A Transaction Model for Management of Replicated Data with Multiple Consistency Levels

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

***Abstract*—We present 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 updates. This model is useful in managing large-scale replicated data with different consistency guarantees to make suitable trade-offs between data consistency and performance. 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

In large-scale systems, supporting transactions with strong consistency for serializability imposes scalability and availability limitations due to distributed coordination. Replication management protocols with weaker consistency models provide lower latencies for transactions and high availability, but guarantee only eventual consistency [6] or causal consistency [19], [15]. 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 [6], [5], [19], [15] have been developed supporting weaker consistency models for replicated data management in large-scale systems.

Our work is motivated by the observation that rather than providing a single consistency model for transactions in replicated data management systems, 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 [21], [14], [12]. The RedBlue consistency model [14] uses operation commutativity to relax ordering guarantees. The Salt [21] model provides a framework for simultaneously supporting ACID and BASE transactions in a system and provides rules for their isolation. The framework presented in [12] places data items in different consistency categories and adaptively changes the category of data items at runtime.

In this paper we present a transaction management model for managing replicated data with different consistency levels to make suitable trade-offs between data consistency and performance. This transaction management model is based on snapshot isolation (SI) [2] with weaker semantics that guarantee causal consistency of snapshots as proposed in [19], [17]. We use the CSI [17] model as a building-block in our work. 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.

For experimental evaluations we developed a testbed system, implementing it over the transaction management system we had earlier developed for the CSI model for partially replicated data [16]. 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 multilevel consistency model, and compared it with the performance under a single consistency model. Our evaluations show significant benefits of the proposed approach.

In the next section we present the related work. Section III presents an overview of the CSI model. Section IV presents the proposed multi-level consistency model. Section V describes how the CSI model is extended to support serializable transactions. The mechanisms for supporting concurrent commutative updates are presented in Section VI, and for eventual consistency in Section VII. Section VIII presents an example of an e-commerce application based on the multi-

978-1-4799-9926-2/15/$31.00 ©2015 IEEE

level consistency model. Section IX presents the results of our experimental evaluations.

II. Related Work

The issues with scalability in data replication with strong consistency requirements are discussed in [10]. 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) [2] and weaker consistency models such as eventual and causal consistency.

Snapshot isolation (SI) model [2] is based on multi-version data management, utilizing optimistic concurrency control, in which a transaction reads only committed version of data. After execution, the transaction goes through a validation phase to check for *write-write* conflicts with concurrently committed transactions. The problem of transaction serializability in snapshot-isolation model has been extensively studied [9], [3], [4], [18], [11]. The work in [9] characterizes the conditions necessary for non-serializable transactions in the SI model.

The SI model poses scalability limitations in wide-area environments because of the serial execution of validation operations. Parallel Snapshot Isolation (PSI) [19] 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 [17] model improves upon the PSI model to reduce false causal dependencies.

Several projects have pursued the goal of simultaneously supporting different consistency models. The system presented in [1] provides mechanisms for supporting causal consistency in systems with eventual consistency. The RedBlue consistency model [14] requires analysis of the transaction operations to split a transaction into two parts, one containing commutative shadow operations. The Salt [21] model requires rewriting an ACID transaction as a BASE transaction consisting of a series of *alkaline* nested transactions. Both RedBlue and Salt models require analysis and rewriting of application level transactions. The approach in [12] places data into different consistency categories and adaptively changes 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 proposed multi-level model is centered around the CSI model [17], which is based on a weaker form of snapshot isolation model proposed in PSI [19]. The system consists of a set of distributed database sites. Each site is identified by a unique $siteId$, S_i for $i \in (1..n)$. Each site has a local database that supports multi-version data management. Transactions can execute at any site. A transaction first commits locally and then its updates are propagated to other sites asynchronously. Transactions committing at a site are ordered using a local sequence number. 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 CSI model provides the following guarantees for transaction execution:

- Transaction Ordering: A partial order relationship ($\prec$) is defined over the set of transactions, as described below:
  - *causal ordering*: If transaction T_j reads any of the updates made by transaction T_i, then T_i causally precedes 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.
- Causally Consistent Snapshot: A transaction observes a *consistent snapshot* which satisfies 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 visible.

The details of the transaction execution protocol can be found in [17]. A brief overview is presented below. Each site maintains a vector clock. When a transaction begins, it is assigned the current vector clock value as its snapshot time. It reads the latest version of the items in its read-set based on this snapshot time. All writes are performed on local copies of data items. After the transaction has completed execution, it goes through a validation phase. For each item in its write-set, it checks for *write-write* conflicts with other concurrent transactions. For each data item, there is a designated *conflict resolver* for checking *write-write* conflicts. The transaction performs validation as a *two-phase* protocol with the conflict resolvers for the items in its write-set. In the prepare phase, each resolver checks if the latest version of the item is visible in transaction'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. The transaction commits if no conflict is detected. It is assigned a commit sequence number $seqno$ from a monotonically increasing local counter at its execution site. The commit timestamp for a transaction is a pair $<$*siteId, seqno*$>$. It now applies updates to the local database with this commit timestamp as version numbers. The local site's vector clock is advanced appropriately and a commit/abort message is sent to all the conflict resolvers.

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 highest level is the *SR level*, which guarantees serializable transactions. 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 next lower level weakens the CSI model to allow concurrent updates to an item if they are commutative. 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. All transactions execute with the basic protocol of the CSI model, but with different conflict resolution policies. Therefore, all transactions exhibit the isolation properties of the SI model [2]. A transaction always reads committed data, and the updates of a transaction become visible only when it commits. The atomicity property of causal snapshots guarantees that either all or none of the updates of a transaction are visible. 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.

V. SR Level – Serializable Transactions

Snapshot isolation based transaction execution can lead to non-serializable executions [2]. An *anti-dependency* between two concurrent transactions T_i and T_j is a *read-write (rw) dependency*, denoted by $T_i \overset{rw}{\rightarrow} T_j$, implying that some item in the read-set of T_i is modified by T_j. It is shown in [9] that a non-serializable execution must always involve a cycle in which there are two consecutive *anti-dependency* edges of the form $T_i \overset{rw}{\rightarrow} T_j \overset{rw}{\rightarrow} T_k$. In such situations, there exists a *pivot* transaction [9] with both incoming and outgoing rw dependencies. In the context of traditional RDBMS, several techniques [4], [11] have been developed utilizing this fact to ensure serializable transactions.

We now introduce additional mechanisms in the CSI model to support serializable transactions. The pivot prevention approach for ensuring serializability requires checking for *read-write* conflicts among concurrent transactions. We adopt this

TABLE I
Data Consistency Levels and Transaction Protocols

Level	Consistency Properties	Transaction Model
SR	Strong consistency - Globally serializable transactions; ACID properties for transactions	SI-based serializable transactions
CSI	Causal ordering of transaction updates; Per-item update ordering; Fork-join model of snapshots for concurrent updates on different items	Causal Snapshot Isolation
CSI-CM	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	Causal Snapshot Isolation with commutative updates
ASYNC	Causal ordering of transaction updates; E.g. logging, set insertion, content distribution	Causal Snapshot Isolation with concurrent updates

approach for ensuring the consistency properties of 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.

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.

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 of the new item based on the information read from a data item at a lower consistency level. Ensuring the consistency of its contents is an application-specific function. However, creation of new items at the SR level raises the well-known issues related to predicate locks [8] and phantoms [2].

VI. CSI-CM – Commutative Level

One can exploit operation commutativity to support greater concurrency by reducing the probability of transaction aborts due to *write-write* conflicts. Fundamental concepts in exploiting commutativity of operations for concurrency control are presented in [20]. We now present mechanisms to support concurrent updates that are commutative.

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 that all 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* [7] to determine if an operation would commute with all of the operations that are commit-pending at the resolver.

In CSI-CM 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.

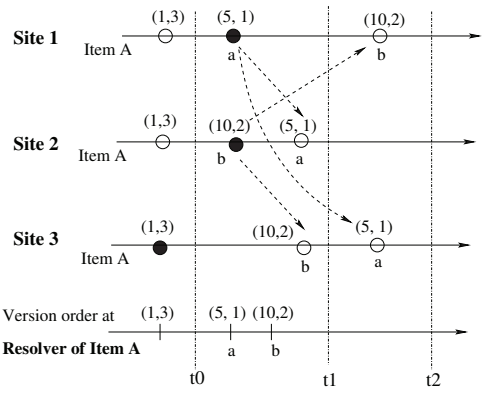

Fig. 1. Fork-Join of Snapshots in the CSI-CM Model

Figure 1 shows an example of fork-join scenario in commutative operations. The figure shows the timeline of the versions for item A at three sites and the commit order for these versions, given by the resolver. Initially, for item A all sites have the latest version with timestamp *(1,3)*, which was created by a transaction at site 3. At time $t0$, the snapshots at all the three sites are the same. Here we have two commutative operations a and b, which are executed by concurrent transactions at sites 1 and 2, respectively. The resolver commits the transaction with timestamp *(5,1)* at site 1 before committing the transaction with timestamp *(10,2)* at site 2. The item versions created at their execution sites are shown by solid circles and the versions created by remote updates are shown by empty circles. 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 values of item A differ at these sites, reflecting a fork. 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, 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* [7] 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. 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.

The PSI model's use of commutativity is limited to *cset* objects, which are based on commutative replicated data types (CRDT) [13]. 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* [7]. 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 [14] 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 [7]. This simplifies the task of the developer.

VII. Async Level – Asynchronous Updates

This is the weakest consistency level in the hierarchy presented in Table 1. For data items at this level, no conflict checking is performed. 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 Multilevel Consistency Model

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 products 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. Here the data items are placed at four levels: SR, CSI, CSI-CM, and ASYNC. Transactions access 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. 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* 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 product description. This object is placed at the CSI level because we want all updates to this item to be total ordered. The next two items are at the CSI-CM level to permit 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*. 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. 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. 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. The *AccountList* field refers to a list of UIDs for vendor account balance objects of *Integer* type. All account objects are placed at the SR level. 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 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 of the *PurchaseItems* transaction contains the following items: products' *Price* objects (SR level) for the purchase items; user *Account* (SR level); and vendor *Account* (SR level). The write-set of this transaction contains the following items: user *Account* (SR level); vendor *Account* (SR level); user *PaymentRecord* (CSI); product *Inventory* (CSI-CM); and user *ActivityLog* (ASYNC). The *UpdatePrice* transaction, which executes 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 the *UserInfo* object of the specified user.

The following three transactions execute at the CSI-CM level. The *UpdateInventory* transaction increments the inventory of a specified product. It updates the product's *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 above. 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 measured the peak throughput at the load for which the commit rate for each individual transaction type was at least 90%. To assess the scale-out capabilities we conducted these experiments for different number of sites.

A. Testbed System

The testbed for the proposed model was developed by extending the system that we had built for supporting the PCSI model [16]. This system supports key-value based multi-version database, which can be maintained either in memory or in HBase. The database is sharded into disjoint partitions. A partition can be replicated at any number of the sites.

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 *UserProfile* 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.

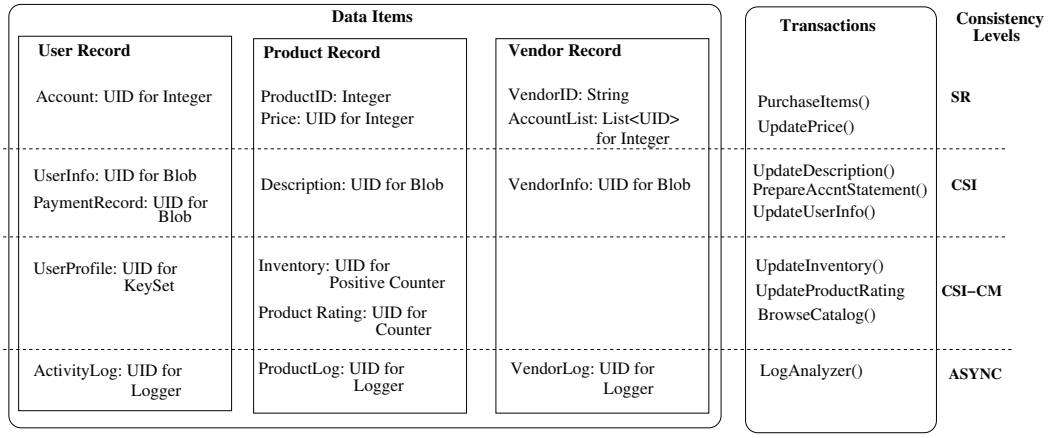

Fig. 2. An Example of Data and Transaction Hierarchy based on Multilevel Consistency Model

B. Benchmark Workload

We implemented the above e-commerce application on our testbed. 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 was set to 4. It should be noted that a larger system configuration reflects a larger database size.

We defined two benchmark workloads, called BW1 and BW2, which consist of a mix of transaction 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.

Transaction	Item-set		Txn Fraction (%)	
Types	R	W	BW1	BW2
PurchaseItems	5	7	15	25
UpdatePrice	5	5	5	5
UpdateDescription	5	5	5	5
PrepareAccntStmnt	1	1	5	5
UpdateUserInfo	1	1	10	10
UpdateInventory	5	5	5	15
UpdateProductRating	1	1	20	35
BrowseCatalog	3	0	35	0

TABLE II
TRANSACTIONS IN THE BENCHMARK WORKLOAD

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 served as a 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 performance of benchmark BW1 on a system with 8 sites for the three models. In these experiments, for each model the load level was set such that the commit rate for all transaction types was at least 90%. The table shows the transaction throughput (transactions/second) and average latency (msec), and the commit rates. The throughput with the Multilevel model is more than the SR model by a factor of 2.37, and by 1.67 compared to CSI. The average latency in the SR model is about 73% more than those with the Multilevel and CSI models. For each model, the commit rates for individual transaction types depend on contention among transactions and the imposed load, therefore, the commit rates cannot not be compared across models.

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.

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. The proposed multi-level model supports serializable transactions for strong consistency, and weaker consistency

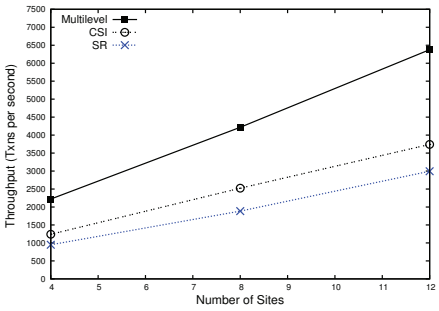

Fig. 3. Throughput of Benchmark Workload BW1 with Multilevel, CSI, and SR Models

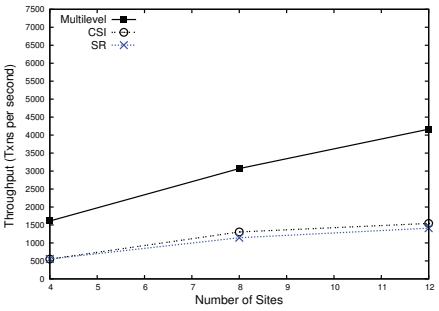

Fig. 4. Throughput of Benchmark Workload BW2 with Multilevel, CSI, and SR Models

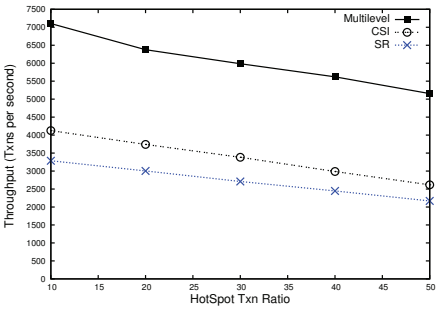

Fig. 5. Throughput of Benchmark Workload BW1 under different fractions of hot-spot transactions with Multilevel, CSI, and SR Models (Sites=12)

Transaction Commit Rate			
Transaction Types	SR (%)	CSI (%)	Multilevel (%)
PurchaseItems	91.16	90.4	90.57
UpdatePrice	95.8	96.86	94,19
UpdateDescription	96.87	97.22	95.69
PrepareAccntStmnt	99.95	99.95	99.89
UpdateUserInfo	100	99.96	100
UpdateInventory	95.32	94.25	100
UpdateProductRating	99.67	99.57	100
BrowseCatalog	99.25	100	100
Cumulative rate	97.74	97.88	98.07
Transaction Throughput (txns/second)			
Cumulative throughput	1779	2524	4217
Avg. Latency (msec)	52 msec	30 msec	31 msec

TABLE III
TRANSACTION COMMIT RATES IN BENCHMARK WORKLOAD BW1 ON 8 SITE SYSTEM AND REPLICATION DEGREE OF 4

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 and the Minnesota Supercomputing Institute.

REFERENCES

[1] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proc. of ACM*, SIGMOD '13, pages 761–772, 2013.

[2] 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*, pages 1–10. ACM, 1995.

[3] M. Bornea, O. Hodson, S. Elnikety, and A. Fekete. One-copy serializability with snapshot isolation under the hood. In *IEEE ICDE'11*, pages 625 –636, april 2011.

[4] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34:20:1–20:42, December 2009.

[5] 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.*, 1:1277–1288, August 2008.

[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.*, 41:205–220, October 2007.

[7] J. Eberhard and A. Tripathi. Semantics Based Object Caching in Distributed Systems. *IEEE Trans. on Parallel and Distributed Systems*, December 2010.

[8] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, Nov. 1976.

[9] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30:492–528, June 2005.

[10] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of ACM SIGMOD'96*, pages 173–182, 1996.

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

[12] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.*, pages 253–264, Aug. 2009.

[13] M. Letia, N. Preguiça, and M. Shapiro. Consistency without concurrency control in large, dynamic systems. *SIGOPS Oper. Syst. Rev.*, 44(2):29–34, Apr. 2010.

[14] 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*, pages 265–278, 2012.

[15] 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*, pages 401–416, 2011.

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

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

[18] S. Revilak, P. O'Neil, and E. O'Neil. Precisely Serializable Snapshot Isolation (PSSI). In *ICDE'11*, pages 482–493.

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

[20] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.*, 37:1488–1505, December 1988.

[21] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining acid and base in a distributed database. In *Proc. of USENIX OSDI'14*, pages 495–509, 2014.