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

Anand Tripathi
Department of Computer Science & Engineering
University of Minnesota, Minneapolis, Minnesota, 55455 USA
Email: tripathi@cs.umn.edu

ABSTRACT

We present here a transaction model for managing replicated data with different consistency guarantees to make suitable trade-offs between data availability, scalability, and consistency for different data items. Data is organized in a hierarchy which is based on consistency levels and the associated transaction management protocols. The model presented here is based on imposing certain constraints on information flow across data layers in this hierarchy to ensure the desired consistency guarantees for data items at different levels. The transaction model proposed here simultaneously supports transactions with different consistency levels, which include ACID transactions for strong consistency, and weaker consistency models such as snapshot isolation (SI), causal snapshot isolation (CSI), CSI with commutative updates, and CSI with unsynchronized concurrent updates. The building block for this transaction model is the snapshot isolation model.

1. INTRODUCTION

Management of replicated data in distributed systems poses fundamental trade-offs between data consistency, scalability, and availability [10]. In such systems supporting transactions with strong consistency for serializability requires distributed coordination and synchronous update propagation to other sites before committing the transaction. This incurs high latencies for transactions and it cannot tolerate network partitioning. Replication management protocols with weaker consistency models can utilize asynchronous update propagation. Such models provide lower latencies in transaction execution and high availability, but guarantee only eventual consistency [7] or causal consistency [18, 12]. 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 [18, 12] have been developed recently with focus on supporting causal consistency.

Many data management systems for cloud datacenters distributed across wide-area have been proposed [7, 6, 2, 12, 18]. Dynamo [7] uses asynchronous replication with eventual consistency but does not support transactions. PNUTS [6] does not provide transactions, but provides a stronger consistency level than eventually consistency, called as eventual timeline consistency. It also supports a model of data management with different consistency guarantees for data. Megastore [2] provides transactions over a group of entities using synchronous replication. COPS [12] provides causal consistency, but does not provide transaction functionality, except for snapshot-based read-only transactions. Eiger [13] provides both read-only and update transactions with causal consistency but requires maintaining causal dependencies on per object level. PSI [18] provides transaction functionality for geo-replicated data using asynchronous replication, guaranteeing causal consistency. The CSI [14] model improves upon the PSI model to eliminate false causal dependencies. Rather than providing a single model for transactions in replicated data management system, it is desirable to support a model in which transactions with different levels of consistency guarantees can be supported simultaneously. We present here a transaction management model for managing replicated data with different consistency models to make suitable trade-offs between data availability, scalability, and consistency for different data items. The building block for this transaction model is the snapshot isolation model. The transaction model proposed here simultaneously supports transactions with different consistency levels, which include ACID transactions for strong consistency, and weaker consistency models such as snapshot isolation (SI), causal snapshot isolation (CSI), CSI with commutative updates, and CSI with concurrent updates. The proposed model is based on asynchronous update propagation. 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. The model presented here is based on imposing certain constraints on information flow across data layers in this hierarchy to ensure the desired consistency guarantees for data items at different levels.

In the proposed transaction management model, data items are replicated at all site. A transaction can be executed at any of the sites and it can access any of the items but with certain constraints, which we identify in this paper. The underlying model for transaction management is based on snapshot isolation (SI) [3] with weaker semantics that guarantee causal consistency of snapshots as proposed in [18, 14]. The updates of a transaction are propagated asynchronously...
chronously to other sites. Using the CSI model as a building-block, we introduce additional mechanisms to provide two stronger guarantees, one to support SI-based transactions and other for serializable transactions. On the other hand, we also introduce new mechanisms to provide two models with weaker consistency, one based operation commutativity, and the other supporting unordered concurrent updates.

The rest of the paper is organized as follows. In the next section we give an overview of the CSI model, which serves as the building-block for the transaction model for supporting multiple consistency levels. In Section 2 we present the hierarchy of consistency levels and the corresponding transaction protocol. Here we outline the basic principles for ensuring consistency guarantees for data at a given level. Section 3 describes how the CSI model is extended to support transactions with two stronger levels of consistency guarantees: serializable transactions and snapshot isolation based transactions. Section 4 describes the mechanisms we introduce in the CSI model to support two weaker levels of consistency guarantees. Conclusions are presented in the last section.

2. CAUSAL SNAPSHOT ISOLATION

The building-block of our transaction management framework is the Causally-coordinated Snapshot Isolation (CSI) [14] model. It based on a weaker form of snapshot isolation model as proposed in Parallel Snapshot Isolation (PSI) [18]. Transactions executing at a single site are ordered using a local sequence number. Transaction updates are propagated asynchronously to remote sites. This model does not enforce a global ordering of transactions, however, it ensures causal ordering while applying transactions at remote sites. The CSI model differs from PSI with respect to the approach for ensuring causal consistency by reducing false dependencies.

2.1 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. Data items are replicated at all the sites. For each data item, there is a designated conflict resolver site which is responsible for checking for write-write conflicts for that item. Transactions can execute at any site. Read-only transactions can be executed locally without needing any coordination with remote sites. Update transactions need to coordinate with conflict resolver sites for write-write conflict checking for the items in their write-sets.

2.2 CSI Consistency Guarantees

In the CSI model, transactions are ordered according to a causal ordering. At a remote site, transactions are applied according to their causal ordering. The snapshot observed by a transaction may not always reflect the latest versions of the accessed items, but it is guaranteed to be consistent as described below. The CSI model provides the following guarantees for transaction execution:

- Snapshot Isolation: As in the case of traditional snapshot isolation, the CSI model guarantees that when two or more concurrent transactions update a common data item, only one of them is allowed to commit.
- Transaction Ordering: The CSI model provides ordering guarantees for transactions based on two properties: causal ordering, and per-item global update 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 $T_i$ creates a newer version for any of the items modified by $T_j$, 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: The CSI model guarantees that a transaction observes a consistent snapshot which has the following two properties of atomicity and causality:
  - Atomicity: In a consistent snapshot either all or none of the updates of a transaction are visible.
  - Causality: If a snapshot contains updates of transaction $T_i$, then updates of all transactions causally preceding $T_i$ are also contained in it.

2.3 CSI Protocol

As noted earlier, the total ordering on transactions is not enforced. This eliminates the need of a global sequencer. Instead, 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 $<\text{siteId}, \text{seqno}>$. Similarly, a data item version is identified by a pair $<\text{siteId}, \text{seqno}>$. The local sequence number is assigned only if the transaction is guaranteed to commit, i.e. only if there is no write-write conflict. Thus, there are no gaps in the sequence numbers of the committed transactions. A transaction first commits locally and then its updates are propagated to other sites asynchronously. A remote site, upon receiving a remote transaction’s updates, applies the updates provided that it has also applied updates of all the causally preceding transactions. The transactions from a particular site are always applied in the order of their sequence numbers. All the updates of a transaction are applied to the local database as an atomic operation, which also includes updating a local vector clock.

Each site maintains a vector clock, which we denote by $VC$, 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 $S_j$.

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. Note that, because there is no total ordering, the versions can not be compared based on the sequence numbers. For example, if there are two versions of a particular data item, one created by a transaction at site $s_1$ with sequence number 5 and other created by a transaction at site $s_2$ with sequence number 7, we can not compare the sequence numbers to determine version ordering because there is no total ordering among transactions at sites $s_1$ and $s_2$. Instead the versions are ordered based on the order in which the conflict resolver validates the corresponding transactions. The write-write conflict detection ensures that writes on a data item are serialized. For each data item $x$, each site maintains a version log which basically indicates the order of the versions. When $t$ performs a read operation on $x$, we check for every version $<j,n>$, starting from the version that is applied most recently, if the version is visible in the transaction’s snapshot, i.e. if $VT^t_i[j] \geq n$. We then select the latest version that is visible in $t$’s snapshot. All writes by $t$ are kept in a local buffer until the commit time.

**Commit protocol:** If $t$ has modified one or more items, then it performs write-write conflict checking using a two-phase commit (2PC) protocol with the conflict resolver sites responsible for those items. In the prepare message to each site, $t$ sends $VT^t_i$ and the list of items it has modified for which that site is the conflict resolver. Each site checks, if the latest versions of those items are visible in $t$’s snapshot and that none of the item is locked. The locking is performed to avoid conflicts with any concurrent conflict checking operations by other transactions. If this check fails, then the resolver sends a ‘no’ vote. Otherwise, it locks the corresponding items and sends a ‘yes’ vote. If $t$ receives ‘yes’ votes from all conflict resolvers, $t$ is assigned a monotonically increasing local sequence number by $t$’s local site, obtained from localSequencer counter. First, $t$ commits locally, applying the updates to the local database. The local site’s vector clock is advanced appropriately. It now sends a commit message, containing the sequence number, to all the conflict resolvers. Otherwise, in case of any ‘no’ vote, $t$ is aborted and an abort message is sent to all the conflict resolvers. Upon receiving a commit or abort message, a conflict resolver releases the locks, and in case of commit it records the new version number as a 2-tuple: $(<siteId, seqno>)$. After performing these operations, 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. PSI model uses the transaction’s snapshot vector timestamp $VT_t$ to indicate causal dependencies. Thus, $t$’s updates are applied at a remote site only when that site’s vector clock is advanced up to $VT^t_i$, so that all the events that were visible when $t$ started its execution are visible at the remote site when $t$ is applied there. However, this can induce false dependencies, as not all the updates that were visible at $t$’s execution site when $t$ started are necessarily seen by $t$. In contrast, the CSI model uses effective causal snapshot, which indicates, for each site, the latest event from that site which is ‘seen’ by the transaction based on the items it read or modified. In other words, CSI captures causal dependencies with respect to a transaction rather than a site. The effective causal snapshot for a transaction $t$, executed at a site $S_i$ is defined as a vector timestamp denoted by $VT^t_e$, and it is determined as follows. $VT^t_e[i]$ is set equal to $n-1$ where $n$ is $t$’s sequence number. This indicates that $t$ can be applied only when the transaction immediately preceding $t$ at site $S_i$ has been applied. The other elements of $VT^t_e$, i.e. those corresponding to the remote sites, are determined as follows:

$$\forall j; j \neq i : VT^t_e[j] = \max\{\text{seqno} \mid \exists x \text{ s.t.}(x \in \text{read-set}(p) \vee x \in \text{prevwrites}(p)) \wedge (\text{version}(x) =< j, \text{seqno}>)$$

Here, prevwrites($p$) is the set of latest versions visible at site $S_i$ for the items that $t$ modified. The inclusion of this information ensures that for a particular data item $x$ modified by $t$ a remote site has seen all the preceding versions of $x$ before applying the update by $t$.

### 3. MULTI-LEVEL CONSISTENCY MODEL

In Table 1 we present a hierarchy of data consistency levels and the associated transaction management protocol that can be supported in the proposed model. As noted above, the building-block for implementing these protocols is the CSI model. We outline in this table the consistency properties of the transactions at different levels of this hierarchy, and then present a simple rule that would ensure the consistency guarantees for each level.

The replicated data items are organized along this hierarchy. A data item can belong to only one level. Level 1 corresponds to the highest level consistency and all transactions updating data at this level are guaranteed to be serializable. This level ensures ACID properties for transactions. The next weaker level of consistency corresponds to snapshot isolation (SI) based transactions as defined in [3]. The transactions at this level are globally serialized. The next level below this is the CSI model which provides the consistency properties described in [18]. The transactions are causally 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 [18] for the snapshots at different sites. In the CSI model all updates to an item are serialized. The next lower level in this hierarchy allows concurrent updates to an item if they are commutative. Such updates may get applied to different sites in different orders, thus resulting in the fork-join model even with respect to updates to an item. The lowest level in the hierarchy allows uncoordinated updates to an item. This model is suitable for maintaining logs or inserting some data in a set. All updates are still applied at all sites according to their causal ordering.

We propose the following rules to ensure the consistency properties of the data items in this hierarchy. These rules constrain 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 a lower level number in this hierarchy. A lower level number implies stronger consistency guarantees.

- **Write-Down** A transaction can update items that at its own level or at a higher level number. A higher level number implies lower consistency guarantees.

We refer to these rules at simply read-up/write-down. These rules basically prevent a transaction from using information from a weaker consistency level data to update a stronger consistency level data item. A transaction can update data
items that are at its own level or at a weaker consistency level. For example a transaction at level 1 can read and modify data at level 1, and it will be guaranteed to be serializable with respect to data items at level 1. However, the same transaction may also update certain data items that belong to weaker consistency levels, and this transaction may not be serializable with respect to other transactions at those levels. In this example, a serializable transaction at level 1 may insert some log record in a log at level 5, or it may update some items at level 3 using the CSI protocol. The consistency properties of data at level 1 is guaranteed as no information flows from weaker consistency data items to strong consistency data items.

With this model transactions at different levels can execute concurrently. In the next sections we present how the proposed model can be implemented by building upon the CSI model. We first outline the mechanisms for supporting serializable transaction and snapshot isolation based transactions.

4. CONSISTENCY LEVELS ABOVE CSI

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 based transactions presented in [9]. 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 [3, 9]. Fekete et al. [9] have shown that a non-serializable execution must always involves 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 [9] with both incoming and outgoing rw dependencies. In the context of traditional RDBMS, several techniques [4, 5, 17, 11] have been developed utilizing this fact to ensure serializable transaction execution. The essence is to either detect or prevent the occurrence of a pivot transaction or detect any cycle among transactions in the commit protocol phase.

Based on these concepts we investigated the pivot prevention approach for implementing serializable SI-based transactions on a key-value based system in a datacenter environment [16, 15]. We utilized a dedicated validation service for detecting anti-dependencies for serializability and write-write conflicts for the basic snapshot isolation model. The pivot prevention approach requires checking whether an item read by a transaction is modified by any concurrent transaction. Our experiments showed the scalability of such an approach of using a dedicated validation service for implementing serializable SI-based transactions. We propose to utilize such an approach in the context of the CSI model, as described below.

4.1 Serializable Transactions

For supporting serializable transactions in the CSI framework we introduce a dedicated validation service for detecting read-write and write-write conflicts for all data items at level 1. We introduce a virtual site $S_{vi}$ in the CSI protocol, representing this validation service. All vector clocks contain a field corresponding to this site. In the commit protocol step of the CSI protocol, the transaction sends its read-set and write-set to the validation service. The validation service assigns a sequence number to the transaction if validation is successful, otherwise the transaction is aborted. For the sake of simplicity, first assume that the transaction involves updating of data items at level 1 only. Such a transaction is assigned a sequence number only by the validation site, and no local sequence number is assigned by the transaction execution site. One can logically view this transaction to have executed at the validation site. In the update propagation message, the effective vector clock contains for site $S_{vi}$ a value equal to this sequence number minus 1.

The commit protocol requires additional coordination if the serializable transaction also modifies some data items at some higher numbered levels. Suppose that a level 1 transaction modifies data items at level 1 and 3. In this case the basic CSI protocol needs to be executed with all conflict resolver sites and one also has to get successful validation from the validation service. All conflict resolver sites and the validation service need to make the same decision to either commit or abort the transaction. There are several approaches that can be taken to coordinate this. If the decision is to commit the transaction, then a sequence number is assigned by the local execution site in addition to one assigned by the validation site. The effective vector clock in update propagation has to reflect both these sequence numbers, as noted above.

4.2 Snapshot Isolation based Transactions

The approach for supporting snapshot isolation based transactions is similar to the one presented above for serializable transactions. For data items to be managed at level 2 using the SI model [3], we introduce another virtual site $S_{vi}$ in the CSI protocol. A validation service is used for detecting write-write conflicts among concurrent transactions in regard to all data items at level 2. A validation request to this service contains only the write-set information for the transaction.

If a level 1 transaction involves updating items at levels 1 and 2 then the validation services at both levels 1 and 2 have to coordinate. If it also involves updating of items at level 3, then coordination has to also involve the conflict resolver sites of those items. Similar observation applies when a transaction at level 2 also updates items at the weaker consistency levels.

5. CONSISTENCY LEVELS BELOW CSI

We now present two extensions to the CSI model to support consistency levels that are below the CSI model. In the CSI model, all updates to an item are serialized. 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. Another model which is even more weaker allows unsynchronized concurrent updates to an item. In both these models, the property of causal consistency in update propagation is preserved.

5.1 Commutative Updates

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 [19, 8]. The PSI model also provides support for commutative operations. In the CSI model the following extensions are introduced to support commutative parameters. 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 transactions snapshot, are 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. Thus multiple concurrent transactions at different sites may concurrently modify an item if their operations commute. Their update propagation messages contain the operation name and the parameters rather than the updated values. These updates may get applied in any order 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 if no new updates are applied.

With a steady stream of concurrent commutative updates it is possible that a transaction with a non-commutative operation may repeatedly abort due to write-write conflicts. 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.

### 5.2 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 conflict in regard to data items at this level. This level is useful in case of 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, in case of a log, it does not matter if the records appear in different order at different sites, and the only requirement is that eventually all records are present in the log. The causality property in transaction update propagation is still preserved at this consistency level.

### 6. CONCLUSION

We have presented here a transaction model for replicated data with different consistency guarantees. The transaction model proposed here simultaneously supports transactions with different consistency levels. The Causal Snapshot Isolation (CSI) model serves as the building-block for this transaction management framework. We introduce new mechanisms in this model to provide stronger consistency levels which include serializable transactions and snapshot isolation based transactions. We also introduce mechanisms to support weaker consistency models to support commutative concurrent updates and unsynchronized concurrent updates. Data is organized in a hierarchy which is based on the consistency levels supported by these models. Information flow through transactions between data at different levels is constrained to guarantee the consistency properties of data.

### 7. REFERENCES


---

**Table 1: Data Consistency Levels and Transaction Protocols**

<table>
<thead>
<tr>
<th>Level</th>
<th>Consistency Properties</th>
<th>Transaction Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Strong consistency - Globally serializable transactions</td>
<td>SI-based serializable transactions</td>
</tr>
<tr>
<td></td>
<td>ACID properties for transactions</td>
<td>SI-based transaction</td>
</tr>
<tr>
<td>2</td>
<td>Snapshot Isolation - per item update ordering</td>
<td>Causal Snapshot Isolation</td>
</tr>
<tr>
<td></td>
<td>Global sequence ordering of transactions</td>
<td>with Commutative Updates</td>
</tr>
<tr>
<td>3</td>
<td>Causal ordering of transaction updates; Fork-join model of snapshots for concurrent updates on different items</td>
<td>Causal Snapshot Isolation</td>
</tr>
<tr>
<td></td>
<td>Per item update ordering;</td>
<td>with unordered concurrent updates</td>
</tr>
<tr>
<td>4</td>
<td>Causal ordering of transaction updates</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Permits concurrent commutative updates on an item</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fork-join model of snapshots for all concurrent updates, including commutative concurrent updates on an item</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Causal ordering of transaction updates; Unordered updates – e.g. logging, set insertion</td>
<td></td>
</tr>
</tbody>
</table>


