TR 15-009

Design and Evaluation of a Transaction Model with Multiple Consistency Levels for Replicated Data

Anand Tripathi, BhagavathiDhass Thirunavukarasu

May 20, 2015
Design and Evaluation of a Transaction Model with Multiple Consistency Levels for Replicated Data

Anand Tripathi and BhagavathiDhass Thirunavukarasu
Department of Computer Science
University of Minnesota, Minneapolis, Minnesota, 55455 USA
Email: (tripathi, thiru022)@umn.edu

Abstract—We present here a transaction model which simultaneously supports different consistency levels, which include serializable transactions for strong consistency, and weaker consistency models such as causal snapshot isolation (CSI), CSI with commutative updates, and CSI with asynchronous concurrent updates. This model can be useful in managing replicated data with different consistency guarantees to make suitable trade-offs between data availability, performance, and consistency of different data items. Data and the associated transactions are organized in a hierarchy which is based on consistency levels. Certain rules are imposed on transactions to constrain information flow across data at different levels in this hierarchy to ensure the required consistency guarantees. The building block for this transaction model is the snapshot isolation model. We present an example of an e-commerce application structured with data items and transactions defined at different consistency levels. We have implemented a testbed system for replicated data management based on the proposed multilevel consistency model. We present here the results of our experiments with this e-commerce application to demonstrate the benefits of this model.

I. INTRODUCTION

Management of replicated data in distributed systems poses fundamental trade-offs between data consistency, scalability, and availability [18]. In such systems supporting transactions with strong consistency for serializability requires distributed coordination. This imposes scalability and availability limitations in large-scale systems. Replication management protocols with weaker consistency models provide lower latencies in transaction execution and high availability, but guarantee only eventual consistency [14] or causal consistency [33], [27]. 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 [14], [9], [33], [27] have been developed supporting weaker consistency models for replicated data management in large-scale distributed systems. Dynamo [14] uses asynchronous replication with eventual consistency but does not support transactions. PNUTS [9] provides a stronger consistency level than eventually consistent, called eventual timeline consistency. It also supports different consistency guarantees for read operations. However, it does not support transactions. COPS [27] provides causal consistency, but does not provide transaction functionality, except for snapshot-based read-only transactions. Eiger [28] provides both read-only and update transactions with causal consistency but requires maintaining causal dependencies on per object level. PSI [33] supports transactions using a weaker form of snapshot isolation based on causal consistency. The CSI (Causally Coordinated Snapshot Isolation) [30] model is based on the PSI model and it reduces false causal dependencies in update propagation.

Our work is motivated by the observation that rather than providing a single consistency model for transactions in replicated data management system, it is desirable to simultaneously support transactions with different levels of consistency guarantees. This allows in building scalable applications by selecting suitable transaction models for data items with different consistency requirements. Thus one can use low latency transaction management models for data items with weak consistency requirements but at the same time use transactions with strong consistency for critical data. The problem of simultaneously supporting different transaction models and consistency levels has been addressed by some recent research projects [35], [25], [21]. The Gemini system [25] provides a model called red-blue consistency, which uses operation commutativity to relax ordering guarantees. The Salt [35] model provides a framework for simultaneously supporting ACID and BASE transactions in a system and provides rules for their isolation. The framework presented in [21] places data items in different consistency categories, and supports changing the category of data items adaptively at runtime.

In this paper we present a transaction management model for managing replicated data with different consistency models to make suitable trade-offs between availability, scalability, and consistency for different data items. This transaction management model is based on snapshot isolation (SI) [5] with weaker semantics that guarantee causal consistency of snapshots as proposed in [33], [30]. The proposed model simultaneously supports transactions with different consistency models, which include serializability for strong consistency, and weaker models such as CSI (causal snapshot isolation), CSI with commutative updates, and CSI with asynchronous concurrent updates. Consistency requirements are associated with data items, and data is organized in a hierarchy which is based on consistency levels and the associated transaction management protocols. A transaction can be executed at any of the sites. A transaction’s access to data items is restricted by certain constraints, which ensure that the required consistency guarantees are preserved. These constraints impose certain
restrictions on information flow across data layers in the consistency hierarchy. Using the CSI [30] model as a building-block, we introduce additional mechanisms to support serializable transactions for stronger consistency guarantees. We also introduce mechanisms to support two weaker consistency models, one to support commutative operations and the other for supporting asynchronous concurrent updates.

For experimental evaluations of the proposed model we developed a testbed system, implementing it over the transaction management system we had earlier developed for the CSI model for partially replicated data [29]. We present here an example of modeling an e-commerce application to support transactions with different consistency guarantees. We measured the performance of this application by executing transactions with the proposed multi-level consistency model, and compared it with the performance under a single consistency model. Our evaluations show significant benefits of the proposed approach.

The rest of the paper is organized as follows. In the next section we present the related work. Section III presents an overview of the CSI model, which serves as the building-block for the proposed model. In Section IV we present a hierarchy of consistency levels and the corresponding transaction protocols along with the basic principles for ensuring consistency guarantees for data at different levels. Section V describes how the CSI model is extended to support serializable transactions. Section VI describes the mechanisms for supporting concurrent commutative updates. Section VII presents the weakest level in this model where updates require eventual consistency. In Section VIII we present an example of an e-commerce application structured based on the multi-level consistency model. In Section IX we present the results of our performance evaluation experiments conducted using the e-commerce application on our testbed system. Conclusions are presented in the last section.

II. RELATED WORK

The issues with scalability in data replication with strong consistency requirements are discussed in [19]. Such issues can become critical factors for data replication in large-scale systems and geographically replicated databases. This has motivated use of snapshot isolation (SI) [5] and weaker consistency models such as eventual and causal consistency.

The snapshot isolation (SI) model [5] is considered attractive for performance and scalability. It is based on multi-version data management, utilizing optimistic concurrency control [22] in which a transaction reads only committed version of data. Replicated data management using snapshot isolation (SI) has been investigated by several projects [26], [13], [20]. SI-based database replication using lazy replication approach is investigated in [13], however that approach used the primary-backup model. Many of the systems for SI-based database replication [26], [20] use eager replication with atomic broadcast to ensure that the replicas observe a total ordering of transactions. The problem of transaction serializability in snapshot-isolation model has been extensively studied [17], [6], [7], [32], [20]. The work in [17] characterizes the conditions necessary for non-serializable transaction executions in the SI model. Several approaches have been developed to avoid serialization anomalies in SI based transactions.

The snapshot isolation (SI) model poses scalability limitations in wide-area environments because of the serial execution of validation operations. Parallel Snapshot Isolation (PSI) [33] model addresses this issue by providing a weaker model of snapshot isolation based on causal consistency. Validation to check for write-write conflicts for different items can be performed in parallel at different sites. This model imposes a causal ordering on transactions. In the PSI model the snapshot view of different sites can diverge, but eventually they converge to the same view once all updates have been applied. This is called the fork-join model. The CSI [30] model improves upon the PSI model to reduce false causal dependencies. Both PSI and CSI require full replication. The PCSI [29] protocol generalizes the CSI model to support partial replication of data. Several other systems [2], [31] have supported transaction management for partially replicated data using snapshot isolation.

Several distributed data storage systems based on the key-value model have been developed for scalability and availability requirements [8], [14], [23], [4]. Cassandra [23] can support both weak consistency and strong consistency by appropriate configuration of quorum size. Bigtable [8] supports strong consistency but provides only single-row transactions. Other systems provide transactions over multiple rows with certain constraints. For example, Megastore [4], and G-store [11] provide transactions over a group of entities. In ElasTraS [12] ACID transactions are supported only over a single partition. Spanner [10] provides serializable transactions under global-scale replication. However, it relies on special purpose hardware such as GPS or atomic clocks.

The goal of simultaneously supporting different consistency models has been pursued by several projects. The system presented in [3] provides mechanisms for supporting causal consistency in systems with eventual consistency. The Red-Blue consistency model of the Gemini system [25] requires analysis of the transaction operations to split a transaction into two parts, one containing commutative shadow operations. The Salt [35] model requires rewriting an ACID transaction as a BASE transaction consisting of a series of alkaline nested transactions. Both Gemini and Salt models require analysis and rewriting of application level transactions. The approach presented in [21] places data into different consistency categories and supports adaptively changing the category of an item based on its access patterns. Consistency categories are associated with data and not transactions. This system provides probabilistic guarantees of consistency, which may get violated at times. Our model associates consistency levels with both data and transactions, and it ensures that the consistency guarantees are always satisfied.
III. BACKGROUND: CAUSAL SNAPSHOT ISOLATION (CSI)

The CSI model is based on a weaker form of snapshot isolation model proposed in PSI [33]. The proposed multi-level model is implemented in our testbed system with partial replication using the PCSI [29] protocol. However, for the sake of simplifying our presentation here we will assume that the data items are fully replicated.

A. System Model

The system consists of multiple geographically distributed database sites $S_i$ such that $i \in (1..n)$. Each site is identified by a unique $siteId$. Each site has a local database that supports multi-version data management. For each data item, there is a designated conflict resolver which is responsible for checking for write-write conflicts for that item. Transactions can execute at any site. Update transactions need to coordinate with conflict resolver sites for write-write conflict checking for the items in their write-sets. Transactions executing at a site are ordered using a local sequence number, and updates are propagated asynchronously to remote sites.

B. CSI Consistency Guarantees

In the CSI model, transactions are ordered according to a causal ordering. This model ensures causal ordering while applying transaction updates at remote sites. The CSI model provides the following guarantees for transaction execution:

- **Snapshot Isolation:** As in the case of traditional snapshot isolation, the CSI model performs write-write conflict detection to guarantee that when two or more concurrent transactions update a common data item, only one of them is allowed to commit.

- **Transaction 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 $T_j$.

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

- **Causally Consistent Snapshot:** A transaction observes a consistent snapshot which has the following two properties of atomicity and causality. In a consistent snapshot either all or none of the updates of a transaction are visible. If a snapshot contains updates of transaction $T_i$, then updates of all transactions causally preceding $T_i$ are also contained in it.

C. Overview of the CSI Protocol

A transaction is assigned a commit sequence number $seqno$ from a monotonically increasing local sequence counter maintained by its execution site. Thus, the commit timestamp for a transaction is a pair $<siteId, seqno>$. A data item version is identified by the commit timestamp of the transaction which creates it. The local sequence number is assigned only if the transaction is guaranteed to commit, i.e. only if there is no write-write conflict. A transaction first commits locally and then its updates are propagated to other sites asynchronously. A remote site, upon receiving a remote transaction’s updates, applies the updates provided that it has also applied updates of all the causally preceding transactions. The transactions from a particular site are always applied in the order of their sequence numbers.

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

**Transaction’s Snapshot Time:** A transaction $t$ executing at site $S_i$ is assigned, when it begins execution, a snapshot timestamp vector $VT_i^t$, which is set equal to the current vector clock $VC_i$ value. When $t$ performs a read operation for item $x$, we determine the latest version of $x$ that is visible to the transaction according to its snapshot timestamp vector.

**Commit protocol:** If transaction $t$ has modified one or more items, then for each such item it performs validation to check for write-write conflicts with other concurrent transactions. This validation is performed as a two-phase validation protocol with the conflict resolvers which are responsible for the items in the transaction’s write-set. In prepare phase, each resolver checks, if the latest version of the item is visible in $t$’s snapshot and that the item is not currently locked by any other concurrent validation request. The locking is performed to avoid conflicts with any concurrent validation requests by other transactions. If this check fails, then the resolver sends a ‘no’ vote. Otherwise, it locks the item and sends a ‘yes’ vote. If the transaction receives ‘yes’ votes from all conflict resolvers, it obtains a commit timestamp and applies updates to the local database. The local site’s vector clock is advanced appropriately. It now sends a commit message, containing the commit timestamp to all the conflict resolvers. Otherwise, in case of any ‘no’ vote, the transaction is aborted and an abort message is sent to the conflict resolvers. Upon receiving a commit or abort message, a conflict resolver releases the locks, and in case of commit it records the latest version number as a 2-tuple: $<siteId, seqno>$. The local site asynchronously propagates $t$’s updates to all the other sites.

**Update propagation:** For ensuring causal consistency, $t$’s updates are applied at remote sites only after all the causally preceding transactions have been applied. The CSI model uses the notion of effective causal snapshot vector ($VT_i^t$) to indicate causal dependencies. This vector is based on the read-set and write-set of the transaction, indicating for each site the latest event from that site which is ‘seen’ by the transaction. The transaction updates are applied at a remote site only when that site’s vector clock is advanced up to this vector.
IV. MULTI-LEVEL CONSISTENCY MODEL

The items in the replicated data store are organized along a hierarchy of consistency levels. A data item can belong to only one level. Similarly each transaction in the system is designated to execute at exactly one of the levels. In Table 1 we present a hierarchy of data consistency levels and the associated transaction management protocols. This table outlines the consistency properties of the transactions at different levels of this hierarchy. We present in this section the rules for ensuring the consistency guarantees for each level.

A higher level in the hierarchy corresponds to a stronger consistency level. The top-most level corresponds to strong consistency level, and all transactions updating data at this level are guaranteed to be serializable. This level ensures ACID properties for transactions. We refer to it as SR level. The next level below this is the CSI model which provides the consistency properties described in Section III. We refer to it as CSI level. The transactions are causally ordered with updates to a data item total ordered. The snapshots at different sites can diverge due to concurrent updates on different items, but when all updates have been applied at two sites then their snapshots become identical. This is the fork-join model [33] for snapshots at different sites.

The next lower level in this hierarchy weakens the CSI model to allow concurrent updates to an item if they are commutative. If multiple concurrent transactions update an item with commutative operations, then they all are able to commit under this model. Such updates may get applied at different sites in different order, thus resulting in the fork-join model even with respect to updates to a single item. We refer to this consistency level as CSI-CM level.

The lowest level in the hierarchy allows asynchronous updates to an item. We refer to it as ASYNC level. Conflict checking is not required for updates to data items at this level. This consistency level is suitable for appending records to logs or inserting items in a set.

A transaction executes at a specific level in this hierarchy. The following rules are enforced to ensure the consistency properties of the data items organized in different consistency levels by constraining information flow across levels.

- **Read-Up** A transaction at a level in this hierarchy can read only those data items that are at the same level or at stronger consistency levels in this hierarchy.
- **Write-Down** A transaction can update items that are at its own level or at weaker consistency levels.

These rules are inspired by similar kinds of models in the area of information security, such as the Bell-LaPadula model. We refer to these rules as read-up/write-down. These rules basically prevent a transaction from reading information from weaker consistency level data items to update a stronger consistency level data item. A transaction can update data items that are at its own level or at weaker consistency levels. For example a transaction at the SR level can read items only at the SR level but it can update items at any of the levels.

A transaction at the SR level is guaranteed to be serializable with other transactions at the SR level, but only with respect to the data items at this level. If a transaction at the SR level also updates certain data items that belong to weaker consistency levels, then such a transaction may not be serializable with other SR level transactions, with respect to such data items. For example two transactions at the SR level may also append records to certain log-files which belong to the ASYNC consistency level. The order of the records appended by these transactions to different log-files may not conform to their serialization order at the SR level. The consistency properties of data at the SR level are guaranteed as no information flows from weaker consistency levels to the data items at this level.

In the proposed model, transactions belonging to different consistency levels can execute simultaneously. Transactions at any of the levels always execute with the basic protocol of the causal snapshot isolation model, but with different conflict resolution policies. Therefore, a transaction exhibits the isolation properties of SI based transactions [5]. A transaction always reads committed data, and the updates of a transaction become visible only when it commits. The consistency property of snapshots guarantees that either all or none of the updates of a transaction are visible in a snapshot. Moreover all updates are applied at remote sites in their causal order. In the next sections we present how the proposed model can be implemented by building upon the CSI model. We first outline the mechanisms

<table>
<thead>
<tr>
<th>Level</th>
<th>Consistency Properties</th>
<th>Transaction Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>Strong consistency - Globally serializable transactions; ACID properties for transactions</td>
<td>SI-based serializable transactions</td>
</tr>
<tr>
<td>CSI</td>
<td>Causal ordering of transaction updates; Per-item update ordering;</td>
<td>Causal Snapshot Isolation</td>
</tr>
<tr>
<td></td>
<td>Fork-join model of snapshots for concurrent updates on different items</td>
<td></td>
</tr>
<tr>
<td>CSI-CM</td>
<td>Causal ordering of transaction updates; Permits concurrent commutative updates on an item; Fork-join model of snapshots for all concurrent update including commutative concurrent updates on an item</td>
<td>Causal Snapshot Isolation with commutative updates</td>
</tr>
<tr>
<td>ASYNC</td>
<td>Causal ordering of transaction updates;</td>
<td>Causal Snapshot Isolation</td>
</tr>
<tr>
<td></td>
<td>E.g. logging, set insertion, content distribution</td>
<td>with concurrent updates</td>
</tr>
</tbody>
</table>
V. SR LEVEL – SERIALIZABLE TRANSACTIONS

We now introduce additional mechanisms in the CSI model to support serializable transactions. These mechanisms are based on some of the fundamental characteristics of snapshot isolation (SI) based transactions presented in [17]. An anti-dependency [1] between two concurrent transactions \( T_i \) and \( T_j \) is a read-write (rw) dependency, denoted by \( T_i \xrightarrow{rw} T_j \), implying that some item in the read-set of \( T_i \) is modified by \( T_j \). This is the only kind of dependency that can arise in the SI model between two concurrent transactions. Snapshot isolation based transaction execution can lead to non-serializable executions as shown in [5]. It is shown in [17] that a non-serializable execution must always involve a cycle in which there are two consecutive anti-dependency edges of the form \( T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k \). In such situations, there exists a pivot transaction [17] with both incoming and outgoing rw dependencies. In the context of traditional RDBMS, several techniques [6], [7], [20] have been developed utilizing this fact to ensure serializable transaction execution. The pivot prevention approach requires checking whether an item read by a transaction is modified by any concurrent transaction.

We utilize such an approach for ensuring the consistency of the data items and transactions at the SR level. For this the validation phase also involves the conflict resolvers for the items in the read-set of the transaction. The conflict resolver for an item at the SR level performs anti-dependency checks in addition to checking for write-write conflicts. We refer to this type of resolver as SR resolver. Each data item at the SR level is associated with an instance of this type of conflict resolver. In the validation request to a resolver of this type, the transaction indicates read or write mode for the item. The resolver maintains lock in either the read or write mode. For a read-mode validation request, the SR resolver checks if the latest version of the item is visible in the transaction’s snapshot and no one is holding the write mode lock. If this condition holds, it gives a read lock to the request and returns a ‘yes’ vote, otherwise a ‘no’ vote is sent. The read-mode lock can be granted to any number of concurrent read-validation requests, and for this a counter is maintained for the number of currently granted read locks. For a write-mode validation request, the SR resolver checks that the latest version is visible in the transaction’s snapshot and the lock is free.

All concurrent transactions at the SR level are serializable with respect to the data items at the SR level. This property follows from the observation that an SR level transaction can never become a pivot. The basis for this observation is that such a transaction cannot have an outgoing anti-dependency because (1) its read-set can contain only the data items that are at the SR level, and (2) the transaction is committed only if no read-write conflicts with other concurrent transactions are found for any of the items in its read-set. This conflict checking is performed using the SR resolvers corresponding to the read-set items.

SR level transactions may not be serializable with respect to data items which they may update at the weaker consistency levels. For example if two SR level transactions concurrently append log records to two logfiles at the ASYNC level, their records may appear in different order in the logfiles.

For certain types of transactions the read-up/write-down rule can be relaxed. Specifically, when a transaction is creating a new item at a level, it can possibly construct the contents the new item based on the information read from a data item at a lower consistency level. The integrity and consistency of the new item’s contents is an application-specific function of the transaction creating that item. However, creation of new items at the SR level raises the well-known issues related to predicate locks [16] and phantoms [5]. These issues can be addressed by appropriately including some indexing objects in the read/write sets of those transactions.

VI. CSI-CM – COMMUTATIVE LEVEL

We now present an extension to the CSI model to support a consistency level that is below the CSI consistency level. In the CSI model, all updates to an item are total ordered by its resolver. In case of concurrent updates to an object, only one transaction is able to commit. We now present mechanisms to support concurrent updates that are commutative. One can exploit operation commutativity to support greater concurrency in transaction execution by reducing the probability of transaction aborts due to write-write conflicts. The concept of exploiting commutativity of operations in concurrency control has been extensively investigated in the past [34].

For supporting concurrent commutative updates, the basic resolver in the CSI model is extended as described below. The validation request to the conflict resolver for an item contains the operation identifier along with the parameters. The conflict resolver checks if any newer versions of the item, not present in the requesting transaction’s snapshot, have been created by operations that commute with the operation in the validation request. If so, it gives ‘yes’ vote for that transaction. Otherwise it aborts the transaction. In case of a ‘yes’ vote, the resolver keeps track of all commit-pending requests for which a ‘yes’ vote has been given but the commit/abort decision is not yet known.

We use the notion of method license [15] to determine if an operation would commute with all of the operations that are commit-pending at the resolver. In this model it is possible for concurrent transactions modifying the same item with commutative operations to commit. Their update propagation messages contain the operation name and the parameters rather than the updated values. A remote site recomputes the updated value of an item based on this information. The commutative updates of such concurrent transactions may get applied in different orders at different sites. Thus snapshots of different sites can fork even with respect to a single item. Eventually they will converge to the same value when all such concurrent updates have been applied. We illustrate this with an example described below.
Figure 2 shows an example of commutative operations and the resulting fork-join scenario. In this example we have three sites. The figure shows the timeline of the versions for item A at these sites, and it also shows the order in which these versions were committed by the resolver. Initially all sites have the version with timestamp (1,3) as the latest version of item A. This version was created by a transaction at site 3. At time t0, the snapshots at all the three sites are the same. In this example we have two commutative operations a and b, which are executed by two transactions concurrently at sites 1 and 2, respectively. The resolver is able to commit both these transactions. The transaction with timestamp (5,1) at site 1 commits before the transaction with timestamp (10,2) at site 2. The item versions created at the transaction execution sites are shown by a solid circle, whereas a version created by a remote update is shown by an empty circle. At time t1, site 1 has not seen the update of b, site 2 has seen the update of a, and site 3 has seen the update of only b but not a. At this point the view of the versions and value of item A differs at these sites. At time t2 both these updates have been applied at all the three sites but in different orders. Because these two updates commute, the value of item A will be the same at all these sites. At this point the snapshots and values of item A have converged, reflecting the join point.

In the above example, if an operation c that does not commute with a and b is executed on item A by some concurrent transaction, then the resolver will send a 'no' vote for the validation request by such a transaction, thereby aborting it. Later, on re-execution, such a transaction will be able to commit only after the versions created by a and b have become visible in its snapshot and no conflicting operation is being concurrently executed by any other transaction. With a steady stream of concurrent commutative updates it is possible that a transaction with a non-commutative operation may repeatedly abort. To prevent such a case, the conflict resolver may stop granting commit permissions to new commutative validation requests once it sees some number of non-commutative requests failing due to write-write conflicts.

For a CSI-CM level object, the resolver is defined based on the notion of method license [15] to check if an operation commutes with a group of concurrent operations. The resolver maintains some information about the state of the object and the concurrent commit-pending operations. This information is used for checking if a method should be permitted to execute concurrently with the commit-pending methods. For example, consider a Hashtable object which maintains a set of keys. Its operations insert(key, value), delete(key), and isMember(key) all commute with each other if the values of their key parameter are distinct. Therefore, a resolver for such an item needs to maintain only the set of keys for which these operations are currently commit-pending. A typical resolver also groups methods into different commutative groups such that the methods in a group always commute with each other, but methods in different groups do not commute. For example, in case of a Hashtable object, the operation listKeys to enumerate all keys does not commute with the insert and delete operations.

Consider another example where we have an integer object. Any number of add and subtract operations commute with each other but none commutes with the multiply operation when the parameter of multiply is not 1. Any number of multiply operations commute with each other.

Other replication management systems have also exploited commutativity of operations for greater concurrency. The PSI model’s use of commutativity is limited to cset which are based on the notion of commutative replicated data types (CRDT) [24]. For such objects all operations always commute unconditionally, thus requiring no conflict checking. Our model provides a more general framework where the resolver for an object checks for the commutativity of a group of concurrent operations, taking into account their parameters and the object state, which is abstracted in the form of a method license [15]. If all methods of an object commute unconditionally, then we can eliminate conflict-checking for such an object, as in the PSI model. In the RedBlue consistency model [25] one has to perform commutativity analysis for a set of transactions which can potentially span several objects. In contrast, in our model the commutativity analysis is confined to an object as an abstract data type, using the methodology presented in [15]. This simplifies the task of the application developer.

VII. ASYN CH LEVEL – UNSYNCHRONIZED CONCURRENT UPDATES

This is the weakest consistency level in the hierarchy presented in Table 1. For data items at this level, in the CSI protocol no conflict checking is performed, i.e. there is no checking for write-write or read-write conflicts for data items at this level. This level is useful for data items such as logs or sets, where a transaction appends a record to a log, or inserts an item in a set. The order in which these operations are performed does not matter. For example, consider a log where it does not matter if the records appear in different order at different sites, but the only requirement is that eventually all records are appended to the log. The causality property in transaction update propagation is still preserved at this
consistency level. This model can be utilized for updating data (such as web documents) in content distribution networks.

VIII. AN EXAMPLE OF DATA MODEL AND TRANSACTIONS WITH MULTILEVEL CONSISTENCY

We illustrate here an example of modeling a database with data items and transactions at multiple consistency levels. This example relates to an e-commerce application maintaining a set of product and user records, as shown in Figure 2. The data items encapsulated in these records are placed at different consistency levels based on the application requirements. In this example the data items are placed at four levels: SR, CSI, CSI-CM, and ASYNC. This example includes several transactions pertaining to different levels and accessing these items according to the read-up and write-down rules.

A. Data Modeling

A product record contains six fields. The ProductID is an integer number which uniquely identifies the product, and this value is stored in the product record. The other four fields are unique IDs (UID) which are keys for objects stored in the key-value store. The Price field refers to an Integer type object in the storage system. This item belongs to the SR level because we want the transactions updating the price and those reading it for purchase operations to be serializable. The Description field refers to an object which stores a blob containing object description such as images and text. This object is placed at the CSI level because we want all updates to this item to be total ordered and propagated using causal ordering. The next two items are at the CSI-CM level to permit concurrent commutative updates. The Inventory field refers to a PositiveCounter type object. This type of objects support two commutative operations: increment and decrement. This object can have only non-negative integer values. The ProductRating field refers to a Counter type object, which supports two commutative operations: increment and decrement. This object can have any integer value. The last item ProductLog refers to an append-only logger object at the ASYNC level.

A user record comprises of five fields. The UserID is an integer number which uniquely identifies the user in the system. The Account field refers to an Integer object reflecting the current available funds in the user’s account. This is placed at the SR level as we want the update transactions on these items to be serializable. The UserInfo field refers to a blob object containing personal information of the user, such as contact information. The PaymentRecord points to a blob object which stores the invoice details of the user’s purchase transactions. These two objects are placed at the CSI level to ensure that the updates to these items are total ordered. The User Profile field refers to a Set object, which contains information about the user’s interest categories. This object supports three operations – add(key), remove(key), isPresent(key) – and concurrent operations on different keys are commutative. To allow such commutative concurrent updates, this object is placed at the CSI-CM level. The last field Activity Log belongs to ASYNC level and points to a logger object.

A vendor record contains four fields : VendorID which is a string, uniquely representing a vendor in the e-commerce application. The next field AccountList refers to a list of references for accounts objects of Integer type, which are cash balances of the vendor. All account objects are placed at the SR level to ensure that all updates to these items are serializable. The VendorInfo points to a blob object at the CSI level, containing details about the vendor. The last field VendorLog points to a logger object which is placed at the ASYNC level.

B. Transaction modeling

This e-commerce application contains nine transactions which belong to different consistency levels. As shown in Figure 2, two transactions are defined at the SR level. These transactions are serializable with respect to the data items at the SR level. The PurchaseItems transaction involves purchase of a set of items by a user. This transaction reads the current

Fig. 2. An Example of Data and Transaction Hierarchy based on Multilevel Consistency Model
price, computes the payment amount, deducts this amount from the user's account and adds it to one of the vendor accounts. This transaction decrements the inventory for each purchase item and commits only if the specified quantity is available in the inventory. This transaction updates the user's PaymentRecord and ActivityLog objects. The read-set and the write-set items of this transaction are as follows:

Read-set of PurchaseItems transaction:
- Product Price (SR level) objects for the purchase items
- User Account (SR level)
- Vendor Account (SR level)

Write-set of PurchaseItems transaction:
- User Account (SR level)
- Vendor Account (SR level)
- User PaymentRecord (CSI)
- Product Inventory (CSI-CM)
- User ActivityLog (ASYNC)

The UpdatePrice transaction, at the SR level, modifies the prices of a specified set of items. Its write-set consists of the Price objects of the specified products.

The following three transactions are defined at the CSI level. The UpdateDescription transaction updates the description of a product. Its write-set contains the Description object.

The PrepareAccntStatement transaction reads a specified user's PaymentRecord object and creates a statement object which is placed at the CSI level.

The UpdateUserInfo transaction reads and updates a User-Info object of the specified user.

The next set of transactions described below execute at the CSI-CM level. The UpdateInventory transaction increments the inventory of a specified product. Its write-set contains the product Inventory. It updates the Inventory object which is also at the CSI-CM level.

The UpdateProductRating transaction is run when a user up-votes or down-votes the rating of a product. Its write-set contains ProductRating, which is a Counter object.

The BrowseCatalog transaction executes at the CSI-CM level. Its read-set contains a product's Price, Inventory, and Description objects. This is a read-only transaction.

The LogAnalyzer is a read-only transaction at the ASYNC level and it reads various logs at this level for analysis.

IX. TESTBED SYSTEM AND EXPERIMENTAL EVALUATIONS

We developed a testbed system and conducted evaluation experiments using the e-commerce application described in Section VIII. For this purpose we developed a benchmark workload using this application. The goal of these experiments was to compare the performance of this workload executing under the proposed multi-level model with its execution under a single consistency model such as SR or CSI. The performance measure is the peak throughput for which the cumulative commit rate for the workload mix is around 95%. Moreover, we require that at the peak throughput the commit rate for each individual transaction type is at least close to 90%. To assess the scale-out capabilities we conducted these experiments for different number of sites.

A. Testbed System

The testbed for the proposed multi-level consistency model was developed by extending the system that we had built for supporting the PCSI model for partial data replication [29]. This system supports key-value based multi-version database, which can be maintained either in memory or in HBase as persistent storage. In the PCSI model a key-value database is shard into disjoint partitions. A partition can be replicated at any number of the sites. A transaction can read or write data in any of the partitions. The PCSI system used a simple conflict resolver for each object to detect write-write conflicts.

To build our testbed environment, we modified the PCSI system to associate a specific type of resolver object with each data item. The resolver type of an item depends on the consistency level of that data item, and one can install any required type of resolver for a particular data item. In the testbed we provide two system-defined conflict resolvers. One is for the SR level items to perform both read-write and write-write conflict checking, and other for the CSI level items to perform only write-write conflict checking. For an item at the CSI-CM level one has to define a specific type of resolver based on the commutative properties of the object methods.

For implementing the benchmark application we developed resolvers for three types of objects: Set, Counter, and Positive Counter. The Set type objects are used for implementing User-Profile objects. A Counter type object maintains an integer value and provides two methods: increment and decrement. It is used for implementing ProductRating objects. The type PositiveCounter is similar to the Counter type except that it cannot have negative integer values. This type is used for implementing ProductInventory objects.

B. Benchmark Workload

We implemented the example e-commerce application from Section VIII on our testbed system. The database is sharded into partitions, and in our experiments the number of partitions was set equal to the number of sites in the system. Each partition contained 2000 product items, 20000 users, and 500 vendor account objects. This database is maintained in memory. We conducted experiments with three system configurations containing 4, 8, and 12 sites, respectively. The degree of replication for the partitions was set to 4. It should be noted that a larger system configuration reflects larger database size.

We defined two benchmark workloads, called BW1 and BW2, which consist of a mix of transactions types in this e-commerce application. These benchmarks emulate a Web shop in a manner similar to the TPC-W benchmark. Table II lists the transactions and their fraction in the benchmark workload. The benchmark workload mix BW1 consists of eight transactions. This workload reflects a shopping mix with a large fraction of read-only browsing transactions. The benchmark workload mix BW2 reflects purchasing activities with a large fraction of purchase related transactions. All transactions in this benchmark workload involve updating one or more items. Table II shows the number of items in the read-set (R) and the write-set
(W) of a transaction type. For the PurchaseItems transaction the number of purchase items is set to three, and these products are randomly selected. The UpdatePrice, UpdateDescription, and UpdateInventory transactions update the corresponding items of five randomly selected products.

In these benchmarks we modeled popular products which are more frequently selected by the transactions. Twenty percent of the products were considered as hot, i.e. more popular than the others, and 20% of the transactions involved accessing only these hot-spot products. We refer to them as hot-spot transactions. Similarly we also modeled active users who initiate transactions more frequently than others. Twenty percent of the users were considered as active, and 20% of the user-centric transactions were initiated by active users. This emulates contention on data items in real environments. All items accessed by a transaction were selected from partitions present at its execution site.

In our experiments we have considered different system configurations corresponding to three different contention levels: CM, CM, and CM. We have separately evaluated the effect of different fractions of hot-spot transactions in the workload mix. For a system with 12 sites we measured the peak throughput of BW1, varying the hot-spot transaction fraction from 10% to 50%, reflecting different contention levels. Figure 5 shows the results of this evaluation for the three models under different contention levels. The multilevel model consistently performs better than SR and CSI at all contention levels.

C. Experimental Evaluations

We conducted our experimental evaluations on a computing cluster. Thus these evaluations are indicative of performance in a datacenter environment. In this cluster, each node had 8 CPU cores with 2.8 GHz Intel X5560 Nehalem EP processors, and 22 GB main memory. A node in the cluster served as a database site in our experiments.

We evaluated the performance of this benchmark workload under three system configurations corresponding to three consistency models: SR, CSI, and Multilevel. For the SR configuration, SR resolvers were installed for all data items and all transactions executed at the SR level. In the second configuration, CSI resolvers were installed for all data items and all transactions executed at the CSI level. The third configuration was used for multi-level execution of transactions according to the hierarchy shown in Figure 2. In this configuration, different types of resolver objects were installed for the data items according to their consistency levels.

We executed the benchmark workload on system configurations with 4, 8 and 12 sites. We measured the peak throughput based on the requirements for the commit rates noted above. Figures 3 and 4 show the peak throughput of the benchmark BW1 and BW2, respectively, for SR, CSI, and Multilevel models. These figures show the peak throughput for number of sites 4, 8, and 12. Based on the evaluations for these three system sizes we find that the Multilevel model performs better than SR by factors of 2.11 for BW1 and 2.86 for BW2. In comparison to the CSI model, the performance gain factors for the Multilevel model are 1.64 for BW1 and 2.6 for BW2.

Table III shows the commit rate and throughput of transactions in benchmark workload BW1 on a system with 8 sites for the three models. We also show here the cumulative commit rates for these models. With the Multilevel model, the CSI-CM level transactions tend to have high commit rate due to commutative operations.

We separately evaluated the effect of different fractions of hot-spot transactions in the workload mix. For a system with 12 sites we measured the peak throughput of BW1, varying the hot-spot transaction fraction from 10% to 50%, reflecting different contention levels. Figure 5 shows the results of this evaluation for the three models under different contention levels. The multilevel model consistently performs better than SR and CSI at all contention levels.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Item-set</th>
<th>Txn Fraction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Purchased</td>
<td>R W</td>
<td>15 25</td>
</tr>
<tr>
<td>Items</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdatePrice</td>
<td>5 5 5 5</td>
<td></td>
</tr>
<tr>
<td>UpdateDesc</td>
<td>5 5 5 5</td>
<td></td>
</tr>
<tr>
<td>PrepareAcnt</td>
<td>1 1 5 5</td>
<td></td>
</tr>
<tr>
<td>Stmtnt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateUser</td>
<td>1 1 10 10</td>
<td></td>
</tr>
<tr>
<td>Info</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateInven</td>
<td>5 5 5 15</td>
<td></td>
</tr>
<tr>
<td>tory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateProdc</td>
<td>1 1 20 35</td>
<td></td>
</tr>
<tr>
<td>tateRating</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BrowseCatalog</td>
<td>3 0 35 0</td>
<td></td>
</tr>
<tr>
<td>TABLE II</td>
<td>TRANSACTIONS IN THE BENCHMARK WORKLOAD</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transaction Commit Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction</td>
</tr>
<tr>
<td>Types</td>
</tr>
<tr>
<td>BW1</td>
</tr>
<tr>
<td>PurchaseItems</td>
</tr>
<tr>
<td>UpdatePrice</td>
</tr>
<tr>
<td>UpdateDescription</td>
</tr>
<tr>
<td>PrepareAcntStmtnt</td>
</tr>
<tr>
<td>UpdateUserInfo</td>
</tr>
<tr>
<td>UpdateInventario</td>
</tr>
<tr>
<td>UpdateProductRating</td>
</tr>
<tr>
<td>BrowseCatalog</td>
</tr>
<tr>
<td>Cumulative rate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transaction Throughput (txns/second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cumulative throughput</td>
</tr>
</tbody>
</table>

X. CONCLUSION

We have presented here a transaction model for replicated data with different consistency guarantees. This model simultaneously supports transactions with different consistency levels. The Causal Snapshot Isolation (CSI) model serves as the building-block for this transaction management framework. This model supports serializable transactions for strong consistency, and weaker consistency models which include CSI, CSI with commutative updates, and CSI with asynchronous updates. Data and transactions are organized in a hierarchy which is based on these consistency models. This model ensures the consistency guarantees of data at each level in this hierarchy by constraining the information flow across different levels. We developed a testbed for replicated data management supporting this multi-level model. We show here the utility of the proposed model using an e-commerce application implemented on our testbed. Our evaluations show that the multi-level model consistently performs better than the SR and CSI models across different contention levels and system sizes while exhibiting scale-out capability.
Acknowledgments: This work was supported by NSF Award 131933. Computing resources for this work were provided by the Minnesota Supercomputing Institute.

REFERENCES