaction an advance sequence number following the escrow range. We identify and address the various coordination issues that arise in adopting this approach. Another issue that arises is the possibility of deadlocks when several transactions update a common set of multiple remote partitions. Deadlocks can arise in applying updates of two transactions which obtained remote sequence numbers from a common set of partition replicas but the sequence order implied by two replicas are not consistent. Addressing this problem requires that if two or more transactions request sequence numbers from a common set of remote partition replicas, then their requests must be handled in the same order by all of them.

In the next section we discuss the related work on transaction support for partially replicated data in cloud/cluster computing systems. Section III presents a brief overview of the PCSI model. In Section IV we present the escrow based advance sequence number assignment scheme for transactions updating remote partitions. We elaborate on the various problems that arise in adopting this approach and present our design to address these problems. In Section V we present the results of our performance evaluation experiments and demonstrate the scalability of this new protocol.

II. Related Work

Several data management systems for cloud datacenters distributed across wide-area have been developed in the recent years [6], [4], [2], [13], [20]. Dynamo [6] uses asynchronous replication with eventual consistency, whereas PNUTS [4] provides a stronger consistency level than eventually consistency, called as *eventual timeline consistency*, but both these systems do not support transactions. Megastore [2] provides transactions over a group of entities using synchronous replication. COPS [13] provides causal consistency for snapshot-based read-only transactions. Eiger [14] provides both read-only and update transactions with causal consistency, maintaining causal dependencies on per object level. PSI [20] provides snapshot isolation based transaction support guaranteeing causal consistency.

The techniques for transaction management in systems with partial replication of data have been investigated by others in the past [11], [19], [16], [9], [18]. The approach presented in [11] uses epidemic communication that ensures causal ordering of messages using vector clocks. Other approaches [19], [18], [17] are based on the state machine model, utilizing atomic multicast protocols. These approaches support *1-copy serializability*. Non-Monotonic Snapshot Isolation (NMSI) [1] and PCSI [15] are both based on the causal snapshot isolation model.

Orbe [7] supports partitioned and replicated databases with causal consistency, but it does not support multi-key update transactions. It uses two-dimensional vector clocks for capturing causal dependencies. GentleRain [8] uses physical clocks to eliminate dependency check messages, but similar to Orbe it lacks a general model of transactions. Spanner [5] 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 [12] provides a consistency scheme called *red-blue consistency*, which uses operation commutativity to relax certain ordering guarantees for better performance.

III. Background: Overview of the PCSI Model

We first present here an overview of the PCSI model for transaction management in partially replicated key-value based data storage systems. More detailed description of this model can be be found in [15]. The PCSI model provides the following guarantees. A transaction's updates are applied at a site only after the updates of all transactions causally preceding it have been applied there. Atomicity of a transaction's updates to multiple partitions is ensured, i.e. either all or none of the updates of a transaction become visible at a site. If two or more concurrent transactions update the same item, then only one is able to commit and all others are aborted, following the snapshot isolation model. This means that updates to an item are total ordered. All updates to a replica are total ordered, but different items in the same partition can be concurrently updated at different replicas.

A. Partition Vector Clocks and Causal Dependency View

Each site is identified by a unique $siteId$. A site maintains a sequence counter for each of its local partitions, which is used to assign sequence numbers to the local transactions modifying items in that partition. 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, by the replica of that partition at the site identified by $siteId$. For this, each partition replica maintains a monotonically increasing counter called *Sequence Number (SN)*, which is used for assigning sequentially increasing commit timestamps. The *commit timestamp vector* ($\mathcal{C}_t$) of transaction t is a set of timestamps assigned to the transaction by the replicas of the modified partitions. For a partition q modified by transaction t, the commit timestamp assigned by q is denoted by $\mathcal{C}_t^q$.

Each replica of a partition maintains a vector clock referred to as the *partition view ($\mathcal{V}_p$)*. The vector clock of a partition replica indicates the sequence numbers of the transactions from the other replica sites which have been applied locally.

B. Transaction Execution

Before executing read/write operations, a transaction must obtain a globally consistent snapshot for the partitions to be accessed, as described in [15]. The snapshot consists of a vector clock value for each partition. All read operations on a partition are performed according to the snapshot obtained for that partition. The writes are buffered till the commit time. When a transaction is ready to commit, it executes a validation protocol to check for update conflicts with concurrently committed transactions.

C. Transaction Validation

For each data item, there is a designated *conflict resolver* which is responsible for checking for concurrent update conflicts for that item. The conflict resolver maintains an ordered list of the commit timestamps ($<siteId, seq>$) of all the committed versions of the item. The transaction coordinates with the conflict resolvers for all items in its write-set to check that no concurrent transaction has created an item version newer than the latest version visible in the transaction's snapshot. The transaction commits only if none of the items in its write-set have an update conflict, otherwise it is aborted. This coordination is done using a two-phase-commit (2PC) protocol. In the first phase, a conflict resolver votes "yes" if no conflicts are found, and locks the item to prevent other concurrent validations or updates. On successful validation, i.e. when all votes are "yes", the transaction is committed and a commit timestamp is obtained from a replica of each of the partitions being updated by the transaction. These timestamps form the commit timestamp vector $\mathcal{C}_t$ for the transaction. The commit/abort decision, along with the commit timestamp in case of commit, is communicated to the conflict resolvers. The conflict resolvers release the lock and record the new version's timestamp in case of commit.

D. Update Propagation

When a transaction commits, its updates are applied to local partitions and then the update messages are sent to the remote sites using the update propagation mechanism described below. For ensuring causal consistency, the causal dependencies of the transaction are computed and this information is communicated with the update propagation message. The causal dependencies of transaction t are captured by a set of vector clocks, called *transaction dependency view* ($\mathcal{TD}_t$). A vector clock $\mathcal{TD}_t^p$ in this set corresponds to partition p and identifies all the causally preceding transactions pertaining to that partition. With the update propagation message for transaction t the $\mathcal{TD}_t$ and $\mathcal{C}_t$ values are communicated to the remote sites. A remote site applies the updates only if it has applied updates of all the causally preceding transactions.

At the remote site, say site k, 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 modified partition p for which the remote site stores a replica, the site checks if $\mathcal{V}_p$ of the replica contains all the events preceding the sequence number value present in $\mathcal{C}_t^p$, i.e., for each modified partition p the following check is done.

$$\mathcal{V}_p[\mathcal{C}_t^p.siteId] = \mathcal{C}_t^p.seq - 1 \tag{1}$$

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 transaction t to any non-local partitions are ignored. Furthermore, when applying the transaction t, the partition dependency views of the modified partitions are updated using $\mathcal{TD}_t$ and $\mathcal{C}_t$ values as described in [15].

E. Remote Partition Update

As noted earlier, in our previous work [15] we used the notion of *ghost replica* to support updates to remote partitions by a transaction. 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 its main function is to assign local commit timestamps to transactions at its site, using a local sequencer. No updates are propagated to the ghost replicas of any partition.

IV. Escrow based remote update protocol

We found that the ghost partition based approach for updating non-local partitions limited the scalability of the PCSI protocol when partitions could be updated by transactions executing at any of the sites. The scalability gets limited because the vector clock of a partition tends to become as large as the number of sites. This motivated us to explore an alternate approach, which we present below. A transaction updating items in a non-local partition gets the sequence number from one of the sites which has a replica of that partition. With this approach, consider the case where a site, say S_1, has the sequence number (SN) 100 for partition P_1, and a transaction at some remote site S_2 needs a sequence number from S_1 for partition P_1. If S_1 gives the "next" sequence number 101, then this approach has a major drawback because all the local transactions at S_1 will stall (as they cannot get the next sequence number) until the remote transaction commits at site S_2 and its update is applied at site S_1. This is because the

PCSI protocol orders transactions updating a partition replica using the sequence numbers assigned by the replica.

The stalling of local transactions as noted above affects the transaction response times and system throughput. The number of stalled transactions depends on several factors. It depends on the local load and the time it takes for a remote transaction's update to be applied at the site issuing the sequence number.

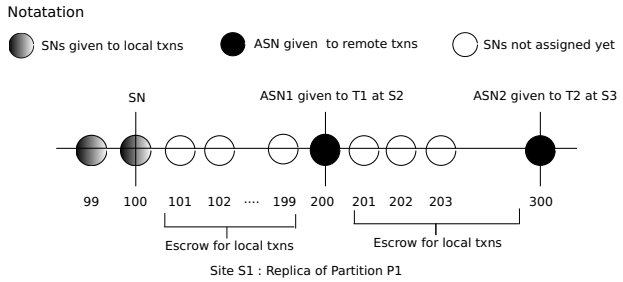

Fig. 1: Notion of ASN and Escrow

To address the problem mentioned above, the approach we adopted is to reserve a range of sequence numbers for local transactions when a transaction from a remote site requests a sequence number. This reserved range of sequence numbers serves as an *escrow* for local transactions. A remote site requesting a sequence number is given a number after the escrow range. We call it *Advance Sequence Number* (ASN). Figure 1 shows that the current sequence number of the replica of partition P_1 in site S_1 is 100, and the escrow value is set to 100. When remote transactions T_1 from site S_2 and T_2 from site site S_3 request for sequence numbers from P_1, ASNs 200 and 300 are given out to the T_1 and T_2 respectively. With this assignment of sequence numbers, T_1 causally precedes T_2, denoted by $T_1 \prec T_2$.

Figure 2 shows the sequence of events involved in the transactions T_1 and T_2: obtaining ASN, committing the transaction, and propagating the updates. The notation T_1:$w(P_1, P_2)$ means that the transaction T_1 updates partitions P_1 and P_2. Note that as $T_1 \prec T_2$, T_1's update will be applied at S_1 before applying T_2's update, even if T_2's update reaches S_1 first, thus guaranteeing transaction ordering.

In our implementation of the escrow technique, each partition replica keeps track of two sequence numbers: *Sequence Number* (SN), as in the PCSI model, and the last *Advance Sequence Number* that was given out by this replica, ASN_{last}. Whenever a remote transaction requests an ASN, it is computed using the formula given below:

$$ASN_{next} := max(SN, ASN_{last}) + escrow \qquad (2)$$

Each partition replica also maintains an ASN List, a list of sequence numbers that the partition has given out to other site(s). Also, ASN_{first} is the first ASN in the ASN List. Naturally, the next question, which we address below, is how to set the escrow value.

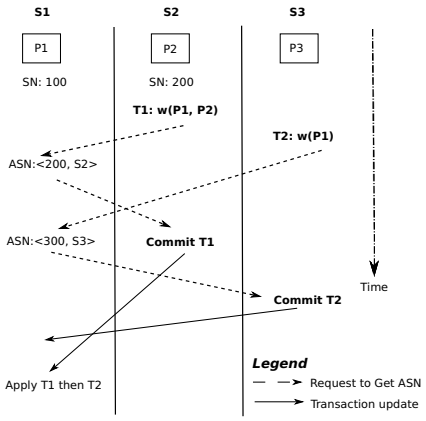

Fig. 2: Single ASN Scenario

A. Setting of Escrow Value

Case 1: If the value of escrow is too small with respect to the local transaction execution rate, then the local transactions will consume all numbers in the escrow range before the update of the remote transaction to which the sequence number ASN_{first} was assigned is applied to the local replica. This will result in increased stalling of local transactions, therefore leading to reduced throughput and commit rate at the local site. In our design we abort such stalled transactions instead of blocking them while they are holding locks on conflict resolvers, as required by the validation protocol.

Case 2: If the value of escrow is too big, local transactions will not fill the escrow range. By the time the update of the remote transaction with the sequence number ASN_{first} for the partition reaches the partition replica, we've two options. One option is to queue the remote transaction and let the local transactions use the remaining escrow range. The second option is to generate a special message to fill the remaining escrow range and to advance the vector clock of the partition. This message is then propagated to all the other sites that have the partition's replica. The former option is undesirable because it blocks all transactions that causally follow the remote transaction, thus causing increased queuing delay and reduced throughput.

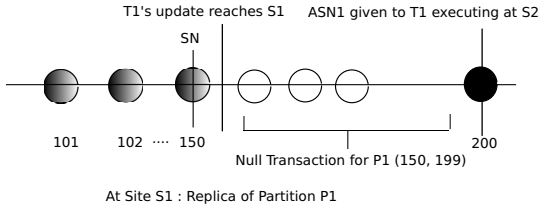

Fig. 3: Generation of Null Transactions

To help understand the second scenario, we consider transaction T_1 from the example presented in Figure 2. T_1 executed at site S_2 and obtained ASN value 200 from site S_1 for updating partition P_1. Figure 3 shows a scenario for the state of site S_1 when it receives the update of T_1 from S_2. In this

scenario the current SN of P_1 at site S_1 is 150. If all the causal dependencies for the remote transaction are satisfied, we would like to apply the transaction right away. Hence, we need to advance SN for the local replica of P_1 to $(ASN_{first} - 1)$, i.e. 199, and inform the other replicas of P_1. For this, we send a special update message to the other replicas which says: if the value of vector clock $\mathcal{V}_{P_1}[S_1] = 150$, then increment the value to 199. We call this special message *null transaction*, and its sole purpose is to advance the vector clocks.

We set the escrow value used by a replica based on the local update rates. For this purpose, we observe for each assigned ASN the number of local transactions that commit or abort due to stalling during the period when the ASN is given out to a transaction and its update is received and applied at the local site. We compute the average value of this number over a sliding window. This number indicates the local execution rate, and the escrow value is periodically set proportional to this number, with a factor slightly greater than 1.

B. Protocol for Single Remote Partition Update

A transaction updating some remote partition requests an ASN from a site which has a replica of that partition. To obtain ASN for partition P from its replica at a remote site, transaction T executing at site S invokes the remote interface function *GetSingleASN* at that remote site. This function is simple as shown below in Algorithm 1. The ASN is generated by adding an escrow to the maximum of the current SN or the last ASN value given out by the replica.

Algorithm 1 Function for obtaining single ASN

```
Initial state:: ASNList is empty
function GETSINGLEASN(T, P, S)
    // T requesting ASN for partition P
    // T is executing at site S
    [ begin atomic region
    ASN_next ← max(SN, ASN_last) + escrow
    ASN_last ← ASN_next
    ASNList.appendRecord( <ASN_next, S> )
    end atomic region ]
    return ASN_next
```

When the transaction T's update from site S reaches the site that gave out the ASN for partition P, P's ASN_{first} is checked against the T's commit timestamp component corresponding to P, i.e. C_T^P. If $C_T^P.seq$ is equal to ASN_{first} of partition P, and if all the other causal dependencies for T are satisfied, then a *null transaction* is generated to fill the remaining unconsumed part of the escrow range, the transaction's update is applied locally, and the $<ASN_{first}, S>$ entry is removed from the ASN List.

C. Issues with Multiple Remote Partition Update

We now illustrate the problems that can arise with the ASN scheme when multiple remote partitions are updated by a transaction. When two transactions concurrently update a common set of remote partition replicas, if the ASNs are not handed out in a total ordered fashion there is a potential of deadlocks when trying to apply the transactions' updates at remote sites.

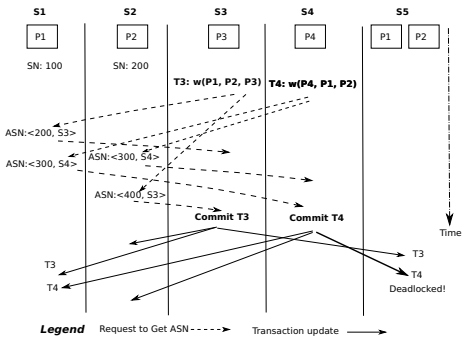

Fig. 4: Multiple ASN Scenario

In Figure 4, transactions T_3:$w(P_1, P_2, P_3)$ and T_4:$w(P_4, P_1, P_2)$ get the ASNs from sites S_1 and S_2 for partitions P_1 and P_2 respectively. T_3 gets the ASN 200 for partition P_1 from site S_1 before T_4, whereas T_4 gets the ASN 300 for partition P_2 from site S_2 before T_3. This implies that $T_3 \prec T_4$ for partition P_1, but $T_4 \prec T_3$ for partition P_2. Moreover, when the update messages of T_3 and T_4 reach site S_5, both updates get stuck. This is because w.r.t P_1, $T_3 \prec T_4$ and w.r.t P_2, $T_4 \prec T_3$, and hence both transactions cannot be applied, resulting in deadlock. Consequently, what this means is that all transactions that have a sequence number obtained from site S_1, after transaction T_3, for partition P_1 or a sequence number from site S_2, after transaction T_4, for partition P_2 cannot be applied at S_5.

A solution to this problem is to total order the transactions which update a common set of partition replicas. This means that the requests for obtaining ASNs by transactions with common set of partition replicas must be total ordered.

D. Protocol for obtaining ASNs from Multiple Partitions

We present here the protocol for obtaining ASNs when a transaction updates multiple remote partitions. For each partition, it selects a remote replica site for obtaining ASN. The transaction sequentially requests ASNs from remote partition replicas in an ascending order of their partition ids. It calls the remote interface function *GetMultipleASN* at each of the replicas' sites, shown in Algorithm 2. Each such invocation request includes a list called $\mathcal{P}_{list}$ containing $<PartitionId, siteId>$ for all of the remote partition replicas from which T is requesting ASNs. The request message also contains a predecessor transaction list, called $\mathcal{T}_{list}$, which is explained next. The response for an ASN request will contain the ASN itself and an updated $\mathcal{T}_{list}$. Each transaction, T_i, in the $\mathcal{T}_{list}$ will satisfy all the following conditions:

- T_i obtained ASN from a partition replica P before T
- T_i's updates were not seen by P when it gave ASN to T
- T_i has a common set of remote partition replicas with T

The following information about T_i is stored in the $\mathcal{T}_{list}$: its Txn-ID T_i, site-ID S, and $\mathcal{P}_{list}$.

Consider the scenario outlined in Figure 4. When transaction T_4 requests ASN from the partition replica P_1 at site S_1, in addition to the ASN 300 it also receives the predecessor list which contains an entry for T_3. When T_4's request for ASN reaches the partition replica P_2 at site S_2, P_2 looks at the predecessor list and finds that T_3 should be given ASN before T_4 to ensure total ordering. It checks if it has already *seen* T_3, i.e. it has either seen T_3's request and given out an ASN to T_3, or it has decided not to handle in the future any delayed request from T_3, thereby forcing it to abort. If P_2 at site S_2 has not *seen* T_3, we've two options. One option is to make T_4 wait until T_3's request arrives at S_2. This way if T_4 is delayed by a long time period then the $responseTime$ of T_4 takes a hit and so does the throughput at site S_4. The other option, adopted in our design, is to make T_4 wait only for a time-out period, which was set to 200 msec in our experiments. If T_3 arrives before the time-out period, then T_3 precedes T_4 else P_2's ASN_{next} can be given to T_4 and T_3 will be aborted when its request reaches S_2. A pseudocode of this protocol is provided in Algorithm 2.

Algorithm 2 Obtaining ASNs for multiple partitions

Initial state:: PredecessorList and ASNList are empty.
function GETMULTIPLEASN(P, T, S, $\mathcal{T}_{list}$, $\mathcal{P}_{list}$)
// This is executed when transaction T at site S
// requests ASN for partition P at site $mySiteId$
// $\mathcal{T}_{list}$ is the predecessor transaction list of T
// $\mathcal{P}_{list}$ is list of $<PartitionId, siteId>$
if ($seenTxn[T] = ABORT$) **then** return $NULL$
if ($\exists\ T_i \in \mathcal{T}_{list} \mid \{seenTxn[T_i] = NULL \wedge <P, mySiteId> \in \mathcal{P}_{list}$ of $T_i\}$) **then**
wait(time-out period)
[begin atomic region
for all $T_i \in \mathcal{T}_{list}$ **do**
if ($seenTxn[T_i] = NULL$) **then**
$seenTxn[T_i] \leftarrow ABORT$
$ASN_{next} \leftarrow max(SN, ASN_{last}) + escrow$
$ASN_{last} \leftarrow ASN_{next}$
$\mathcal{T}_{list}$.append(PredecessorList)
PredecessorList.appendRecord($<T, S, \mathcal{P}_{list}>$)
ASNList.appendRecord($<ASN_{next}, S>$)
$seenTxn[T] \leftarrow TRUE$
end atomic region]
return $<ASN_{next}, \mathcal{T}_{list}>$

When a site applies a transaction's update to a local replica for which it generated an ASN, its ASN entry is removed from the ASN List. Such a transaction is removed from the PredecessorList of the replica even earlier, when the transaction's update message is received by the site.

E. Null Transaction generation in case of Multiple ASNs

When the number of remote partitions updated by a transaction is more than one, then the generation of *null transaction* requires some additional considerations otherwise it can result in increased abort rates. Consider the scenario shown in Figure 5, T:$w(P_1, P_2, P_3)$ executing at site S_3 updates two remote partitions P_1 and P_2. It obtained the ASN for P_1 from site S_1 and the ASN for partition P_2 from site S_2. Note that the sites S_1 and S_2 both have replicas of partitions P_1 and P_2. Consider the case when transaction T's update reaches site S_1 and the causal dependencies of T on all update partitions but P_1 and P_2 are satisfied. Even if P_1 at S_1 generates a *null transaction* advancing its SN to ($C_T^{P_1}.seq - 1$), which is $ASN_{first} - 1$, site S_1 cannot apply the transaction T because of its causal dependency on partition P_2. This will cause all local transactions updating P_1 at S_1 to stall and abort. A similar situation can happen at site S_2 for partition P_2.

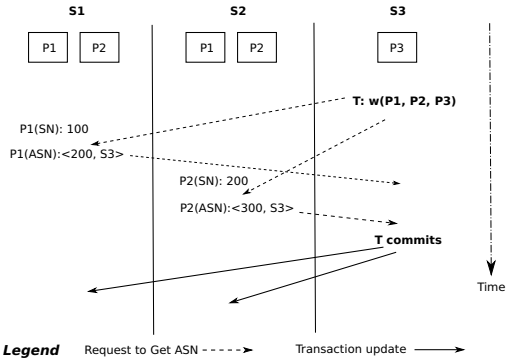

Fig. 5: Increased abort rate due to Multiple ASN

We adopt the following approach to coordinate the generation of the *null transactions* across all the remote update partitions of a transaction. There are two cases to be considered here when a transaction's updates are ready to be applied:

Case 1: Site which generated an ASN does not have replicas of any other remote update partitions. In Figure 5, if site S_1 did not have partition P_2, then this would be the case. In this case, the site generates a *null transaction* to reduce the gap between the current SN and ASN_{first} to 1, without any delay. This is similar to the case of applying updates of a transaction which writes items in a single remote partition.

Case 2: Site which generated an ASN has replicas of all or some of the other remote partitions updated by the transaction. This case corresponds to the scenario in Figure 5, where sites S_1 and S_2 both have replicas of all the remote partitions updated by the transaction T. We want that the gap between the current sequence number SN and ASN_{first} of each of the remote update partition should reduce to 1 within a small real-time window. At each site, a thread periodically checks if the updates can be applied and during this step the gap for a partition is reduced by half its current value, thus retaining some part of the escrow to avoid stalling of local transactions.

V. SCALABILITY EVALUATIONS

We implemented the escrow-based remote write mechanism in the testbed system which we had previously developed for the PCSI model. The focus of our evaluation was on the scalability of the escrow-based protocol for a variety of workloads consisting of transactions which read or modify data

Workload	Benchmarks (Mix of transactions with local and remote read/write partitions)				
	Local Only	All Remote Read	All Remote Write	Remote Read/Write-(50/50)	5% Remote Write
A Read=2 Write=1	All read/write partitions local	1 remote read partition	1 remote write partition	1 remote partition: read/write (50/50)	5% txn: 1 remote write partition
B Read=2 Write=2	All read/write partitions local	1 remote read partition	1 remote write partition	1 remote partition: read/write (50/50)	5% txn: 1 remote write partition
C Read=2 Write=3	All read/write partitions local	2 remote read partitions	2 remote write partitions	2 remote partition: read/write (50/50)	5% txn: 2 remote write partitions

TABLE I: Transaction workloads and benchmarks with different numbers of read/write partitions

items stored in remote partitions. Our focus was on evaluating how the system scaled out with the number of nodes in the cluster. The number of partitions was set equal to the number of nodes, thus larger configurations served larger data-sets to reflect system scale-out. Each partition contained 1 million items, each of 100 bytes. Each partition was replicated on three nodes, and each node contained replicas of three partitions. The period for propagating updates to remote nodes was set to 1 second. We measured the peak throughput achieved for a variety of workloads on different cluster sizes. We performed the scalability evaluations on a computing cluster of 100 nodes. In this cluster, each node had 8 CPU cores with 2.8 GHz Intel X5560 Nehalem EP processors, and 22 GB main memory.

A. Benchmarks and Transaction Workloads

We developed a custom load generator to evaluate the impact of various parameters such as the number of partitions read or written by a transaction, the number of items in a partition read or written, the number of remote partitions accessed and the percentage of transactions accessing remote partitions. We defined three workload classes which we refer to as A, B, and C. These three workload classes represent progressively increasing resource requirements. For each of these three workloads, we defined five benchmarks representing different mix of transactions with varying number of remote partitions. The characteristics of these workloads with the five different benchmark mix are shown in Table I. All transactions involved updating of one or more partitions. Column 1 of Table I shows the number of partitions read and updated in these workloads. All transactions read 4 items from each of 2 distinct partitions. The partitions and items were selected randomly using uniform distribution. The number of partitions updated by the transactions in workload A was 1. This number was 2 for in workload B, and 3 for workload C. The number of items updated in a partition was set to 2.

The characteristics of five benchmarks are summarized in Table I. In the "Local-Only" benchmark all accessed partitions are local. It serves as a reference to measure the impact of remote partition access. In the "All Remote Read" mix, all transactions access some remote partition for reading only. In "All Remote Write", all transactions write some remote partition. Among the five benchmarks, this has the highest resource demand. In "Remote Read/Write (50/50)", all transactions access some remote partition for either reading or writing with equal probability. The benchmark "5% Remote Write" consists of a mix with 5% of the transactions updating some remote partition.

B. Scalability of Escrow-based Remote Write Protocol

For scalability evaluations we measured the peak throughput (txns per sec) of the five benchmarks for each of the three workloads on system configurations consisting of 20, 40, 60, 80, and 100 nodes. The peak throughput was measured when the commit rate was at least 90%. We also measured average transaction response time, time required for two-phase validation, and delays in applying updates at remote sites due to causal dependencies.

Figures 6, 7, and 8 show the peak throughput of the five benchmarks for workload classes A, B, and C, respectively. For a given system size, the throughput for Workload A is higher than those for B and C across all benchmarks. Figure 6 shows the scalability of all benchmarks for workload A. As expected, "Local-Only" has the highest throughput and "All Remote Write" has the lowest. The throughput for the other three benchmarks was between these two. In workload A, for the "Local-Only" benchmark, updates are propagated to 2 other nodes, whereas for "All Remote Write" mix this number is 3. The transactions with remote partition update also incurs the overheads of the ASN protocol. Similar scalability trends are reflected for all benchmarks in workloads B and C. Performance scales out across all benchmarks. In workload B a transaction updates 2 partitions and one of these can be remote. The number of nodes to which a transaction's updates are propagated ranges from 3 to 5. We find that this number impacts the throughput. For this reason, workload C has the lowest throughput. A transaction updates 3 partitions, and 2 can be remote. The number of nodes to which the updates are propagated ranges from 6 to 8. "All Remote Write" mix also incur overheads of multiple ASN coordination.

Transaction workload	Response time avg (median) msec	Validation time avg (median) msec	Update delay avg (median) secs
A	34.99 (15)	4.05 (3)	70 (41)
B	44.85 (20)	8.35 (5)	80 (49)
C	47.05 (26)	12.26 (10)	114 (59)

TABLE II: Response time and latencies

Table II shows the average and median values for the response time, validation latency, and delays in applying remote updates at a node due to causal dependencies. Response time

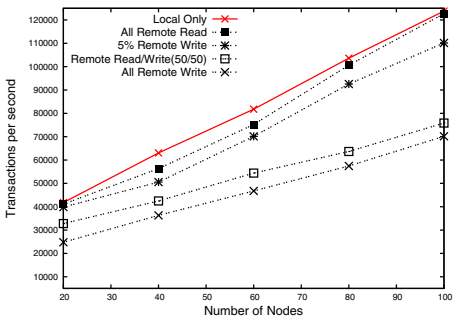

Fig. 6: Throughput for workload A

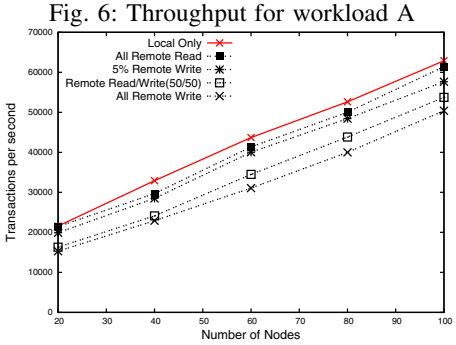

Fig. 7: Throughput for workload B

is measured from the start of a transaction to the time it is locally applied and queued for propagation to remote nodes. We also measure the delay incurred in performing two-phase validation with the conflict resolvers. The update delay is the time for which a remote transaction's updates are buffered locally because causal dependencies are not satisfied. The average response times ranged between 34 to 47 msec for these three workloads. The median values were close to 20 msec. Validation time increased in the workloads with higher number of remote update partitions. The causal delays tend to be high in the range of several seconds, and this increased for workloads with more update partitions.

VI. Conclusion

We have presented here a scalable protocol for transactions with remote partition writes in systems with partial replication of data in cloud computing environments. This protocol enhances the scalability of the PCSI model using the notion of escrow based ordering of remote partition updates while providing causal consistency. We have presented and addressed here the problems associated with the escrow technique. We evaluated the scalability of this protocol using three different workload classes and five benchmarks on a cluster of 100 nodes. These evaluations show that the escrow based remote write protocol provides a scalable approach for supporting transactions in partially replicated data systems.

Acknowledgements: This work was supported by National Science Foundation grant 1319333 and Minnesota Supercomputing Institute.

References

[1] 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.

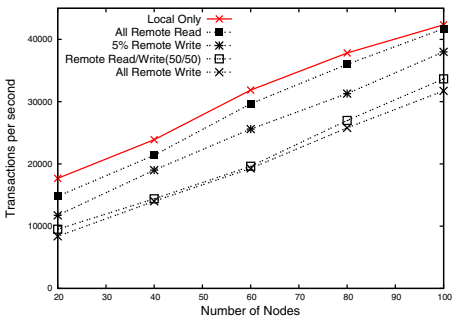

Fig. 8: Throughput for workload C

[2] 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.

[3] 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.

[4] 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.

[5] J. C. Corbett and et al, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, Aug. 2013.

[6] 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.

[7] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *Proceedings of the 4th ACM Symposium on Cloud Computing*, ser. SOCC'13, 2013.

[8] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "GentleRain: Cheap and Scalable Cauasal Consistency with Physical Clocks," in *Proceedings 5th ACM Symposium on Cloud Computing*, ser. SOCC '14, 2014.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] ——, "Stronger semantics for low-latency geo-replicated storage," in *Proc. of USENIX NSDI'13*, 2013, pp. 313–328.

[15] V. Padhye, G. Rajappan, and A. Tripathi, "Transaction Management using Causal Snapshot Isolation in Partially Replicated Databases," in *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS'2014)*, 2014.

[16] S. Peluso, P. Romano, and F. Quaglia, "Score: a scalable one-copy serializable partial replication protocol," in *Proc. of Middleware'12*, 2012, pp. 456–475.

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

[18] N. Schiper, P. Sutra, and F. Pedone, "P-store: Genuine partial replication in wide area networks," in *IEEE SRDS*, 2010, pp. 214–224.

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

[20] 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.