Transaction and Data Consistency Models for Cloud Applications

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Vinit A Padhye

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Anand Tripathi

February, 2014
Acknowledgements

I am grateful to my advisor Prof. Anand Tripathi for providing me with continuous support and guidance throughout my PhD program which has been immensely helpful to me. My parents and my sister provided the emotional support needed to take up this challenge and complete it successfully. Aai, Baba, and Shilpa - without your support this journey would not have been possible.

I want to thank my committee members, Prof. David Lilja, Prof. Jon Weissman, and Prof. Abhishek Chandra for serving on my committee and providing their guidance on improving this thesis. I am also indebted to a number of other professors from whom I gained immensely valuable knowledge over the course of the past five years.

I am thankful to the Computer Science department, the Graduate School of University of Minnesota, and the National Science Foundation for providing the needed financial support. The research work carried out in this thesis would not have been possible without the financial support from these institutions. The opportunities provided by the Computer Science department to work as a teaching assistance not only supported me financially but also provided a valuable experience that would benefit me in future.

I am grateful and fortunate to have a number of close, supportive, and entertaining friends who made this journey enjoyable and memorable. Neha, Anup, Pratap, Shruti, Vineet and Vineeth - thank you for giving me the most memorable and awesome years of my life.

I am also fortunate and happy that I got the opportunity to spend five amazing years in a beautiful city of Minneapolis. The cold weather of this city provided enough warmth that I could call it my home-outside-India.
Dedication

In dedication to my loving parents and my sister, who supported me throughout this journey.
Abstract

The emergence of cloud computing and large-scale Internet services has given rise to new classes of data management systems, commonly referred to as NoSQL systems. The NoSQL systems provide high scalability and availability, however they provide only limited form of transaction support and weak consistency models. There are many applications that require more useful transaction and data consistency models than those currently provided by the NoSQL systems. In this thesis, we address the problem of providing scalable transaction support and appropriate consistency models for cluster-based as well as geo-replicated NoSQL systems. The models we develop in this thesis are founded upon the the snapshot isolation (SI) model which has been recognized as attractive for scalability.

In supporting transactions on cluster-based NoSQL systems, we introduce a notion of decoupled transaction management in which transaction management functions are decoupled from storage system and integrated with the application layer. We present two system architectures based on this concept. In the first system architecture all transaction management functions are executed in a fully decentralized manner by the application processes. The second architecture is based on a hybrid approach in which the conflict detection functions are performed by a dedicated service. Because the SI model can lead to non-serializable transaction executions, we investigate two approaches for ensuring serializability. We perform a comparative evaluation of the two architectures and approaches for guaranteeing serializability and demonstrate their scalability.

For transaction management in geo-replicated systems, we propose an SI based transaction model, referred to as causal snapshot isolation (CSI), which provides causal consistency using asynchronous replication. The causal consistency model provides more useful consistency guarantees than the eventual consistency model. We build upon the CSI model to provide an efficient transaction model for partially replicated databases, addressing the unique challenges raised due to partial replication in supporting snapshot isolation and causal consistency. Through experimental evaluations, we demonstrate the scalability and performance of our mechanisms.
# Contents

Acknowledgements \hspace{1cm} i  
Dedication \hspace{1cm} ii  
Abstract \hspace{1cm} iii  
List of Tables \hspace{1cm} vii  
List of Figures \hspace{1cm} viii  

1 Introduction \hspace{1cm} 1  
\hspace{1cm} 1.1 Overview of NoSQL systems \hspace{1cm} 2  
\hspace{1cm} \hspace{1cm} 1.1.1 Transactional Consistency in NoSQL Systems \hspace{1cm} 4  
\hspace{1cm} \hspace{1cm} 1.1.2 Data Consistency in NoSQL Systems \hspace{1cm} 5  
\hspace{1cm} 1.2 Research Objectives \hspace{1cm} 6  
\hspace{1cm} 1.3 Thesis Outline \hspace{1cm} 8  

2 Research Problems and Contributions \hspace{1cm} 9  
\hspace{1cm} 2.1 Snapshot Isolation Model \hspace{1cm} 9  
\hspace{1cm} \hspace{1cm} 2.1.1 Serializability Issues in Snapshot Isolation \hspace{1cm} 11  
\hspace{1cm} 2.2 Transaction Support for Cluster-Based Systems \hspace{1cm} 16  
\hspace{1cm} \hspace{1cm} 2.2.1 Design Approach \hspace{1cm} 17  
\hspace{1cm} \hspace{1cm} 2.2.2 Issues in Decoupled Transaction Management Model \hspace{1cm} 18  
\hspace{1cm} \hspace{1cm} 2.2.3 System Architecture Models \hspace{1cm} 19  

iv
2.3 Transaction Support for Geo-replicated Systems .................................. 20
   2.3.1 Prior work on transaction management in replicated database systems .. 20
   2.3.2 Issues in extending SI model for geo-replication .............................. 21
2.4 Research Contributions ................................................................................ 22

3 Transaction Management in Cluster-based Systems .................................. 25
   3.1 Transaction Management Framework ................................................... 26
      3.1.1 Timestamps Management ............................................................... 27
      3.1.2 Eager vs Lazy Update Model ......................................................... 28
      3.1.3 Transaction Validation ................................................................. 29
      3.1.4 Cooperative Recovery ................................................................ 29
   3.2 Decentralized Model for Basic Snapshot Isolation ................................. 30
      3.2.1 Storage System Requirements ....................................................... 30
      3.2.2 Transaction Data Management Model ............................................ 30
      3.2.3 Transaction Management Protocol for Basic SI Model .................... 32
      3.2.4 Cooperative Recovery Protocol .................................................... 35
   3.3 Decentralized Model for Serializable SI Transactions ............................ 35
      3.3.1 Implementation of the Cycle Prevention Approach .......................... 36
      3.3.2 Implementation of the Cycle Detection Approach ............................ 38
   3.4 Service-based Model ............................................................................. 40
      3.4.1 Replicated Service Design ............................................................. 41
      3.4.2 Service Fault Tolerance ................................................................. 44
   3.5 Scalability Evaluations .......................................................................... 45
      3.5.1 TPC-C Benchmark ........................................................................ 46
      3.5.2 Preliminary Evaluations ................................................................. 47
      3.5.3 Scalability Validations on a Large Cluster ....................................... 49
      3.5.4 Scalability of Conflict Detection Service ....................................... 50
      3.5.5 Impact of Transaction Size ............................................................ 51
   3.6 Fault Tolerance Evaluations ................................................................. 52
      3.6.1 Experiment Setup .......................................................................... 52
### 4 Transaction Management in Geo-replicated Systems

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Causal Snapshot Isolation Model</td>
<td>59</td>
</tr>
<tr>
<td>4.1.1</td>
<td>System Model</td>
<td>60</td>
</tr>
<tr>
<td>4.1.2</td>
<td>CSI Model</td>
<td>60</td>
</tr>
<tr>
<td>4.2</td>
<td>Causal Snapshot Isolation with Partial Replication</td>
<td>63</td>
</tr>
<tr>
<td>4.3</td>
<td>Issues in supporting causal consistency under partial replication</td>
<td>64</td>
</tr>
<tr>
<td>4.4</td>
<td>Partitioned-CSI Model</td>
<td>67</td>
</tr>
<tr>
<td>4.4.1</td>
<td>System Model</td>
<td>67</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Vector Clocks and Partition Dependency View</td>
<td>67</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Transaction Execution Protocol</td>
<td>69</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Execution of Multi-site Transactions</td>
<td>77</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Discussion of Protocol Correctness</td>
<td>79</td>
</tr>
<tr>
<td>4.4.6</td>
<td>Example</td>
<td>81</td>
</tr>
<tr>
<td>4.4.7</td>
<td>Implementation Details</td>
<td>83</td>
</tr>
<tr>
<td>4.5</td>
<td>Evaluations</td>
<td>85</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Experiment Setup</td>
<td>86</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Scalability of P-CSI</td>
<td>87</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Impact of Non-local Transactions</td>
<td>90</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Impact of Transaction Size</td>
<td>91</td>
</tr>
<tr>
<td>4.5.5</td>
<td>Cost of Ensuring Causality</td>
<td>91</td>
</tr>
<tr>
<td>4.6</td>
<td>Session Consistency Model</td>
<td>94</td>
</tr>
<tr>
<td>4.7</td>
<td>Summary</td>
<td>97</td>
</tr>
</tbody>
</table>

### 5 Conclusion and Future Directions

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>References</td>
<td></td>
<td>102</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Conflict Table Example ........................................ 42
3.2 Participant Table Example .................................. 43
3.3 Coordinator Table Example ................................. 43
3.4 Impact of Transaction Size .................................... 52
4.1 Initial State of Sites ........................................... 81
4.2 System Performance on Amazon EC2 (Using InMemory Storage) ........................................ 89
4.3 Impact of Transaction Size .................................... 92
4.4 Impact of Read/Write Set Size .............................. 93
4.5 Impact of Propagation Frequency .......................... 93
List of Figures

2.1 Snapshot Isolation Model ............................................. 11
2.2 Pivot transaction ..................................................... 12
2.3 Non-Serializable Execution of $T_1$ and $T_2$ ......................... 13
2.4 Potential dependency with a committed concurrent transaction ........ .... 14
2.5 Potential dependency with a committed non-concurrent transaction ........ 15
2.6 Decentralized and Decoupled Transaction Management Model ................. 18
3.1 Transaction Protocol Phases ........................................... 26
3.2 STS and GTS Counters ............................................... 27
3.3 TransactionTable Structure ............................................. 31
3.4 StorageTable Structure ............................................... 32
3.5 Transaction throughput under the scale-out model ......................... 47
3.6 Average transaction response times under the scale-out model ............... 47
3.7 Execution time for different protocol phases ................................ 48
3.8 Transaction throughput under the scale-out model ......................... 50
3.9 Average transaction response times under the scale-out model ............... 50
3.10 Scalability of the conflict detection service ................................ 51
3.11 Abort Statistics for Different Failure Rates (Timeout values ranging from 50ms to 500ms) .................................................. 53
3.12 Average Time to detect failures in Validation phase ....................... 54
3.13 Average Time to detect failures in CommitIncomplete phase ............... 55
3.14 Average gap between STS and GTS for different failure rates and timeout values 56
4.1 Example of Effective Causal Snapshot ........................................ 63
4.2 Issues in supporting causality under partial replication ................. 65
4.3 Causally inconsistent snapshot due to remote reads ..................... 66
4.4 Issues in obtaining atomically consistent snapshot ....................... 67
4.5 Transaction Throughput Scalability ......................................... 84
4.6 Transaction Throughput Scalability with InMemory Storage ............ 88
4.7 Transaction Throughput Scalability with HBase Storage ................. 88
4.8 Transaction Latency .......................................................... 90
4.9 Visibility Latency ............................................................. 91
4.10 Impact of Non-Local Transactions ......................................... 92
Chapter 1

Introduction

Recent years have seen emergence of the cloud computing model, in which organizations owning large computing infrastructure provide their computing resources in on-demand manner to build and deploy services and applications. Today, such cloud computing platforms are being utilized by many businesses as it relieves them from the burden of procuring and managing large computing infrastructure. Along with the emergence of cloud computing, we have also seen emergence of large scale web-based services such as web search, social networking, online stores, etc. The development and operation of such services is greatly benefited by the cloud computing platforms as they provide an elastic pool of computing resources to dynamically scale the service by adding or removing computing resources on-demand. This model of scalability is referred to as the scale-out model or horizontal scaling model. The scale-out model allows the services or application to efficiently handle surges in load by dynamically allocating more resources to meet the load demands.

The emergence of cloud computing model and large-scale Internet services presents several research challenges related to data management. These challenges have given rise to new classes of data management systems, commonly referred to as NoSQL systems [12, 17, 29]. The designs of the NoSQL systems were largely dictated by the requirements of large scale services. Such services typically have millions of users and are required to manage large amount of service data. Scalability and high availability are crucial requirements of such services and they often have
stringent response time requirements. The scalability and availability properties of traditional relational database systems were found to be inadequate to address the requirements of today’s large-scale Internet services [13, 12]. Furthermore, the nature of cloud computing environments also influenced the designs of NoSQL systems. In such a large environment, component failure is expected and hence fault tolerance is an essential requirement of any cloud-based system. Owing to these requirements, the design of the NoSQL systems deviated significantly from the traditional SQL-based relational database systems as described below.

1.1 Overview of NoSQL systems

The NoSQL systems have adopted a simple key-value based model where a data item is represented in form of a \(<key, value>\) pair, where \textit{key} serves as the primary key for accessing the item. Some systems such as Bigtable [12] and Cassandra [29] provide a richer data model, in which the value is composed of multiple \textit{columns} and individual columns can be grouped into different \textit{column families}. A key-value pair acts as the unit of distribution, replication and data access. NoSQL systems do not provide relational database features such as secondary index or join operations.

Scalability and availability in NoSQL system is typically achieved through replicating and distributing data across multiple nodes. The nodes may belong to a LAN-based cluster in a single cloud datacenter or may be distributed across multiple datacenters or database sites situated in different geographic regions. Geo-replication is typically used when the service users are distributed across multiple geographic regions. Geo-replication can also be used for disaster tolerance. The replication model can either use full replication scheme or a partial replication scheme. In a full replication scheme, a data item is replicated across all nodes/sites in the system, and thus a single node/site contains the entire database. In a partial replication scheme, a data item is replicated across one or more nodes/sites and a single node/site does not contain the entire database. Typically, on a datacenter or cluster level, NoSQL systems use partial replication since the database size is often large enough such that a single node can not store the entire database. A geo-replicated system typically consists of multiple geographically distributed datacenters and it can either use partial or full replication scheme when distributing
data across datacenters. In certain cases partial replication may be desired when data items need not be replicated across all datacenters, taking into account the data access patterns. For example, in an online bookstore, service data related to books of a particular language may only need to be replicated in corresponding geographic regions.

Many NoSQL systems use a form of partial replication scheme, called as horizontal partitioning or sharding. In this scheme, the data item space is partitioned into multiple disjoint partitions or shards and each node/site stores one or more partitions. The partitioning scheme can either be ordered partitioning in which a partition contains a sequential range of keys (e.g Bigtable), or it can be hash-based partitioning in which keys are assigned to nodes based on some hashing scheme (e.g Cassandra). In contrast, in vertical partitioning, which is often used in distributed relational databases, a relational table is partitioned into disjoint subsets of columns and such partitions are distributed across multiple nodes. Compared to vertical partitioning, sharding provides more flexibility and scalability for the following reasons. As the database size grows in the number of data items (rows), the database can be divided into more shards. These shards can then be distributed across more number of nodes by increasing the system size, thereby providing more scalability. In contrast, vertical partitioning provides less flexibility since the partitioning is done on columns rather than rows and the number of columns for a table are typically fixed and rarely change. Therefore, NoSQL systems have adopted the model of horizontal partitioning.

Data replication poses fundamental trade-offs between various properties of a replicated database system, namely consistency, availability, partition tolerance, and operation latency. The CAP conjecture [22] describes the interplay between consistency, availability and partition tolerance and states that under network partitions a system can not provide both consistency and availability and must compromise one of the two properties. With respect to geo-replicated systems, there are further trade-offs between consistency and operation latency that arise even under normal operations, i.e without network partitions. The requirement of low latency, which is crucial in many current web applications, typically makes it difficult to provide strong consistency in geo-replicated systems because of high communication latencies in wide-area networks. Addressing these trade-offs, the NoSQL systems have generally adopted design approaches that provide weaker form of consistency levels in comparison to relational databases. In regard to
the consistency models of the NoSQL systems, we classify two aspects of system consistency: \textit{transactional consistency} and \textit{data consistency}. We elaborate on these two aspects and discuss the spectrum of consistency models provided by the NoSQL systems.

1.1.1 Transactional Consistency in NoSQL Systems

A transaction is a primitive provided for executing a sequence of operations on a set of data items in atomic manner. The consistency guarantees provided by a transaction system are expressed in form of ACID guarantees which represents guarantees of \textit{atomicity}, \textit{consistency}, \textit{isolation}, and \textit{durability} properties. The \textit{serializability} guarantee is the strongest consistency guarantee provided by a transaction system, which states that any concurrent execution of a set of transactions is guaranteed to produce the same effect as executing them in some serial order. With regard to transactional consistency guarantees, many NoSQL systems either do not provide any transaction semantics or provide transaction semantics with certain limitations. For example, systems such as Bigtable [12] and HBase [4] provide only single-row transaction. Megastore [7], and G-store [16] provide transactions only over a predefined group of data items. Sinfonia [2] provides a restrictive form of transaction semantics. The primary reason for not supporting serializable transactions is that it requires distributed synchronization among the nodes storing the data items involved in the transaction, thereby limiting the scalability of the system. For certain applications, such as web search and social networking, such limited support for transactions has been found to be adequate. However, many applications such as online shopping stores, online auction services, financial services, while requiring high scalability and availability, still need strong transactional consistency guarantees.

Recently, some researchers have addressed this problem and proposed techniques for providing richer models for transaction support. However, there are various issues which still need further investigation. For example, the Percolator system [41] developed over Bigtable provides multi-item or multi-row transactions but does not guarantee serializability. The Calvin system [53] provides serializable transactions but requires that the read and write set items of a transaction be known in advance. The Spanner [14] system developed by Google provides serializable transaction for geo-replicated data. However it’s implementation is based on using
physical clocks in contrast to logical clocks and requires special purpose hardware such as GPS clocks or atomic clocks. Thus the approach taken in that work can not be easily utilized in other systems.

1.1.2 Data Consistency in NoSQL Systems

With regards to data consistency, the trade-off is between consistency level on one hand and availability and operation latency on other hand. In a replicated data storage system, *strong data consistency* requires that for a data item all the storage nodes storing a copy of that item must always reflect the same value for the item. A weaker form of data consistency level, called *eventual consistency*, requires that if no new updates are made to the data item then all storage nodes storing a copy of the item will eventually converge on the last updated value of the item. Strong data consistency requires *synchronous replication* in which the updates of a transaction are propagated synchronously to other replicas before committing the transaction. This incurs high latencies and reduces availability under network partitions or node failures. On the other hand, *eventual consistency* can be provided through *asynchronous replication* which provides lower latencies and higher availability. In asynchronous replication, the updates are first committed locally and asynchronously propagated to other replicas at a later time. While providing strong data consistency in a LAN-based cluster environment is possible, as demonstrated by systems such as Bigtable, it is found to be impractical in case of large geo-replicated systems [17, 13]. Therefore, many geo-replicated systems have adopted the model of eventual consistency through asynchronous replication. However, the guarantees provided by eventual consistency model are not adequate for many applications and it also puts burden on the application programmers to program their applications to deal with the inconsistencies in acceptable manner.

There are various other consistency models that fall between eventual consistency and strong consistency, such as *causal consistency* [30] or *session consistency* [52]. The causal consistency model defines causal relationships between events in a distributed system, such as sending or receiving messages, updating state or a data item, etc. The causal consistency model guarantees that if the effect of an event is visible to a process, then effects of all the causally preceding
events are also visible to the process. Causal consistency provides stronger guarantees than the eventual consistency model. For example, consider a social networking application such as twitter where users posts and replies to status messages. If Alice posts a message and Bob replies to that message, then under causal consistency, if another person sees Bob's message then the message from Alice to which Bob replied will also be visible to him. This guarantee is not ensured under the eventual consistency model. In session consistency model, the user or an application session observes a database state consistent with it’s own view of the database based on the read/write operations performed by it. This model includes various guarantees such as read-your-writes consistency [52] which guarantees that the session observes the effects of its own write operations. These guarantees are further discussed in Chapter 4.

Recently, due to the limitations of the eventual consistency model mentioned above, use of a causal consistency has been proposed for geo-replicated systems [50, 33]. Causal consistency model can be supported using asynchronous replication scheme and hence attractive for geo-replicated systems. The examples of geo-replicated systems supporting the causal consistency include PSI [50] and COPS [33]. However, the approaches taken in both these systems have certain drawbacks. For example, the approach for ensuring causal consistency in PSI can induce false causal dependencies leading to unnecessary delays in applying updates at remote sites. The approach taken in COPS requires maintaining and communicating metadata for each item version to indicate its causal dependencies. Moreover, both PSI and COPS systems do not provide efficient support for partial replication. They also do not provide the session consistency guarantees.

1.2 Research Objectives

The initial classes of cloud-based applications for which NoSQL systems were developed, such as web search or social networking applications, did not require strong consistency guarantees. In recent years though many other classes of applications have started utilizing cloud computing platforms and cloud-based NoSQL systems. However, as described above, the lack of transaction support and appropriate consistency models have been an obstacle in this adoption. This
thesis addresses this problem and develops techniques for supporting transactions for cloud applications. The goal is to provide adequate consistency levels without compromising the crucial requirements of NoSQL systems such as scalability and availability. The thesis focuses on both the transactional consistency and the data consistency aspects and develops appropriate models and techniques for cluster-based environment as well as geo-replicated systems.

One of the main foundation of the models and techniques developed in this work is the use of the Snapshot Isolation (SI) \cite{8} model of transaction execution. Snapshot Isolation model was first proposed in \cite{8} for transaction execution in relation database systems. The SI model is attractive for scalability, as observed in the past \cite{8}. The SI model provides more concurrency compared to other concurrency control models such as two-phase locking, since read operations are not blocked due to concurrent write operations. Due to this advantage, the SI model has been widely adopted in both research prototypes as well as commercial database systems such as Oracle and Microsoft SQL Server. However, the SI model has certain drawbacks, such as lack of guarantee in ensuring serializability. In the past, some researchers have investigated this problem in the context of relational database systems and have developed techniques for providing serializable SI-based transactions. The use of this model in NoSQL systems has been proposed only recently \cite{41, 56}. However, the techniques proposed in \cite{41} and \cite{56} do not ensure serializability. Our goal is to develop scalable techniques for supporting serializable SI-based transaction execution in NoSQL systems.

We distinguish between the problem of supporting transactions in cluster-based NoSQL systems and the problem of providing transactions in geo-replicated NoSQL systems, and investigate them separately for following reasons. Typically, in a cluster-based NoSQL system deployed in single datacenter, providing strong data consistency is possible, as demonstrated by Bigtable or HBase system. Thus, in such environments providing strong transactional guarantees such as serializability in a scalable and efficient manner is feasible, as demonstrated in our work. However, providing serializability guarantee in geo-replicated systems can be impractical. Ensuring serializability requires that even read-only transactions must undergo a validation phase to check for serialization conflicts with other concurrent transactions, which requires remote communication. This induces significant overheads due to high latencies in wide-area networks. Therefore, in case of geo-replicated systems our goal is to provide a weaker but useful consistency level that
can be efficiently implemented in wide-area environments.

1.3 Thesis Outline

In the next chapter, we describe in detail the specific research problem addressed in thesis and our research contributions. We describe in detail the SI model, which forms the basis of our transaction management model, and discuss the serializability issues in SI model. We discuss the problem of supporting transaction in cluster-based NoSQL systems and briefly outline the conceptual approach taken by us. Next, we discuss the problem of transaction management in geo-replicated systems and the issues in extending the SI model for geo-replication and supporting causal consistency. Finally, we outline the main contributions of the research work presented in this thesis.

Chapter 3 discusses the problem of providing transaction support for cluster-based NoSQL systems and the techniques we developed towards addressing this problem. We first describe the design framework for supporting ACID transactions and the various design issues involved in it. We then describe the protocol for supporting basic snapshot isolation model and then discuss how to extend it to guarantee serializability. We also present two system architectures for transaction management. Through experimental evaluations, we demonstrate the scalability of our techniques and the system architectures developed by us.

In Chapter 4 we discuss the transaction model for geo-replicated NoSQL systems. We first present a transaction model for full-replication scheme and then adapt it to develop a model for partially replicated databases. The last chapter presents the conclusion and discusses the future research directions.
Chapter 2

Research Problems and Contributions

In this chapter, we discuss in detail the specific research problems addressed in this thesis. As discussed earlier, the Snapshot Isolation model of transaction executions forms the basis of our approaches. Our goal is to develop scalable techniques for SI-based transactions and provide appropriate consistency models that can be efficiently supported in cluster-based systems and geo-replicated systems. We describe below the Snapshot Isolation model.

2.1 Snapshot Isolation Model

Snapshot isolation (SI) based transaction execution model is a multi-version based approach utilizing optimistic concurrency control [28]. A multi-version database [9] maintains multiple versions of data items, in contrast to maintaining only a single copy of an item. An update operation on an item creates a new version of that item. Multi-version databases typically use timestamps to indicate different versions of items. The main purpose of multi-version databases is to provide efficient concurrency control for concurrent transactions. There are also other benefits of multi-version databases, such as time-travel queries which allows to query the database state which existed at some point in the past. A non-multi-version based database typically uses locks.
to ensure that a transaction reading an item does not see the effect of a concurrent transaction which is modifying the same item. The multi-version concurrency control (MVCC) [9] provides a better approach by maintaining multiple versions of items. Thus, a transaction can safely read a committed version without the need of waiting for locks. In snapshot isolation, which is a form of MVCC, a transaction obtains a *snapshot* of the database at its start time. The read operations performed by the transactions observe only the latest committed versions according to this snapshot. Thus, the read operations of a transaction can be performed concurrently with write operations of other concurrent transactions, thereby providing more concurrency compared to lock-based approaches. We describe below the snapshot isolation model in detail.

In this model, when a transaction $T_i$ commits, it is assigned a commit timestamp $T_{Si}^c$, which is a monotonically increasing sequence number. The commit timestamps of transactions represent the logical order of their commit points. For each data item modified by a committed transaction, a new version is created with the timestamp value equal to the commit timestamp of the transaction. When a transaction $T_i$’s execution starts, it obtains the timestamp of the most recently committed transaction. This represents the *snapshot timestamp* $T_{Si}^s$ of the transaction. A read operation by the transaction returns the most recent committed version up to this snapshot timestamp, therefore, a transaction never gets blocked due to any write locks. Two transactions $T_i$ and $T_j$ are concurrent if, and only if, $(T_{Si}^s < T_{Si}^c) \land (T_{Si}^s < T_{Sj}^c)$

A transaction $T_i$ commits only if there exists no committed transaction concurrent with $T_i$ which has modified any of the items in $T_i$’s write-set. That means there exists no committed transaction $T_j$ such that $T_{Si}^s < T_{Sj}^i < T_{Si}^c$ and $T_j$ has modified any of the items in $T_i$’s write-set. Thus, if two or more concurrent transactions have a write-write conflict, then only one of them is allowed to commit. It is possible that a data item in the read-set of a transaction is modified by another concurrent transaction, and both are able to commit. 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 dependency can be considered as an incoming dependency for $T_j$ and an outgoing dependency for $T_i$. This is the only kind of dependency that can arise in the SI model between two concurrent transactions. It is also possible to have mutual anti-dependencies between two concurrent transactions implying a cycle. There are other kinds of dependencies, namely *write-read (wr)* and *write-write (ww)*,
that can exist between two non-concurrent transactions.

We illustrate the SI model with an example shown in Figure 2.1. In this example, transaction $T_1$ reads items $x$ and $y$, and writes item $w$, whereas transaction $T_2$ updates items $x$ and $w$. Figure shows the versions for different items (indicated by the solid dots in the figure) with their associated timestamps. Thus, for example, the first solid dot in the version timeline for item $x$ indicates a version of item $x$ created by transaction with timestamp 80. Transaction $T_1$ starts after the start of transaction $T_2$ but before $T_2$ is committed. Thus, $T_1$ and $T_2$ are concurrent. Transaction $T_1$ starts at a point where the latest committed transaction has commit timestamp 100, thus $T_1$ starts with a snapshot timestamp of 100. Concurrently, $T_2$ commits and creates versions of items $x$ and $w$ with version timestamp of 120. When $T_1$ performs read operation for item $x$, according to SI model the version visible to $T_1$ should be the latest version according to $T_1$’s snapshot. Thus, in this case $T_1$ will read version of $x$ with timestamp 95, rather than reading the version created by $T_2$ with timestamp 120. If $T_1$ modifies item $w$, then it results in a write-write conflict since a committed concurrent transaction $T_2$ has already updated item $w$. In this case, $T_1$ will be aborted.

![Figure 2.1: Snapshot Isolation Model](image)

### 2.1.1 Serializability Issues in Snapshot Isolation

Since the snapshot isolation model was first proposed, it was identified that it can lead to non-serializable executions [8, 20]. When the execution of a set of transactions creates a cycle of
dependencies, such an execution is non-serializable because it is not equivalent to any serial order of those transactions. In the SI model, an execution of a set of transaction can lead to dependency cycles, thus making it non-serializable. It was shown in [1] that in a non-serializable execution of a set of transactions executing under the SI model, the cycle of dependencies must always include at least one anti-dependency. Fekete et al. [20] improved upon this theory further and showed 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 [20] with both incoming and outgoing anti-dependencies. Figure 2.2 shows an example of a pivot transaction. In this example, $T_2$ is the pivot transaction.

We illustrate the serialization anomalies with an example. Consider a banking application which uses SI-based databases. The bank provides joint accounts and allows to withdraw from any of the two joint accounts provided that the total balance of both the accounts does not go below zero. In this example, database contains two accounts $X$ and $Y$ with current balance of $50$ each, i.e. $X = 50$ and $Y = 50$. A withdraw transaction $T_1$ wishes to withdraw $100$ from account $X$. It reads current balance of both $X$ and $Y$ as $50$ and since the withdrawal is permitted it withdraws $100$ and updates the balance of $X$ to -50. However, consider that another withdraw transaction is running concurrently which is also withdrawing $100$ from account $Y$. It will similarly read balance of $X$ and $Y$ as $50$ and withdraw $100$ from $Y$ and update the balance of $Y$ to -50. This leads to a database state where the balance of both the accounts is -50. This violates the integrity of the database. The interleaving of operations of $T_1$ and $T_2$ is shown in figure 2.3. Note that this execution is not equivalent to any serial execution of $T_1$ and $T_2$, thus violating serializability. This anomaly is also called as write-skew.

Figure 2.2: Pivot transaction
In the context of traditional RDBMS, several techniques [10, 11, 43, 24] have been developed to ensure serializability utilizing above concepts. We classify these approaches in two categories, as described below.

**Cycle Prevention Approach**

This approach is based on preventing the conditions that can lead to non-serializable execution of transactions under the SI model. This approach utilizes the theories developed in [1] and [20] which identify the conditions for non-serializable executions as explained above. There are two techniques that come under this approach: the first technique is to prevent the formation of anti-dependencies and the other technique is to prevent formation of pivot transactions. Specifically, in the first technique, when two concurrent transactions $T_i$ and $T_j$ have an anti-dependency, one of them is aborted. This ensures that there can never be a *pivot transaction*, thus guaranteeing serializability. In the context of RDBMS, this approach was investigated in [10]. In the second technique, a transaction is aborted when it forms an incoming as well as an outgoing anti-dependency (thus becoming a pivot). In the context of RDBMS, this approach was investigated in [11]. The first technique is simpler than the second technique, because in the second technique we need to track anti-dependencies for each transaction to detect the emergence of a pivot transaction.

---

**Figure 2.3: Non-Serializable Execution of $T_1$ and $T_2$**

<table>
<thead>
<tr>
<th>Transaction $T_1$</th>
<th>Transaction $T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>begin</td>
</tr>
<tr>
<td>read($X$) → 50</td>
<td>read($X$) → 50</td>
</tr>
<tr>
<td>read($Y$) → 50</td>
<td>read($Y$) → 50</td>
</tr>
<tr>
<td>write($X = -50$)</td>
<td>write($Y = -50$)</td>
</tr>
<tr>
<td>commit</td>
<td>commit</td>
</tr>
</tbody>
</table>
Cycle Detection Approach

This approach is based on explicitly detecting the occurrence of a cycle of dependencies and aborting one of the transaction involved in the cycle to ensure serializability. Specifically, a transaction is aborted when a dependency cycle involving that transaction is detected during its commit protocol. This approach is more complicated than the cycle prevention approach and requires maintaining the dependency graph, called as Dependency Serialization Graph (DSG) [20], for the transactions. In this graph, transactions are represented by nodes and the edges represent the dependencies between them. If during the commit operation of a transaction a dependency cycle is detected involving that transaction, then that transaction is aborted to break the cycle.

In the context of RDBMS, this approach was investigated in [43].

Implementing the above approach requires various issues to be examined. First issue is that even after a transaction commits, we need to maintain some information about it in the dependency graph until some point in the future. Such committed transactions are referred to as zombies in [43]. Also, for efficient execution the dependency graph should be kept as small as possible by frequent pruning to remove any unnecessary transactions that can never lead to any cycle in future.

The read-set and the write-set of a transaction are precisely known only at its commit time, because a transaction may determine which items to read or write during its execution, which may take any arbitrary time. At a transaction’s commit time, all dependencies with previously committed transactions are precisely known. At the commit time, we may detect any \( ww \), \( wr \), and \( rw \) dependencies for this transaction in relation to any of the previously committed transactions. It can have only \( wr \), \( ww \) and \( rw \) incoming dependencies with a previously committed transaction.
non-concurrent transaction. With respect to a committed concurrent transaction, it can have only \textit{rw} dependencies, possibly in both directions. However, any \textit{rw} dependencies to or from any of the currently active transactions cannot be precisely known until that transaction commits. We illustrate below these concepts through two examples.

Figure 2.4(a) shows three transactions, two of which, T1 and T2, are committed. At the time when T2 commits, T3 is active. The \textit{rw-edge} from T2 to T1, and \textit{wr-edge} from T1 to T3 are known at this time. We need to keep this information in the dependency graph because when T3 commits an \textit{rw} dependency from T3 to T2 is found, as shown in Figure 2.4(b). At this time a cycle is formed, indicating a non-serializable execution. This leads us to conclude that when a transaction $T_i$ commits, its node and related edges should still be retained in the dependency graph if there are any other transactions active, i.e. concurrent and not yet reached their commit points, at $T_i$'s commit time. Moreover, all transactions reachable from the committing transaction, such as T1 in this example, should also be retained in the graph.

![Figure 2.5: Potential dependency with a committed non-concurrent transaction](image)

Figure 2.5 shows an example illustrating that sometimes \textit{rw}, \textit{ww}, and \textit{wr} dependencies for a transaction may become known only at its commit time. In this example, when T3 commits, an \textit{rw} dependency to T2, and \textit{ww} and \textit{wr} dependencies from T1 are discovered. This situation results in a cycle.

**Comparison of the approaches for serializability**

Since the cycle prevention approach is a preventive measure, it can sometimes abort transactions that may not lead to serialization anomalies. For example, if an execution of a set of
transaction forms a pivot transaction, i.e. a transaction which has both incoming and outgoing anti-dependencies, but does not lead to create a cycle of dependencies then such an execution is serializable. In this case the abort of the pivot transaction by cycle prevention approach is unnecessary. In contrast, the cycle detection approach aborts only the transactions that can cause serialization anomalies since it explicitly checks for dependency cycles. However, the cycle detection approach requires tracking of all dependencies for every transaction and maintaining a dependency graph to check for cycles. In contrast, in the cycle prevention approach we only need to track anti-dependencies and there is no need for maintaining a dependency graph to check for cycles. Thus, the cycle prevention approach incurs a relatively small overhead compared to the cycle detection approach, but at the cost of some unnecessary aborts.

2.2 Transaction Support for Cluster-Based Systems

We now discuss the challenges in supporting transactions in cluster-based NoSQL systems. As discussed in the previous chapter, providing strong data consistency for replicated data in a cluster environment is feasible. Our primary goal is to support SI-based transactions with serializability guarantees for NoSQL systems that provide strong data consistency. Our objective is to provide transaction support without affecting the scalability and availability properties of the NoSQL system.

As mentioned earlier, the reason for not supporting general multi-row transactions is that it requires distributed synchronization between multiple storage system nodes, since the data items accessed by a transaction can span across multiple storage nodes. As the system is scaled, typically the data gets distributed across larger number of nodes. Thus the synchronization overhead required in executing a transaction increases as the system is scaled, thereby limiting the scalability of the system. We address this problem by taking a different design approach towards supporting transactions in NoSQL system. This approach is mainly based on the design principle of decoupling the transaction management functions from the NoSQL storage system. We first describe below the design approach taken by us and the rationale behind it. We then discuss the unique challenges in realizing this approach.
2.2.1 Design Approach

Our approach is based on decentralized and decoupled transaction management where transaction management functions are *decoupled* from the storage system and performed by the application-level processes themselves, in *decentralized* manner. Figure 2.6 illustrates the seminal elements of this approach. A service hosted in a cloud datacenter environment is accessed by clients over the Internet. The service creates application level processes for performing service functions. These processes belong to the trusted domain of the deployed service. The application-specific data of the service is stored in a key-value based storage system in the datacenter. In our approach, all transaction management functions – such as concurrency control, conflict detection and atomically committing the transaction updates – are performed by the application processes themselves in decentralized manner. These functions are provided to the application in the form of library functions. The metadata necessary for transaction management such as transaction status, read/write sets information, and lock information are is stored in the underlying key-value based storage. The rationale behind this is as follows. The NoSQL system already provides guarantee of low-latency, scalable and reliable access to data as well durability/persistence of data under failures. The integrity of the transaction metadata is crucial for correctness Moreover, this data will be accessed in highly concurrent manner by different transactions. Therefore, it makes sense to utilize the NoSQL system itself for the purpose of storing transaction management data. In summary, our approach has three key points: (1) To decouple the transaction management functions from the storage system (2) To integrate this functions in application level-process as a part of library functions, and (3) To utilize the storage system itself to store the transaction management related metadata.

In the past, some researchers have proposed designs for transaction management based on similar notions. In [34], the authors proposed an approach based on unbundling the transaction management component from the data storage component. A system based on this approach was presented in [31], which includes a a central transaction component/service that provides the transaction functionality. All transaction management function are performed by this component which is deployed using a single server. However, they did not address the problem of scalability of the transaction component which can easily become a performance bottleneck.
under high load. CloudTPS [55] provides a design based on a separate replicated transaction management layer deployed over the underlying storage system, and the ACID transactions are provided by this layer. The applications are not supposed to directly access the underlying storage system. The transaction management layer caches the application data and distribute it over replicated transaction managers. The transaction guarantees are achieved using the two-phase-commit protocol between the transaction managers. However, the CloudTPS design assumes that the workload mainly consists of small transactions that accesses small number of well-identified items. In contrast to these approaches, our design presents a system based on purely decoupled and decentralized design, wherein the transaction management functions are performed by the application process itself. This provides a natural way towards scalability by scaling the application layer. Furthermore, we also present a design based on hybrid approach where the conflict detection is performed using a replicated service.

### 2.2.2 Issues in Decoupled Transaction Management Model

In realizing the transaction management model described above, the following issues need to be addressed. These issues are related to the correctness and robustness of the decentralized transaction management protocol. In our approach, the commit protocol is performed in various
steps by individual application processes, and the entire sequence of steps is not performed as a single atomic action. Not performing all steps of the commit protocol as one atomic action raises a number of issues. The transaction management protocol should ensure the transactional consistency, i.e. the guarantees of atomicity, isolation, consistency and durability, when multiple processes execute the protocol steps concurrently. Moreover, any of these steps may get interrupted due to process crashes or delayed due to slow execution. Such events should not block the progress of other concurrently running transactions. To address this problem, the transaction management protocol should support a model of cooperative recovery; any process should be able to complete any partially executed sequence of commit/abort actions on behalf of another process that is suspected to be failed. Any number of processes may initiate the recovery of a failed transaction, and such concurrent recovery actions should not cause any inconsistencies.

We discussed above how SI-based transactions can lead to serialization anomalies and the techniques for detection and preventing such anomalies. The techniques based on these concepts have been investigated in the context of RDBMS, however their use in the context of NoSQL system remains to be investigated. In this thesis, we develop and implement techniques for both the prevention and detection based approaches for serializable SI transactions for NoSQL systems. These techniques are developed in the context of the design model.

2.2.3 System Architecture Models

Based on the decoupled and decentralized transaction management model, we develop two system architectures. The first architecture is fully decentralized, in which all the transaction management functions are performed in decentralized manner by the application-level processes. The primary advantage of this architecture model is that it does not rely on any central component or service (except the NoSQL storage system) and therefore does not have any scalability bottlenecks of single point of failures. The second architecture is a hybrid model in which only the conflict detection functions are performed using a dedicated service and all other transaction management functions are performed in decentralized manner. We refer to this as service-based
architecture. The advantage of this approach over the first approach is that the conflict detection, which is typically the bottleneck phase in commit protocol and has significant overhead, is performed more efficiently by a dedicated services, providing lower latencies for transaction execution. The disadvantage of this approach is that service can become a scalability bottleneck or single point of failure. To address this problem, we design this service as a replicated system. In this thesis, we design and implement both the architecture models and perform their comparative evaluations.

2.3 Transaction Support for Geo-replicated Systems

We now discuss the issues in supporting transactions in geo-replicated systems. In contrast to cluster-based systems, providing strong consistency in geo-replicated system is not feasible, as identified by the designers of various geo-replicated systems such as Dynamo [17] and PNUTS [13]. Thus, providing strong transaction consistency guarantees such as serializability is not practical in such environments. Our goal therefore is to provide a weaker but meaningful transaction model that can be efficiently implemented in geo-replicated systems. Our goal is to provide a consistency level stronger than the eventual consistency, which can be implemented with asynchronous replication model. We first discuss the issues in transaction management for geo-replicated database systems.

2.3.1 Prior work on transaction management in replicated database systems

Transaction execution models in replicated database systems broadly fall into two categories: symmetric execution model where transactions can be executed at any site containing a replica, or asymmetric model where the transaction is executed only on some designated sites. In cluster-based environments, approaches for data replication management have mainly centered around the use of the state machine model [40, 46] using atomic broadcast protocols. Examples of such approaches include [19, 25, 26, 27, 32, 39, 42, 51]. The approaches based on atomic broadcast are typically not suitable for geo-replicated systems due to high communication latencies in
wide-area environments. The issues with scalability in data replication with strong consistency requirements are discussed in [23]. Such issues can become critical factors for data replication in geographically replicated databases.

Recently, many data management systems for cloud datacenters distributed across wide-area have been proposed [17, 13, 7, 33, 50]. Dynamo [17] uses asynchronous replication with eventual consistency and do not provide transactions. PNUTS [13] also does not provide transactions, but provides a stronger consistency level than eventually consistency, called as *eventual timeline consistency*. Megastore [7] provides transactions over a group of entities using synchronous replication. COPS [33] provides causal consistency, but does not provide transaction functionality, except for snapshot-based read-only transactions. PSI [50] provides transaction functionality with asynchronous replication and causal consistency.

Approaches for transaction management in partially replicated systems are presented in [49, 45, 5, 44]. The notion of *genuine partial replication* introduced in [44] requires that the messages related to a transaction should only be exchanged between sites storing the items accessed by the transaction. These approaches support 1-copy-serializability. In contrast, the approach presented in [47] is based on the snapshot isolation model, providing the guarantee of 1-copy-SI. These approaches are based on the database state machine model [40], utilizing atomic multicast protocols, and therefore are not suitable for WAN environments. In [48] a transaction management model for WAN environments is proposed, but it relies on a single site for conflict detection in the validation phase. The system presented in [5] uses the notion of generalized snapshot isolation (GSI) [18], where a transaction can observe a consistent but old snapshot of the database.

### 2.3.2 Issues in extending SI model for geo-replication

Extending the centralized SI model to replicated databases is a complex problem. The first issue is the requirement of reading from the latest snapshot, which requires that a transaction’s updates are propagated synchronously to other sites. Synchronous replication, especially under wide-area environments, can limit the performance and scalability. Other researchers [50, 18] have recognized these limitations and have proposed solutions in which a transaction may read
from an older but consistent snapshot. The second problem is ensuring the total ordering of transactions across sites. This would require a global sequencer mechanism which can be another scalability bottleneck.

The PSI [50] model which was proposed recently provides a weaker isolation and consistency level suitable for wide-area environments. It allows transactions to read an older but consistent snapshot and does not enforce total ordering of transactions across sites but ensures causal consistency. However, the PSI model has certain drawbacks. The first drawback is that it can induce false causal dependencies which can unnecessarily delay the application of updates at remote sites. The second drawback is that it does not ensure certain session consistency guarantees such as read-your-writes. Third, the PSI model does not support the partial replication model. We develop a transaction model that addresses all these issues and provide SI-based transactions with causal consistency for geo-replicated systems which use full or partial replication scheme.

There are additional issues that arise when supporting partial replication. In partial replication, since a database site may not contain all the data items, certain transactions may need to access data items from some other remote site. When executing such multi-site transactions, we must ensure that the transaction observes a consistent snapshot. Such a snapshot must be consistent with atomicity and isolation properties of Snapshot Isolation as well the properties of causal consistency. We discuss these issues in detail in Chapter 4.

2.4 Research Contributions

The primary contribution of this thesis is the development of techniques to support transaction with appropriate consistency models for NoSQL systems. This greatly extends the usability of the NoSQL systems to applications that require stronger consistency guarantees than those provided currently by the NoSQL systems. Another main contribution of this work is that it provides a spectrum of consistency models or guarantees based on the snapshot isolation model. The consistency models we provide include basic snapshot isolation, serializable snapshot isolation, and causal snapshot isolation. An application can chose the right consistency model based on its requirement and operation environment.

Specifically, the research work presented in this thesis has following contributions.
• **Cluster-based NoSQL systems:**

  • We introduce and investigate the notion of decoupling transaction management from storage system. The importance of decoupled transaction management is two-fold. First, since the transaction management functions are decoupled from storage system, the storage layer can be independently scaled without affecting performance or scalability. Second, this model allows to provide transaction support on top of an existing NoSQL system without needing to modify the implementation of the underlying system. We also present and evaluate a fault-tolerance model based on cooperative recovery in the above decoupled model.

  • We develop scalable techniques for providing serializability in SI-based transactions. We build upon the concepts developed for serializable snapshot isolation in relational database and investigate their applicability for NoSQL systems. Based on this, we implement and evaluate two approaches for ensuring serializability in snapshot isolation model.

  • Based on the decoupled transaction management model, we investigate two system architectures for supporting SI-based transactions - one based on pure decentralized approach and the other based on a hybrid approach which uses dedicated service for conflict detection. We implement both the architectures and perform their comparative evaluation for scalability.

• **Geo-replicated NoSQL systems:**

  • We develop a form of Snapshot Isolation model that can be efficiently supported in geo-replicated systems, i.e. a model which is based on asynchronous replication and does not require global sequencer. This model, called Causal Snapshot Isolation (CSI), provides SI-based transactions with causal consistency over geo-replicated data.

  • We extend the above model to support transactions on partially replicated databases. We present an efficient transaction management protocol that addresses the unique
issues raised in supporting snapshot isolation and causal consistency in partially replicated systems.

- In the CSI framework, we develop techniques to support session consistency guarantees, such as read-your-writes, monotonic-reads, or write-follows-read consistency.

The above contributions are detailed in Chapter 3 and Chapter 4.
Chapter 3

Transaction Management in Cluster-based Systems

In this chapter, we address the problem of providing scalable transaction support for cluster-based cloud datacenter environments and present the techniques we developed towards addressing this problem. In Chapter 2, we discussed the conceptual design approach based on the decoupled transaction management and the rationale behind it. We first describe below the transaction management framework based on this approach. We discuss the various design issues that arise in implementing this approach and the design choices we made in addressing them. Based on this framework, we first discuss the implementation of the fully decentralized system architecture model. In describing the implementation of this model, we first discuss the transaction management protocol that supports the basic snapshot isolation model and cooperative recovery. We then discuss how to extend this protocol to provide serializable transactions. Next, we discuss the hybrid service-based approach and present a design for the conflict detection service and its replication. Finally, we present the detailed evaluations of the presented approaches and techniques.
3.1 Transaction Management Framework

Implementing SI based transactions requires mechanisms for performing the following actions: (1) reading from a consistent committed snapshot; (2) allocating commit timestamps using a global sequencer for ordering of transactions; (3) detecting write-write conflicts among concurrent transactions; and (4) committing the updates atomically and making them durable. Additionally, to ensure serializability we also need to detect or prevent serialization anomalies as discussed above.

In our approach of decoupled and decentralized transaction management, the transaction management functions described above are decoupled from the NoSQL data storage systems and performed by the application processes themselves in decentralized manner. In rest of the chapter, we refer to the NoSQL data storage system used by the application simply as the storage system. The transaction management metadata required for performing these functions is also stored in the storage system. This is to ensure the reliability of this metadata and scalability of transaction management functions which require concurrent access to this metadata.

A transaction execution goes through a series of phases as shown in Figure 3.1. In the active phase, it performs read/write operations on data items. The subsequent phases are part of the commit protocol of the transaction. For scalability, our goal is to design the commit protocol such that it can be executed in highly concurrent manner by the application processes. We also want to ensure that after the commit timestamp is issued to a transaction, the time required for commit be bounded, since a long commit phase of the transaction can potentially block the progress of other conflicting transactions with higher timestamps. Thus, our goal is to perform as many commit protocol phases as possible before acquiring the commit timestamp. We discuss below the various issues that arise in utilizing this approach.

![Figure 3.1: Transaction Protocol Phases](image-url)
3.1.1 Timestamps Management

In the decentralized model, the steps in the commit protocol are executed concurrently by the application processes. Because these steps cannot be performed as a single atomic action, a number of design issues arise as discussed below. There can be situations where several transactions have acquired commit timestamps but their commitment status is not yet known. We also need to make sure that even if a transaction has made its update to the storage system, these updates should not be made visible to other transactions until the transaction is committed. Therefore, we need to maintain two timestamp counters: GTS (global timestamp) which is the latest commit timestamp assigned to a transaction, and STS (stable timestamp), which is the largest timestamp such that all transactions with commit timestamp up to this value are either committed or aborted and all the updates of the committed transactions are written to the storage system. An example shown in Figure 3.2 illustrates the notion of GTS and STS. In this example, STS is advanced only up to sequence number 16 because the commit status of all the transactions up to sequence number 16 is known, however, the commit status of the transaction with sequence number 17 is not yet known. When a new transaction is started, it uses the current STS value as its snapshot timestamp.

In implementing the timestamp management, we first experimented with using the storage system itself to store these counter values. However, we found this approach to be slow, and therefore we use a dedicated service for maintaining these counter values. We refer to this service as TimestampService. Along with maintaining the GTS and STS counter values, the TimestampService is also responsible for assigning the transaction-ids to transactions.

![Figure 3.2: STS and GTS Counters](image-url)
3.1.2 Eager vs Lazy Update Model

An important design issue that arises is when should a transaction write its updates to the storage system. We find two distinct approaches with different performance tradeoffs as discussed below. We characterize them as **eager** and **lazy** update models. In the eager update model, a transaction writes its updates to the storage system during its *active* phase, before acquiring its commit timestamp, whereas in the lazy approach all writes are performed after acquiring the commit timestamp.

In the lazy update approach, the time for executing the commit protocol can become arbitrarily long depending on the size of the data-items to be written. A long commit phase of a transaction would potentially delay the commit decisions of other concurrent and conflicting transactions that have higher commit timestamps. This may affect transaction throughput and system scalability, but it has the advantage that the writes are performed only when the transaction is certain to commit.

In the eager update approach the data is written to the storage system during the active phase, i.e. prior to the commit protocol execution, thereby reducing the execution time for the commit protocol. Also, the transactions can perform their writes overlapped with computation during the *active* phase. The eager update scheme is attractive because its commit protocol execution time does not significantly depend on the size of the write-set of the transaction. Also, it facilitates the roll-forward of a transaction that fails during its commit, since its updates would be already present in the storage system. Due to these advantages we choose to adopt the eager update model instead of the lazy update model.

Implementing the eager update model requires maintaining uncommitted data versions in the storage. For such data versions, we cannot use the transaction’s commit timestamp as the version number because it is not known during the *active* phase. Therefore, in the commit protocol these data versions need to be mapped to the transaction’s commit timestamp. Moreover, ensuring the isolation property requires that such uncommitted versions should not be visible until the transaction commits.
3.1.3 Transaction Validation

The SI model requires checking for write-write conflicts among concurrent transactions. This requires a mechanism to detect such conflicts and a method to resolve conflicts by allowing only one of the conflicting transactions to commit. When two or more concurrent transactions conflict, there are two approaches to decide which transaction should be allowed to commit. The first approach is called *first-committer-wins (FCW)* [28], in which the transaction with the smallest commit timestamp is allowed to commit. In this approach, conflict checking can only be performed by a transaction after acquiring its commit timestamp. This enforces a sequential ordering on conflict checking based on the commit timestamps. This would force a younger transaction to wait for the progress of all the older transactions, thereby limiting concurrency. In contrast, in the second approach, which is called *first-updater-wins (FUW)* [20], conflict detection is performed by acquiring locks on write-set items and in case of conflicting transactions the one that acquires the locks first is allowed to commit. The FUW approach appears more desirable because the conflict detection and resolution can be performed before acquiring the commit timestamp, thereby reducing any sequential ordering based on commit timestamps and reducing the time required for executing the commit protocol. Therefore, we chose to adopt the FUW approach for conflict detection.

In the fully decentralized model, the lock information is stored in the storage system. Application processes running transactions concurrently make requests to acquire locks in the decentralized manner. The locks are acquired atomically using the row-level transactions/atomic operations. In the service-based approach, the validation function is performed by a dedicated conflict detection service.

3.1.4 Cooperative Recovery

There are two problems that arise due to transaction failures. A failed transaction can block progress of other conflicting transactions. A failure of a transaction after acquiring commit timestamp stalls advancement of the STS counter, thereby forcing the new transactions to use old snapshot time, which may likely result in greater aborts due to write-write conflicts. Thus, an appropriate timeout mechanism is needed to detect stalled or failed transactions and initiate their
The cooperative recovery actions for a failed transaction are triggered in two situations: (1) a conflicting transaction is waiting for the commit of a failed transaction, and (2) the STS advancement has stalled due to a failed transaction that has acquired a commit timestamp. The recovery actions in the first situation are performed by any of the conflicting transactions, whereas the failures of the second kind are detected and recovery actions are performed by any application level process or by a dedicated system level process. If a transaction fails before acquiring a commit timestamp, then it is aborted, otherwise the transaction is committed and rolled-forward to complete its commit protocol.

3.2 Decentralized Model for Basic Snapshot Isolation

We first describe the decentralized model for basic snapshot isolation and then discuss how to extend it to ensure serializability. We describe below the metadata that needs to be maintained in the storage system for transaction management. We then describe the various steps in transaction management protocol performed by the application processes.

3.2.1 Storage System Requirements

We first identify the features of the storage system that are required for realizing the transaction management mechanisms presented here. The storage system should provide support for tables and multiple columns per data item (row), and primitives for managing multiple versions of data items with application-defined timestamps. It should provide strong consistency for updates [54], i.e. when a data item is updated, any subsequent reads should see the updated value. Moreover, for the decentralized architecture, we require mechanisms for performing row-level transactions involving any number of columns. Our implementation is based on HBase [4], which meets these requirements.

3.2.2 Transaction Data Management Model

For each transaction, we maintain in the storage system the following information: transaction-id (tid), snapshot timestamp (TSₘ), commit timestamps TSₖ, write-set information, and current
status. This information is maintained in a table named *TransactionTable* in the storage system, as shown in Figure 3.3. In this table, *tid* is the row-key of the table and other items are maintained as columns. The column ‘out-edges’ is used to record information related to outgoing dependency edges, which is required only in the cycle detection approach. In order to ensure that the *TransactionTable* does not become the bottleneck, we set the table configuration to partition it across all the HBase servers. The data distribution scheme for HBase is based on sequential range partitioning. Therefore, if we generate transaction ids sequentially it creates a load balancing problem since all the rows in *TransactionTable* corresponding the currently running transactions will be stored only at one or few HBase servers. Therefore, to avoid this problem we generate transaction ids randomly.

For each application data table, hereby referred as *StorageTable*, we maintain the information related to the committed versions of application data items and lock information, as shown in Figure 3.4. An application may have multiple such storage tables. Since we adopt the eager update model, uncommitted versions of data items also need to be maintained in the storage system. A transaction writes a new version of a data item with its *tid* as the version timestamp. These version timestamps then need to be mapped to the transaction commit timestamp *TSc* when transaction commits. This mapping is stored by writing *tid* in a column named *committed-version* with version timestamp as *TSc*. The column ‘wlock’ in the *StorageTable* is used to detect write-write conflicts, whereas columns ‘rlock’, ‘read-ts’, and ‘readers’ are used in detecting read-write conflicts for serializability, as discussed in the next section.

<table>
<thead>
<tr>
<th>Row Key (Trans ID)</th>
<th>Snapshot Timestamp <em>TS</em></th>
<th>Commit Timestamp <em>TSc</em></th>
<th>Write–Set</th>
<th>Out–Edges</th>
<th>Status (Trans State)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tid1</td>
<td>TS = 100</td>
<td>TS = 150</td>
<td>List of Item IDs</td>
<td>Outgoing Dependency Edges</td>
<td>Active, Validation, Commit–Incomplete, Committed, Aborted</td>
</tr>
<tr>
<td>tid2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.3: TransactionTable Structure
### 3.2.3 Transaction Management Protocol for Basic SI Model

A transaction $T_i$ begins with the execution of the *start* phase protocol shown in Algorithm 1. It obtains its transaction-id ($tid$) and snapshot timestamp ($T_{S_s}$) from *TimestampService*. It then inserts in the *TransactionTable* an entry: $<tid, T_{S_s}, status = active>$ and proceeds to the active phase. For a write operation on an item (specified by row and column keys), following the eager update model, the transaction creates a new version in the *StorageTable* using $tid$ as the version timestamp. The transaction also maintains its own writes in a local buffer to support *read-your-own-writes* consistency. A read operation for the data items not contained in the write-set is performed by first obtaining, for that data item, the latest version of the committed-version column in the range $[0, T_{S_s}]$. This gives the $tid$ of the transaction that wrote the latest version of the data item according to the transaction’s snapshot. The transaction then reads data specific columns using this $tid$ as the version timestamp.

A transaction executes the commit protocol as shown in Algorithm 2. At the start of each phase in the commit protocol it updates its status in the *TransactionTable* to indicate its progress. All status change operations are performed atomically and conditionally, i.e. permitting only the state transitions shown in Figure 3.1. The transaction first updates its status in the *TransactionTable to validation* and records its write-set information, i.e. only the item-identifiers (row keys) for items in its write-set. This information is recorded for facilitating the roll-forward of a failed transaction during its recovery. The transaction performs conflict checking by attempting to acquire write locks on items in its write-set as described below. If a committed newer version of the data item is already present, then it aborts immediately. If
some transaction $T_j$ has already acquired a write lock on the item, then $T_i$ aborts if $tid_j < tid_i$, else it waits for $T_j$ to either commit or abort. This wait/die scheme is used to avoid deadlocks and livelocks. The conflict checking operations for a single item, shown by lines 7-12 in the algorithm, are performed as a single atomic action using the row level transaction feature provided by HBase. On acquiring all locks, the transaction proceeds to the commit-incomplete phase.

**Algorithm 1** Execution Phase for transaction $T_i$

**Start Phase:**

- $tid_i$ $\leftarrow$ get a unique tid from TimestampService
- $TS_i^s$ $\leftarrow$ get current STS value from TimestampService
- insert $tid, TS_i^s$ information in TransactionTable.

**Active Phase:**

*Read item:* /* item is a row-key */

- $tid_R$ $\leftarrow$ read value of the latest version in the range $[0, TS_i^s]$ of the “committed version” column for item
- read item data with version $tid_R$
- add item to the read-set of $T_i$

*Write item:*

- write item to StorageTable with version timestamp = $tid_i$
- add item to the write-set of $T_i$

Once $T_i$ updates its status to commit-incomplete, any failure after that point would result in its roll-forward. The transaction now inserts the $ts \rightarrow tid$ mappings in the committed-version column in the StorageTable for the items in its write-set and changes its status to commit-complete. At this point the transaction is committed. It then notifies its completion to TimestampService and provides its commit timestamp $TS_i^c$ to advance the STS counter. The updates made by $T_i$ become visible to any subsequent transaction, after the STS counter is advanced up to $TS_i^c$. If the transaction is aborted, then it releases all the acquired locks and deletes the versions it has created.
Algorithm 2 Commit protocol executed by transaction $T_i$ for Basic SI model

Validation phase:

if $status = \text{active}$ then
    $status \leftarrow \text{validation}$

insert write-set information in $TransactionTable$

for all $item \in \text{write-set of } T_i$ do

    [ begin row level transaction:
    
    if any committed newer version for $item$ is created then abort
    
    if $item$ is locked then
        if lock-holder’s tid < $tid_i$, then abort else wait
    
    else
        acquire lock on $item$ by writing $tid_i$ in lock column

    :end row level transaction ]

Commit-Incomplete phase:

if $status = \text{validation}$ then
    $status \leftarrow \text{commit-incomplete}$

else abort

$TS^i_c \leftarrow$ get commit timestamp from $TimestampService$

for all $item \in \text{write-set of } T_i$ do

    insert $TS^i_c \rightarrow tid_i$ mapping in the $StorageTable$ and release lock on $item$

$status \leftarrow \text{commit-complete}$

notify completion and provide $TS^i_c$ to $TimestampService$ to advance $STS$

Abort phase:

for all $item \in \text{write-set of } T_i$ do

    if $T_i$ has acquired lock on $item$, then release the lock.

    delete the temporary version created for $item$ by $T_i$
3.2.4 Cooperative Recovery Protocol

When a transaction $T_i$ is waiting for the resolution of the commit status of some other transaction $T_j$, it periodically checks $T_j$’s progress. If the status of $T_j$ is not changed within a specific timeout value, $T_i$ suspects $T_j$ has failed. If $T_j$ has reached commit-incomplete phase, then $T_i$ performs roll-forward of $T_j$ by completing the commit-incomplete phase of $T_j$ using the write-set information recorded by $T_j$. Otherwise, $T_i$ marks $T_j$ as aborted, acquires the conflicting lock, and proceeds further with the next step in its own commit protocol. In this case, the cleanup actions, such as releasing other locks held by the aborted transaction and deletion of temporary versions created by the transactions, can be performed lazily if it does not block any other transaction. The cooperative recovery actions are also triggered when the $STS$ counter cannot be advanced because of a gap created due to a failed transaction. In this case, the recovery is triggered if the gap between $STS$ and $GTS$ exceeds beyond some limit. These recovery actions are triggered by the TimestampService itself based on the gap between $STS$ and $GTS$.

In the above mechanism, setting the proper timeout value is crucial. Setting a high timeout value will cause delays in detecting failures and thus potentially blocking conflicting transactions for a long time. When a transaction fails after acquiring commit timestamp, its timely recovery is crucial since it can block the advancement of $STS$. On the other hand, setting a low timeout value is also not desirable, since it can cause aborts of transactions that have not actually failed. We refer to these as false aborts. The appropriate timeout value depends on the average time taken by a transaction to complete its commit protocol. In Section 3.6, we present detailed evaluation of this aspect.

3.3 Decentralized Model for Serializable SI Transactions

We now describe how the decentralized model for the basic snapshot isolation is extended to support serializable transaction execution using the cycle prevention and cycle detection approaches discussed in Section 2.1.
### 3.3.1 Implementation of the Cycle Prevention Approach

The cycle prevention approach aborts a transaction when an anti-dependency among two concurrent transactions is observed. This prevents a transaction from becoming a pivot. One way of doing this is to record for each item version the *tids* of the transactions that read that version and track the *read-write* dependencies. However, this can be expensive as we need to maintain a list of *tids* per item and detect anti-dependencies for all such transactions. To avoid this, we detect the *read-write* conflicts using a locking approach. During the validation phase, a transaction acquires a *read lock* for each item in its read-set. Read locks on an item are acquired in shared mode. A transaction acquires (releases) a read lock by incrementing (decrementing) the value in a column named `rlock` in the `StorageTable`.

The commit protocol algorithm for the cycle-prevention approach is presented in Algorithm 3. An anti-dependency between two concurrent transactions can be detected either by the writer transaction or the reader transaction. We first describe how a writer transaction can detect a *read-write* conflict with any other concurrent reader transaction. During the validation phase, a writer transaction checks for the presence of a read lock for an item in its write-set at the time of attempting to acquire a write lock on that item. The transaction is aborted if the item is already read locked. Note that we need to detect *read-write* conflicts only among concurrent transactions to detect anti-dependencies. This raises an issue that a concurrent writer may miss detecting a *read-write* conflict if it attempts to acquire a write lock after the conflicting reader transaction has committed and its read lock has been released. To avoid this problem, a reader transaction records its commit timestamp, in a column named ‘read-ts’ in the `StorageTable`, while releasing a read lock acquired on an item. A writer checks whether the timestamp value written in the ‘read-ts’ column is greater than its snapshot timestamp, which indicates that the writer is concurrent with a committed reader transaction. A reader transaction checks for the presence of a write lock or a newer committed version for an item in its read-set to detect *read-write* conflicts. Otherwise, it acquires a read lock on the item.
Algorithm 3 Commit protocol for cycle prevention approach

Validation phase:

for all item ∈ write-set of $T_i$ do

| begin row-level transaction: |
read the ‘committed version’, ‘wlock’, ‘rlock’, and ‘read-ts’ columns for item
if any committed newer version is present, then abort
else if item is already locked in read or write mode, then abort
else if ‘read-ts’ value is greater than $TS_i^c$, then abort.
else acquire write lock on item
:end row-level transaction |

for all item ∈ read-set of $T_i$ do

| begin row-level transaction: |
read the ‘committed version’ and ‘wlock’ columns for item
if any committed newer version is created, then abort
if item is already locked in write mode, then abort.
else acquire read lock by incrementing ‘rlock’ column for item.
:end row-level transaction |

execute commit-incomplete phase shown in Algorithm 2

for all item ∈ read-set of $T_i$ do

| begin row-level transaction: |
release read lock on item by decrementing ‘rlock’ column
if read-ts < $TS_i^r$ then
    read-ts ← $TS_i^r$
:end row-level transaction |

status ← commit-complete
notify completion and provide $TS_i^c$ to TimestampService to advance $STS$

During the commit-incomplete phase, $T_i$ releases the acquired read locks and records its commit timestamps in the ‘read-ts’ column for the items in its read-set. Since there can be
more than one reader transactions for a particular data item version, it is possible that some
transaction has already recorded a value in the ‘read-ts’ column. In this case, $T_i$ updates the
currently recorded value only if it is less than $TS_i^r$. The rationale behind the logic for updating
the ‘read-ts’ value is as follows. For committed transactions $T_1, T_2, ..., T_n$ that have read a
particular data item version, the ‘read-ts’ column value for that item version would contain the
commit timestamp of transaction $T_k$ ($k \leq n$), such that $T_k$ is the transaction with the largest
commit timestamp in this set of transactions. An uncommitted writer transaction $T_j$ that is
concurrent with any transaction in the set $T_1, T_2, ..., T_n$ must also be concurrent with $T_k$ i.e.
$TS_j^w < TS_k^c$, since $T_k$ has the largest commit timestamp. Thus $T_j$ will detect the read-write
contlict by observing that the ‘read-ts’ value is larger than its snapshot timestamp.

3.3.2 Implementation of the Cycle Detection Approach

The cycle detection approach requires tracking all dependencies among transactions, i.e. anti-
dependencies (both incoming and outgoing) among concurrent transactions, and write-read and
write-write dependencies among non-concurrent transactions. We maintain this information in
the form of a dependency serialization graph (DSG) [20], in the storage system. Since an active
transaction may form dependencies with a certain committed transaction, we need to retain
information about such transactions in the DSG.

Detecting Dependencies

For detecting dependencies, we record in $StorageTable$ (in a column named ‘readers’), for each
version of an item, a list of transaction-ids that have read that item version. Moreover, for each
transaction $T_i$, we maintain its outgoing dependency edges as the list of $tids$ of the transactions
for which $T_i$ has an outgoing dependency edge. This information is recorded in the ‘out-edges’
column in the $TransactionTable$, and it captures the DSG structure. The dependencies are
detected and recorded as discussed below.

For detecting dependencies, we include an additional phase called $DSGupdate$ in the transac-
tion protocol, which is performed before the validation phase. In the $DSGupdate$ phase, along
with the basic write-write conflict check using the locking technique discussed in Section 3.2, a
transaction also detects dependencies with other transactions based on its read-write sets. To find \(wr\) dependencies, the transaction finds, for every item version in its read-set, the the \(tid\) of the transaction which wrote that version. Similarly, to derive outgoing \(rw\) anti-dependencies, it finds the \(tid\) of the transaction which either wrote the immediately following version for that data item or is holding lock on that data item, if any. For every item in the write-set of the transaction, it finds the \(tids\) of the reader(s) and writer of the immediately preceding version. This gives the incoming \(rw\) edge(s) and the \(ww\) edge. The dependency edges are then inserted in the TransactionTable. Since our purpose is to find the directed cycles irrespective of the type of edges, we only insert an outgoing edge \(T_1 \rightarrow T_2\), if there is at least one edge of type \(rw\), \(wr\) or \(ww\) from \(T_1\) to \(T_2\).

### Checking for Dependency Cycles

In the validation phase, the transaction checks for a dependency cycle involving itself, by traversing the outgoing edges in the DSG in depth-first-search manner, starting from itself. In the search, it considers only the transactions which are in validation, commit-complete, or commit-incomplete phase. Since the transactions which are in Validation or later phases must have already inserted their dependency edges, there is no possibility that a transaction would miss any dependency edge. It ignores any transaction with larger \(TS_c\), encountered during the search. If it detects a cycle and all the transactions involved in the cycle are already committed, then it aborts. If the transaction detects a cycle with one or more transactions still in the validation phase then it waits for their status to resolve. If a transaction does not detect a cycle, or if any transaction involved in the cycle aborts, then it proceeds further. The cycle checking is performed concurrently and in non-blocking manner by the transactions. Due to the ordering based on \(TS_c\) when two concurrent transactions are involved in the same cycle, only one of them would abort.

### Pruning of DSG

A challenge in this approach is to maintain the dependency graph as small as possible by frequently pruning to remove those committed transactions that can never lead to any cycle in
the future. In order to prune the dependency graph, we remove the unnecessary transactions using the following rule: a committed transaction is removed if it is - (1) not reachable from any currently active transaction and (2) not concurrent with any active transaction. Such an unreachable transaction can not be part of any future cycle for following reasons. Since the transaction is not concurrent with any active transaction, the only new dependencies that can arise for such a transaction in future are of outgoing \(ww\) and \(wr\) types. Since such a transaction does not have any incoming edge, and the only new dependencies edges that can be formed are outgoing edges it can not become part of any future cycle. Thus, such a transaction can be safely removed from the DSG.

3.4 Service-based Model

We observed that the decentralized approach induces performance overheads due to the additional read and write requests to the storage system for acquiring and releasing locks. Therefore, we evaluated an alternative approach of using a dedicated service for conflict detection. In the service-based approach, the conflict detection service maintains in its primary memory the information required for conflict detection. A transaction in its commit phase sends its read/write sets information and snapshot timestamp value to the conflict detection service. We designed this service to support conflict detection for the basic-SI model and the cycle prevention/detection approaches for serializable transaction. Based on the particular conflict detection approach, the service checks if the requesting transaction conflicts with any previously committed transaction or not. If no conflict is found, the service obtains a commit timestamp for the transaction and sends a ‘commit’ response to the transaction process along with the commit timestamp, otherwise it sends ‘abort’. Before sending the response, the service updates the transaction’s commit status in in the TransactionTable in the storage system.

The transaction commit phase executed using this approach is presented below in Algorithm 4. Note that this dedicated service is used only for the purpose of conflict detection and not for the entire transaction management, as done in [55, 34]. The other transaction management functions, such as getting the appropriate snapshot, maintaining uncommitted versions, and ensuring the atomicity and durability of updates when a transaction commits are performed
by the application level processes. For scalability and availability, we designed this service as a replicated service as described below.

**Algorithm 4** Commit algorithm executed by $T_i$ in Service-Based approach

1. update status to *Validation* in *TransactionTable* provided status = *Active*
2. insert write-set information in *TransactionTable*
3. send request to conflict detection service with write-set information and $TS_i^j$
   
   **if** response = *commit* **then**
   
   $TS_i^c \leftarrow$ commit timestamp returned by the service
   
   execute CommitIncomplete phase as in Algorithm 2 except for step 2

   **else**

   execute Abort phase as in Algorithm 2

### 3.4.1 Replicated Service Design

The conflict detection service is implemented as a group of processes. The data item space is partitioned across the replicas using a hashing based partitioning scheme. Thus, a service replica is responsible for detecting conflicts for a set of data items. A service replica stores, for each data item it is responsible for, the information necessary to detect conflicts for that data item. For conflict detection, each replica maintains an in-memory table called as *ConflictTable*, which contains following information for each data item: (1) commit timestamp of the latest committed transaction that has modified the item (write-ts), (2) commit timestamp of the latest committed transaction that has read the item (read-ts), (3) lock-mode (read-mode, write-mode, or unlocked), (4) lock owner : writer-tid (in case of write lock) and list of reader-tids (in case of read lock), (5) list of tids waiting for lock (i.e. pending lock requests) Table 3.1 shows an example of the information maintained in the conflict table. In this example, transaction with tid 989 has currently acquired a write-lock on item x and lock request for tid 7771 is pending. Item y is locked in read mode by transaction with tids 35 and 7771 and no pending lock request. Item z is currently unlocked, i.e. there is no transaction with pending validation request that has read or modified item z.

The validation for a transaction is performed by replica(s) responsible for the data items
<table>
<thead>
<tr>
<th>item-id</th>
<th>write-ts</th>
<th>read-ts</th>
<th>lock-mode</th>
<th>writer/readers</th>
<th>waiting-tids</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>80</td>
<td>100</td>
<td>write</td>
<td>989</td>
<td>7771</td>
</tr>
<tr>
<td>y</td>
<td>85</td>
<td>95</td>
<td>read</td>
<td>&lt;35, 7771&gt;</td>
<td>-</td>
</tr>
<tr>
<td>z</td>
<td>50</td>
<td>75</td>
<td>unlocked</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3.1: ConflictTable Example

in the transaction’s read/write sets. When a transaction’s read/write sets span across more than one replica, the validation is performed by coordinating with other replicas, as described below. A client, i.e. the application process executing a transaction, contacts any of the service replicas to validate the transaction by providing its read-write set and snapshot timestamp information. The contacted service replica then acts as the coordinator in executing the protocol for transaction validation. The coordinator determines the replicas, called participants, that are responsible for the data items in the transaction’s read/write sets. It is possible that the coordinator itself is one of the participants. It then performs a two-phase coordination protocol with the participants, as described below.

In the first phase, the coordinator sends acquire-locks request to all the participants. Each participant then checks read-write and write-write conflicts for the items it is responsible for in the following way. For a write-set item, if the write-ts value is greater than the transaction’s snapshot timestamp then it indicates a write-write conflict. Similarly, if the read-ts value is greater than the snapshot timestamp then it indicates a read-write conflict. For a read-set item, there is a read-write conflict if the write-ts value is greater than transaction’s snapshot timestamp. If any conflict is found, the participant sends ‘failed’ response to the coordinator. If there are no conflicts, then an attempt is made to acquire read/write locks on the items.

We use a deadlock-prevention scheme similar to the one used in [35]. When a transaction $T_i$ is blocked by transaction $T_j$ due to a conflicting lock on an item $x$, if $tid_i$ is greater than $tid_j$ then $T_i$ waits for the commit decision of $T_j$. If $tid_i$ is less than $tid_j$, then an inquire message is sent to the coordinator of $T_j$ to check if $T_j$ has acquired all locks. If not then the lock held by $T_j$ on $x$ is released and given to $T_i$. To simplify the protocol in case of read-write conflicts, if a transaction requests a write lock on an item which is currently held in read-mode the writer transaction is aborted. If the participant acquires all the locks, it sends a ‘success’ response to
the coordinator.

In the second phase, if the coordinator has received a ‘success’ response from all the participants, then the transaction is committed, otherwise it is aborted. The status of the transaction is updated in the TransactionTable. In case of transaction commit the coordinator obtains a commit timestamp from the TimestampService and sends a ‘commit’ message to all participants along with the commit timestamp. Each participant then updates the write-ts and read-ts values for the corresponding items and releases the locks. Any conflicting transaction waiting for the commit decision of this transaction is aborted. In case of abort, each participant releases the locks acquired by the aborted transaction.

It is important to ensure that the size of the ConflictTable is small enough to efficiently store it in memory, especially when the data item space is significantly large. This requires garbage-collection of the information stored in the ConflictTable. Note that, the conflicts will occur only among concurrent transactions. For garbage collection, we use a timestamp counter, called as fossil counter, which is the smallest snapshot timestamp among currently active transactions. Any committed transaction with commit timestamp lower than the fossil counter will not cause any conflict. Thus, in ConflictTable we need to keep the information only for the data items which have been modified by transactions with commit timestamps larger than fossil counter.

<table>
<thead>
<tr>
<th>request-id</th>
<th>transaction-id</th>
<th>read/write sets</th>
<th>lock status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>write: {x}</td>
<td>{x⇒pending}</td>
</tr>
<tr>
<td>Table maintained by participant1, responsible for x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>request-id</th>
<th>transaction-id</th>
<th>read/write sets</th>
<th>lock status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>read: {y}</td>
<td>{y⇒locked}</td>
</tr>
<tr>
<td>Table maintained by participant2, responsible for y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: ParticipantTable Example

<table>
<thead>
<tr>
<th>request-id</th>
<th>transaction-id</th>
<th>read/write sets</th>
<th>lock status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>read: {y} write: {x}</td>
<td>{x⇒pending, y⇒locked}</td>
</tr>
</tbody>
</table>

Table 3.3: CoordinatorTable Example
For tracking the validation requests, each replica maintains two in-memory tables: a ParticipantTable to store information related to the validation requests for which the replica is a participant, and CoordinatorTable to store information for the validation requests for which the replica is the coordinator. The ParticipantTable maintains, for each validation request, the transaction-id, the part of the transaction’s read/write sets pertaining to this participant, and lock status of each item in this set. The CoordinatorTable contains, for each request, the participant replicas, the read/write sets of the transaction and lock status of each item in the set, and responses received from different participants. Table 3.2 and 3.3 show an example of the information maintained in these two tables. In this example, we show the information maintained for the validation request for transaction with tid 7771 which reads y and writes x. The coordinator maintains for this transaction the lock status information of both x and y, whereas participant1 which is responsible for item x maintain this information only for item x and similarly participant2 maintains the lock status information only for item y.

### 3.4.2 Service Fault Tolerance

The failure-detection and the group membership management is performed by the service replicas in decentralized manner. Failure-detection is performed using a heart-beat based mechanism, and for group membership management we use the protocol we developed in [37]. When a replica crashes, a new replica is started which takes over the role of the failed replica. The new replica then performs the recovery protocol as described below.

*Recovering service state:* When a replica fails, there may be uncompleted validation requests for which the replica is either a coordinator or a participant. The new replica needs to recover the information maintained in the CoordinatorTable and ParticipantTable. This information is soft-state and can be recovered from other replicas. The new replica contacts all other existing replicas in the group and obtains information regarding the pending requests for which it was either a coordinator or a participant and the lock status for the items involved in these requests. The reconstruction of ConflictTable is done using the read/write sets information stored in the TransactionTable. However, scanning the entire TransactionTable can become expensive, and hence to reduce this overhead the ConflictTable is periodically checkpointed in stable storage.
Thus, only the transactions committed after the checkpoint start time need to be scanned.

Failure cases: Ensuring the correctness of the two-phase coordination protocol under replica crashes is crucial. With respect to a particular validation request, the crashed replica can either be a coordinator or a participant. In case of the coordinator crash, the client will timeout and retry the request by contacting the new replica or any other replica. There are three failure cases to consider: (1) failure before initiating the first phase, (2) failure before recording the commit status, and (3) failure after recording the commit status. In the first failure case, the client will timeout and retry the request, since none of the replicas would have any information for this request. In the second case, the new coordinator will resend the lock requests. It may happen that some locks have been already acquired, however, lock operations are idempotent, so resending the lock requests does not cause any inconsistencies. When failure occurs after recording the commit status, the new coordinator will first check the commit status and send commit or abort requests to participants accordingly. In case of the participant crash, the validation request cannot be processed until the recovery of the crashed participant is complete.

3.5 Scalability Evaluations

In our evaluations of the proposed approaches the focus was on evaluating the following aspects: (1) the scalability of different approaches under the scale-out model, (2) comparison of the service-based model and the decentralized model in terms of transaction throughput and scalability, (3) comparison of the basic SI and the transaction serializability approaches based on the cycle-prevention and the cycle-detection techniques, (4) transaction response times for various approaches, and (5) execution times of different protocol phases.

During the initial phase of our work, we performed a preliminary evaluation of the proposed approaches to determine which approaches are more scalable and which of these need to be investigated further on a large scale cluster. This evaluation was done using a testbed cluster of 40 nodes on which the number of cores on the cluster nodes varied from 2 to 8 cores, each with 2.3 GHz, and the memory capacity ranged from 4 GB to 8 GB. The final evaluations in the later phase were conducted using a much larger cluster provided by the Minnesota Supercomputing Institute (MSI). Each node in this cluster had 8 CPU cores with 2.8 GHz capacity, and 22 GB
main memory.

3.5.1 TPC-C Benchmark

We used TPC-C benchmark to perform evaluations under a realistic workload. The TPC-C benchmark models a typical order-entry application and is used as a benchmark for traditional online transaction processing (OLTP) applications. The database consists of 9 tables storing information about warehouses, inventory, customers, orders, etc. The transaction workload consists of different types of transactions. For example, it includes a transaction type called ‘NEW-ORDER’ which is used to place a new order for a customer and a transaction type called ‘PAYMENT’ which is used to make payment for an order.

The TPC-C specifications [15] provides detailed specifications for various parameters such as database size, cardinalities of different tables, frequency of different types of transactions, etc. Our implementation of the benchmark workload differs from TPC-C specifications in the following ways. Since our primary purpose is to measure the transaction throughput we did not emulate terminal I/O. Also, to generate more load, we did not induce wait times in between transactions. Since HBase does not support composite primary keys, we created the row-keys as concatenation of the specified primary keys. This eliminated the need of join operations, typically required in SQL-based implementation of TPC-C. Predicate reads were implemented using scan and filtering operations provided by HBase. Since the transactions specified in TPC-C benchmark do not create serialization anomalies under SI, as observed in [20], we implemented the modifications suggested in [11]. These modification introduce a new transaction type called ‘CREDIT-CHECK’ which introduces serialization anomalies, i.e. it can lead to pivot transactions, when a transaction of this type concurrently runs with another transaction of certain types (‘PAYMENT’ and ‘DELIVERY’ transaction type). In the benchmark roughly more than 90% of transactions are read-write transactions. In our experiments we observed that on average a TPC-C transaction performed 8 read operations and 6 write operations.
3.5.2 Preliminary Evaluations

During each experiment, the first phase involved loading data in HBase servers, which also ensured that the data was in the memory of HBase servers when the experiment started. Before starting the measurements, we ran the system for five minutes with initial transaction rate of about 1000 transactions per minute. The purpose of this was to ‘warm-up’ the TransactionTable partitions in HBase servers’ memory. The measurement period was set to 30 minutes, in which we gradually increased the transaction load to measure the maximum throughput. For different cluster sizes, we measured the maximum transaction throughput (in terms of committed transactions per minute (tpmC)) and response times. In our experiments, we used one timestamp server and for the service-based model we used one validation server process.

Figure 3.5 shows the maximum throughput achieved for different transaction management approaches for different cluster sizes. Since there is significant node heterogeneity in our testbed
cluster, we indicate the cluster size in terms of the number of cores instead of the number of nodes. This figure shows the throughput for basic-SI model to understand the cost of supporting serializability. We can observe from Figure 3.5 that scalability of throughput is achieved in both the service-based as well as the decentralized model. However, the service-based approach gives higher transaction throughput than the decentralized approach. As expected, the basic SI model achieves higher throughput compared to approaches for ensuring serializability. This is because of the overhead for performing additional checks required for ensuring serializability. The cycle-prevention approach provides higher throughput than the cycle-detection approach. This is because in the decentralized model the overhead of the cycle-detection approach is significant due to the overhead of maintaining dependency information in the storage system. We also compared the cycle-prevention and the cycle-detection approaches in the context of the service-based model. However, we did not observe any significant difference in the transaction throughput.

Figure 3.6 shows the average transaction response times for various approaches. As expected, the service-based approach gives smaller response times than other approaches. The cycle-detection approach has significant overhead. In the largest configuration, the average response time for the cycle-detection approach is more than double of the same for the cycle-prevention approach. Also, the cycle-detection approach does not scale well in terms of response times for large clusters. Therefore, we conclude that if serializability is required, it is better to use the cycle-prevention approach than the cycle-detection approach.

Figure 3.7: Execution time for different protocol phases
We also measured and compared the time taken to execute the various phases of the transaction protocol for different approaches. Figure 3.7 shows the average execution times for different phases. This data is shown for the evaluations conducted with the largest (96 cores) configuration. The validation phase for the cycle-prevention approach takes more time (approximately by a factor of two) than the validation phase for the basic SI approach. In the cycle-detection approach the $DSG_{update}$ phase induces a significant overhead.

The preliminary evaluations indicated that the service-based and the decentralized cycle-prevention approaches are scalable for supporting serializable transactions. Among these two approaches the service-based approach performs better. We found that the decentralized cycle-detection approach does not scale well.

### 3.5.3 Scalability Validations on a Large Cluster

Based on the above observations, we selected the service-based approach and decentralized cycle-prevention approach for further evaluations over a large scale cluster to validate their scalability. These evaluations were performed using the MSI cluster resources. Using this cluster, we measured maximum transaction throughput achieved for different cluster sizes. The results of these evaluations are presented in Figures 3.8 and 3.9.

Figure 3.8 presents the throughput scalability, and Figure 3.9 shows average response times for various cluster sizes. The largest cluster size used in these experiments corresponds to close to 100 nodes (800 cores). In these evaluations we enabled the synchronous logging option for HBase to ensure that the writes are durable even under crashes of HBase servers. The synchronous logging increases the latency of write operations hence, the response times are generally higher under this setting compared to the response times shown in Figure 3.6 where synchronous logging was disabled.

Both the preliminary evaluations and the evaluations over the MSI cluster demonstrate the horizontal scalability or the scale-out capability of the decentralized and service-based architecture models. The service-based architecture provides higher transaction throughput and lower response times compared to the decentralized model, confirming our earlier observations from the preliminary evaluations.
3.5.4 Scalability of Conflict Detection Service

We also evaluated the scalability of the replicated conflict detection service. In this evaluation, we were mainly interested in measuring the throughput of validation requests. For this purpose, we generated a synthetic workload as follows. A pool of clients generated validation requests for randomly selected read/write sets from an item space of 1 million items. For each request, the size of the read/write sets was randomly selected between 4 to 20 items, with half of the items being read items and half being write items. We measured the throughput as the number of requests handled by the service per second, irrespective of the commit/abort decision, since we are mainly interested in measuring the request handling capacity of the service. Figure 3.10 shows the saturation throughput of the service for different number of replicas. We can see that increasing the number of replicas provides sub-linear increase in throughput, for example, increasing replica size from 4 to 8 provides throughput increase by a factor of 1.35. An important
thing to note here is that the saturation throughput of the conflict detection service, even with a small number of replicas, is significantly higher than the overall transaction throughput of the system. For example, from Figures 3.10 and 3.8, we can see that the saturation throughput of the service with 8 replicas is approximately 24,000 requests per second whereas the saturation transaction throughput with 100 nodes is approximately 5,000 transactions per second. Thus, a small number of replicas for conflict detection service can suffice to handle the workload requirement of a large cluster.

![Figure 3.10: Scalability of the conflict detection service](image)

### 3.5.5 Impact of Transaction Size

Another aspect that we were interested in evaluating is the impact of transaction size, i.e. the number of reads and writes in a transaction, on various performance measures. To evaluate this impact, we created a custom benchmark as follows. We created a single table with 1 million items. The benchmark included three classes of transactions: small size transactions accessing 10 items each, medium size transactions accessing 100 items each, and large size transactions accessing 1000 items each. In all the three classes, half of the accessed items were read-set items and half were write-set items. The items to read and write were randomly selected based on uniform distribution. We performed separate evaluations for each class of transactions by generating transaction load for that class and measured the maximum throughput, average response times, and number of aborts for that transaction class. Table 3.4 shows the results of these evaluations. These evaluations were performed using a cluster of 24 nodes on the MSI.
platform and the decentralized model with cycle prevention approach. No failures were injected during these evaluations.

<table>
<thead>
<tr>
<th>Transaction Size</th>
<th>Max Throughput (transactions/min)</th>
<th>Avg. Response Time (sec)</th>
<th>Percentage of Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>72196</td>
<td>0.31</td>
<td>0.4 %</td>
</tr>
<tr>
<td>100</td>
<td>7088</td>
<td>1.72</td>
<td>17.7 %</td>
</tr>
<tr>
<td>1000</td>
<td>1008</td>
<td>7.2</td>
<td>69.1 %</td>
</tr>
</tbody>
</table>

Table 3.4: Impact of Transaction Size

From Table 3.4, we can observe that as we increase the transaction size the maximum throughput decreases and the average response time increases. This is expected since the maximum throughput and the response time are directly proportion to the number of read/write operations in a transaction. The percentage of aborts increase with the increase in transaction size. This is primarily because under a fixed database size with increase in transaction size the likelihood of transaction conflicts increases.

3.6 Fault Tolerance Evaluations

Our goal was to measure the performance of the cooperative recovery model. In these evaluations, our focus was on observing the following aspects: (1) impact of failure timeout values, (2) time taken to detect and recover failed transactions, and (3) impact of failures on abort rate and $STS$ advancement.

3.6.1 Experiment Setup

We performed these evaluations on a cluster of 24 nodes on the MSI platform. We induced a moderate transaction load of approximately 50000 transaction per minute, which is lower than the saturation load observed in Figure 3.8 for cluster size of 24 nodes. The injection of faults was performed as follows. A transaction randomly stalls during the commit protocol execution with certain probability called as failure probability. The failure probability is calculated based
on the desired failure rate. The failure is injected either in the validation phase or the commit-incomplete phase (after acquiring commit timestamp). We experimented with a setting of 50% failures in validation and 50% failures in the commit-incomplete phase as well as an extreme case with all failures in validation phase. For every injected failure, we measured the delay in detection of that failed transaction as well as time required to perform recovery actions. We also recorded information about the number of transactions that were wrongly suspected to be failed and aborted due to timeouts. We refer to such aborts as false aborts. One would expect that for smaller timeouts the percentage of such false aborts would be higher. The percentage of valid timeout-based aborts depends on the number of injected failures. We performed these experiments for failure rates of 0.1%, 1%, and 10% and timeout values of 50, 100, 300, and 500 milliseconds. In these evaluations, we measured following performance measures: (1) percentage of aborts due to concurrency conflicts such as read-write and write-write conflicts, referred to as self aborts, (2) percentage of valid timeout-based aborts and false aborts, (3) average time for failure detection and recovery, and (4) average gap between STS and GTS under failures.

### 3.6.2 Evaluation Results

<table>
<thead>
<tr>
<th>Failure Rate</th>
<th>Self Aborts</th>
<th>Valid Aborts</th>
<th>False Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1%</td>
<td>15</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>1%</td>
<td>10</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>10%</td>
<td>5</td>
<td>30</td>
<td>5</td>
</tr>
<tr>
<td>Timeout (ms)</td>
<td>50</td>
<td>100</td>
<td>300</td>
</tr>
</tbody>
</table>

![Figure 3.11: Abort Statistics for Different Failure Rates (Timeout values ranging from 50ms to 500ms)](image)

Figure 3.11 shows the abort statistics for various timeout values and failure rates. This data correspond to the setting of 50% failures in the validation and 50% failures in the commit-incomplete phase. We can see that, as expected, the percentage of false aborts increases as...
we decrease the timeout values. The percentage of valid timeout-based aborts depends on the failure rate. Note that, even though we decrease the timeout values from 500 ms to 100 ms, the percentage of total aborts increase only by approximately a factor of two (from 15% to 30%). This is because the transactions that do not conflict with any other transactions are unaffected by the timeout values. If a transaction does not conflict with any other concurrent transaction, it would not be aborted by any transaction irrespective of the timeout values.

The only problem that will arise due to failure of a non-conflicting transaction is the blocking of STS advancement if it has acquired the commit timestamp. However, in that case the transaction would be rolled-forward instead of aborting. We also performed this evaluation with the setting of all failures in the validation phase. We observed that under this setting also the false aborts increase with decrease in timeout values, confirming our earlier observation. Thus, the appropriate timeout value must be large enough so that the number of false aborts is kept minimum. This largely depends on the transaction response time. Therefore, the appropriate timeout value can be chosen by observing the transaction response times. One can also include autonomic mechanisms to set the timeout values by continually observing the response time values at runtime.

Figure 3.12: Average Time to detect failures in Validation phase

Figure 3.12 shows the data regarding average delays in detection of the failure of transactions that were failed in the validation phase. Figure 3.13 shows this data for transactions that were failed in the commit-incomplete phase. Since we were interested in measuring the delays in detecting valid failures, we measured this data only for the failures that were injected by
the failure injection mechanism and not for the transactions that were wrongly suspected to be failed. From Figures 3.12 and 3.13, we can see that the detection delays for transactions failed in the validation phase are significantly higher than that for the transaction failed in the commit-incomplete phase. This is expected because, failure of a transaction failed in the validation phase will only be detected when some other transaction encounters a read-write or write-write conflict with the failed transaction, whereas failure of a transaction failed in the commit-incomplete phase will be detected more promptly due to non-advancement of STS. We can also observe that the failure detection delay for transactions failed in the commit-incomplete phase is mainly dominated by the timeout value: as we decrease the timeout value the failure detection delay decreases. We observed that the time required to perform recovery actions is independent of whether the transaction failed in the validation phase or the commit-incomplete phase. The average time to perform recovery typically ranged between 55 ms to 90 ms.

We also measured the average gap between STS and GTS for various failure rates and timeout values. This data is shown in Figure 3.14. We can see that the average STS gap does not depend on the timeout values. However, as we increase the failure rate the average gap value typically increases due to more number transactions blocking the advancement of STS.
Figure 3.14: Average gap between STS and GTS for different failure rates and timeout values

3.7 Summary

In this chapter we addressed the problem of providing scalable support for ACID transactions on NoSQL systems in a cluster/datacenter environment. As the SI model does not ensure serializability guarantee, we discussed two approaches for making SI-based transactions serializable - the first approach based on preventing dependency cycles and other based on detecting dependency cycles. The primary contribution of this work is the concept of decoupled transaction management, which proposes to decouple the transaction management functions from the storage system. Based on this concept, we present two system architectures for transaction management - a fully decentralized architecture, where all transaction management functions are performed in decentralized manner by the application process and a hybrid model where the conflict detection functions are implemented by a dedicated service. We also presented a fault tolerance model based on cooperative recovery, in which there is no dedicated central component for failure detection and recovery and these functions are performed by the application processes themselves in a cooperative way. Through the experimental evaluations we demonstrate the scalability of our models and techniques. The experimental evaluations show that the hybrid architecture model performs better in terms of transaction throughput and response times. In regards to the approaches for ensuring serializability, we found that the approach based on cycle prevention performs better than the approach based on cycle detection.

The transaction model supported here provides strong transactional consistency with serializability guarantees. This requires that the underlying storage system provides strong data
consistency which is essential in realizing the transaction model described in this chapter. In the next chapter, we discuss the transaction management model for geo-replicated systems. Since in geo-replicated systems providing strong data consistency is not practical, our aim is to provide a weaker but useful consistency model.
Chapter 4

Transaction Management in Geo-replicated Systems

Data replication across different geographically distributed sites is needed for several reasons. Many large-scale web applications have geographically distributed client population. With geo-replication, clients in a particular geographic region can be directed to the servers located at a site close to that region. Geo-replication is also desired for applications that need to be disaster-tolerant. In this chapter, we address the problem of providing transaction support for data replicated across geographically distributed sites. In this regard, there are many issues related to scalability and performance that need to be addressed while applying the SI transaction model to a geographically distributed environment, as discussed below.

The first issue is the requirement of reading from the latest snapshot, which requires that a transaction’s updates are propagated synchronously to other sites. Synchronous replication, especially under wide-area environments, can limit the performance and scalability. Other researchers [50, 18] have recognized these limitations and have proposed solutions in which a transaction may read from an older but consistent snapshot. The second problem is ensuring the total ordering of transactions across sites. This would require a global sequencer mechanism which can be another scalability bottleneck. Some systems [32, 18, 27] have proposed mechanisms based on atomic broadcast for total ordering, typically for cluster-based replication in...
LAN environments. Under WAN environment, high network latencies between the sites can affect the performance of such protocols, and hence ensuring total ordering can be limiting.

### 4.1 Causal Snapshot Isolation Model

We present here a transaction model, called *Causally-coordinated Snapshot Isolation (CSI)*, which provides snapshot isolation (SI) [8] based transactions with causal consistency of data for asynchronously replicated databases. This model builds upon the approach presented in [50], but eliminates false causal dependencies. The CSI model does not enforce a total ordering of transactions across sites, since implementing it requires a global sequencer mechanism, which can become a scalability bottleneck and is not practical under wide-area settings. The CSI model provides total ordering of transactions only within a site, and across sites it provides causal ordering of transactions.

We define the causal ordering ($\prec$) between two transactions as follows. Transaction $T_i$ casually precedes transaction $T_j$ ($T_i \prec T_j$), or in other words $T_j$ is causally dependent on $T_i$, if $T_i$ and $T_j$ are not concurrent and if $T_j$ reads any of the updates made by $T_i$ or creates a newer version for any of the items modified by $T_i$. Also, this relationship is transitive, i.e. if $T_i \prec T_j$ and $T_j \prec T_k$, then $T_i \prec T_k$. The causal ordering defines a partial ordering over transactions.

The CSI model guarantees that a transaction observes a consistent snapshot which has the properties of *atomicity* and *causality* as defined below.

- **Atomicity**: If the transaction sees an update made by a transaction $T_i$ then all other updates of $T_i$ are also visible in its snapshot.

- **Causality**: The database state observed by the transaction is consistent with causal ordering of transactions, i.e. if the transaction sees effects of transaction $T_i$, then effects of all the transactions causally preceding $T_i$ are also visible in its snapshot.

Moreover, the CSI model ensures that when two or more concurrent transactions update a common data item, only one of them is allowed to commit. The snapshot observed by a transaction may not always reflect the latest versions of the accessed items, but it is guaranteed to be consistent.
We present here a brief overview of the Causally-coordinated SI model. The details of this model are presented in [38]. This model forms the basis for the development of the P-CSI protocol.

4.1.1 System Model

The system consists of multiple database sites and each site is identified by a unique siteId. Each site has a local database that supports multi-version data management and transactions. 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 update 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 update conflict checking for the items in their write-sets.

4.1.2 CSI Model

As noted earlier, the total ordering on transactions is not enforced by the CSI model. This eliminates the need of a global sequencer. Instead, a transaction is assigned a commit sequence number seq from a monotonically increasing local sequence counter maintained by its execution site. Thus, the commit timestamp for a transaction is a pair <siteId, seq>. Similarly, a data item version is identified by a pair <siteId, seq>. The local sequence number is assigned only when the transaction is guaranteed to commit, i.e. only when there are no update conflicts. 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 updates of the transactions from a particular site are always applied in the order of their sequence numbers, i.e. a transaction with sequence number n from site i is applied only when all the preceding transactions from site i with sequence number up to n-1 are applied. 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 [21, 36], which we denote by V, indicating the updates
of the transactions from other sites that it has applied to the local database. Thus, a site $i$ maintains a vector clock $V$, where $V[j]$ indicates that site $i$ has applied the updates of all transactions from site $j$ up to this timestamp, moreover, site $i$ has also applied all the other updates that causally precede these transactions. In the vector clock, $V[i]$ is set to the sequence number of the latest transaction committed at site $i$.

**Snapshot-based access**: A transaction $t$ executing at site $i$ is assigned, when it begins execution, a start snapshot timestamp $S_t$, which is set equal to the current vector clock $V$ value of site $i$. 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 start snapshot timestamp. Note that, because there is no total ordering, the versions can not be compared based on the timestamps assigned to the versions. Instead the versions are ordered based on the order in which they are applied. Thus, for a data item, the most recently applied version indicates the latest version of that item. Recall that the causal consistency property described above ensures that a data item version is applied only when all the preceding versions are applied. For each data item $x$, we maintain a version log which 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 or not, i.e. if $S_t[j] \geq n$. We then select the latest version that is visible in $t$’s snapshot. When $t$ performs a write operation, writes are kept in a local buffer until the commit time.

**Commit protocol**: If $t$ has modified one or more items, then it performs update conflicts 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 $S_t$ 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 items is locked by any other transaction in its conflict detection step. 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. 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: \(<siteId, seq>\). After performing these operations, the local site asynchronously propagates t’s updates to all the other sites. If all the items that t has modified have local site as the conflict resolver then t’s validation can be performed entirely locally.

Update propagation: For ensuring causal consistency, t’s updates are applied at remote sites only after the updates of all the causally preceding transactions have been applied. For update propagation, we define the 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 contrast to the PSI model [50] the approach taken in the CSI model avoids false causal dependencies. In other words, we capture causal dependencies with respect to a transaction rather than a site. The effective causal snapshot for a transaction t, executed at a site i is defined as a vector timestamp denoted by \(E_t\), and is determined as follows. \(E_t[i]\) 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 i is applied. The other elements of \(E_t\), i.e. those corresponding to the remote sites, are determined as follows:

\[
\forall j; j \neq i : E_t[j] = \max\{seq | \forall x \text{ s.t.}(x \in \text{readset}(t) \lor x \in \text{prevwrites}(t)) \land (\text{version}(x) =< j, seq >)\}
\]

Here, \(\text{prevwrites}(t)\) is the set of latest versions visible at that site for the items in the write-set of t. If this information about the write-set is not included, then it may happen that for a particular data item x modified by t, a remote site may store the version created by t without having all the preceding versions of x. We call it the missing version problem. This can violate the basic multi-version semantics of snapshot-based access in cases such as time-travel queries, which read from a specific older snapshot. It also complicates the version ordering logic described above. It should also be noted that the effective causal snapshot vector for a transaction is determined at the end of the transaction execution, and therefore the information about the read/write sets is needed only after the transaction execution has finished.

We illustrate the concept of effective causal snapshot timestamp vector with an example
shown in Figure 4.1. We have four sites, A, B, C, and D, with site ids 1, 2, 3, and 4, respectively. Figure 4.1 shows the view of site A, whose current vector clock is \(<5,3,4,5>\) indicating the transactions it has seen from other sites (shown by solid dots). Consider a transaction \(p\), executing at site A, which starts with this snapshot and reads items \(y\) and \(z\) and updates items \(x\) and \(w\). This transaction reads a version of \(y\) which was created by a transaction at \(C\) with sequence number 2, and it reads a version of \(z\) created by a transaction at \(D\) with sequence number 3. For items \(x\) and \(w\), the latest versions visible at site A are \(<1,4>\) and \(<4,2>\), respectively. Thus, if \(p\) commits with sequence number 6, the effective causal snapshot (shown by the dashed line in the figure) for this transaction would be \(<5,0,2,3>\).

![Figure 4.1: Example of Effective Causal Snapshot](image)

The update propagation protocol uses the \(E_t\) value of transactions while propagating their updates. Upon receiving updates, the remote site compares its vector clock with \(E_t\) vector to ensure that an update is applied at that site only when it has seen all the causally preceding updates. On applying the updates, the vector clock of the site is advanced appropriately.

### 4.2 Causal Snapshot Isolation with Partial Replication

The CSI model described above is targeted towards full replication scheme and and hence can not be used efficiently for partially replicated systems. We present below a model, called *Partitioned-CSI (P-CSI)*, which builds upon the CSI model to provide an efficient transaction isolation.
execution model for partially replicated systems. Towards this, we address a number of issues that arise due to partial replication with asynchronous update propagation. The first set of issues is related to ensuring that a transaction observes a consistent snapshot under the partial replication with asynchronous update propagation model. The second set of issues that we address is related to executing a transaction that involves accessing partitions stored at different site. When executing such a transaction, we must ensure that the snapshots used for accessing different partitions together form a globally consistent snapshot with respect to the atomicity and causality properties described above. The P-CSI model ensures these properties.

### 4.3 Issues in supporting causal consistency under partial replication

We now discuss the issues in extending the CSI model to provide causal consistency under partial replication. In a partial replication scheme, data items are not replicated at all sites. Ideally, one of the advantages of partial replication scheme is that the updates of a transaction need to be propagated only to the sites that store the data items accessed by the transaction. This reduces the communication cost compared to the full replication scheme as the updates are not required to be propagated to all sites. However, propagating a transaction’s updates only to the sites containing the items modified by the transaction raises issues with respect to supporting causal consistency.

Ensuring causality guarantees requires that for applying a given transaction’s updates to a partition replica at a site, all its causally preceding events, including the transitive dependencies, must be captured in the views of the local partitions of that site. We illustrate this problem using the example scenario shown in Figure 4.2. In this example, partition P1 containing item x is replicated at sites 1 and 3. Partition P2 containing item z is replicated at sites 1, 2, and 3. Partition P3 containing y is replicated at sites 2 and 4. Transaction T1 executed at site 1 updates item x and creates version x(100). This update is asynchronously propagated to site 3, shown by a dashed arrow in the figure. Transaction T2 executed at site 2 reads x(100) from partition P1 at site 1 and updates y to create version y(200). Later transaction T3 is executed...
at site 2, which reads $y(200)$ and modifies $z$ to create version $z(300)$. Note that version $z(300)$ causally depends on $x(100)$. The update of $T3$ is propagated asynchronously to sites 1 and 3. Suppose the update of $T3$ for $z(300)$ arrives at site 3 before the update of transaction $T1$ for $x(100)$. In case of full replication using the CSI model, all transactions’ updates are sent to all sites and the update of $T3$ in the above scenario would only be applied after the updates of transaction $T1$ and $T2$ are applied. However, with partial replication shown in Figure 4.2, the updates of $T2$ would never be sent to site 3. Therefore, we need update synchronization mechanism that selectively waits for the updates of transaction $T1$ but not $T2$. Applying the update $z(300)$ before applying the update of $x(100)$ will result in causally inconsistent state of partitions $P1$ and $P2$ at site 3.

A straightforward solution for supporting causal consistency requires either (1) maintaining the entire causal dependencies graph for every item version [33], or (2) communicating every update to all the sites in the system so that each site is cognizant of all causal dependencies for a data item version. The first solution is not feasible since the causal dependency graph can potentially become very large. The second solution nullifies the advantage of partial replication, since it requires communicating the updates to all sites [6]. The proposed P-CSI model presents a more efficient protocol which ensures causal consistency and requires propagating the
transaction’s updates only to the sites storing the items modified by the transaction.

Next we illustrate the issues that arise when executing a transaction that needs to access some partitions stored at a remote site. Suppose, in the example shown in Figure 4.2, at site 4 transaction \( T4 \) is executed which reads items \( x, y, \) and \( z \). This transaction reads \( y(199) \) from the local partition \( P3 \) and reads \( x(100) \) and \( z(300) \) from site 1 since site 4 does not contain partitions \( P1 \) and \( P2 \). In Figure 4.3, we show the causal relationships between the versions of items \( x, y, z \). We also show the snapshot observed by \( T4 \), which is causally inconsistent because it contains \( z(300) \) but not the causally preceding version \( y(200) \).

Figure 4.3: Causally inconsistent snapshot due to remote reads

Another issue that arises when reading data from remote partitions under asynchronous propagation is related to the atomicity property of consistent snapshots. We illustrate this with an example shown in Figure 4.4. Here partition \( P1 \) containing item \( x \) is replicated at sites 1 and 2, and partition \( P2 \) containing \( y \) is replicated at sites 1 and 3. Transaction \( T1 \) executed at site 1 updates \( x \) and \( y \), creating versions \( x(100) \) and \( y(50) \). The updates of this transaction are propagated asynchronously to sites 2 and 3. Suppose that site 3 executes transaction \( T2 \) which reads item \( x \) and \( y \). \( T2 \) is executed before site 3 applies the update of \( T1 \) for version \( y(50) \). \( T2 \) reads \( y(49) \) from its local partition and reads \( x(100) \) from site 1. This reflects an atomically inconsistent snapshot of partitions \( P1 \) and \( P2 \) with respect to items \( x \) and \( y \). In the
next section, we present the P-CSI model to address such issues in ensuring causal consistency.

**Figure 4.4: Issues in obtaining atomically consistent snapshot**

### 4.4 Partitioned-CSI Model

#### 4.4.1 System Model

We consider partial replication of database that consists of a finite set of data items partitioned into multiple disjoint data partitions. The system consists of multiple sites, and each site contains one or more partitions. Each partition is replicated across one or more sites. Each site supports multi-version data management and SI-based transactions.

As in the case of CSI, for each data item, there is a designated conflict resolver site which is responsible for checking for update conflicts for that item. Transactions can execute at any site. A transaction may access items in multiple partitions. For partially replicated databases, the P-CSI model ensures all the guarantees provided by CSI. These properties are ensured even when a transaction accesses items from multiple sites.

#### 4.4.2 Vector Clocks and Partition Dependency View

Our goal in designing the P-CSI model is to avoid the need of propagating updates to all sites. Our solution to this problem is based on maintaining vector clocks on per-partition basis. In the following discussion, we refer to the partitions stored at a site as the local partitions of that
site. In the P-CSI model, each site maintains a vector clock for each local partition, referred to as the partition view ($V_p$). The partition view $V_p$ for a local partition $p$ maintained by site $j$ indicates the sequence numbers of transactions from all sites that have updated any of the items in partition $p$ and have been applied to the local partition at site $j$. Thus, the value $V_p[k]$ indicates that site $j$ has applied all the transactions pertaining to partition $p$ from site $k$ up to this much timestamp as well as all the causally preceding transactions. The $V_p$ value at a particular time identifies the state of the partition $p$ visible at site $j$ at that time. A site also maintains a sequence counter for each of its local partitions, which is used to assign sequence numbers to local transactions modifying items in that partition. A transaction executing at a site obtains, during its commit phase, a local sequence number for each partition it is modifying.

Events of interest in the system are the update events performed by a transaction. A transaction may update multiple partitions thus resulting in distinct update events in different partitions. We define an atomic event set as the set of all update events of a given transaction. In any consistent snapshot either all or none of the events of an atomic event set are present.

A site also maintains for each local partition $p$ a partition dependency view ($D_p$) which is a set of vector clocks. It represents a consistent global snapshot, capturing both atomicity and causality properties. For causality, it identifies for each of the other partitions the events that have occurred in that partition and that causally precede the partition $p$’s state as identified by its current partition view. In other words, $D_p$ indicates the state of other partitions on which the state of partition $p$ is causally dependent. For atomicity, it captures the atomic event sets of all the transactions applied to partition $p$. The partition dependency view $D_p$ consists of a vector clock for each other partition. Formally, $D_p$ is a set of vector clocks $\{D^1_p, D^2_p, \cdots, D^n_p\}$, in which an element $D^q_p$ is a vector clock corresponding to partition $q$. Each element of the vector clock $D^q_p$ identifies the transactions performed on partition $q$ that causally precede the transactions performed on partition $p$ identified by $V_p$. Note that partition $q$ may or may not be stored at site $j$. Also, note that vector clock $D^p_p$ is the same as $V_p$. A site maintains partition dependency views for each local partition. We describe below how the partition dependency views are maintained and how they are used to ensure causal consistency.
4.4.3 Transaction Execution Protocol

We now give a brief overview of the transaction execution model of P-CSI and then discuss the various issues involved in it. For simplicity of the discussion, we first describe the execution of a transaction at a site which stores all the partitions accessed by the transaction. Later, we discuss how a transaction that requires accessing partitions at remote sites is executed. A transaction $t$ goes through a series of execution phases. In the first phase it obtains a start snapshot time ($S_t$), which is a set of vector clocks corresponding to the partitions to be accessed by the transaction. An element $S^p_t$ in $S_t$ indicates the start snapshot time for partition $p$. The transaction then performs read operations using the start snapshot time. P-CSI protocol ensures that the snapshot observed by a transaction is causally consistent.

When the transaction reaches its commit point it needs to coordinate with the conflict resolver sites of the items in its write-set, as in the CSI protocol, to check for update conflicts. Read-only transactions can be executed locally without needing any coordination with remote sites. After successful validation, the transaction executes its commit phase. In the commit phase, the transaction obtains from the local site a sequence number for each partition that it modified. This sequence number is used to assign a timestamp to the transaction for that partition. A timestamp is defined as a pair $<\text{siteId}, \text{seq}>$, where $\text{seq}$ is a local sequence number assigned to the transaction by the site identified by $\text{siteId}$. The commit timestamp vector ($C_t$) of transaction $t$ is a set of timestamps assigned to the transaction corresponding to the partitions modified by the transaction. Thus, the commit timestamp vector $C_t$ of transaction $t$ modifying $n$ partitions is a set of timestamps $\{C^1_t, C^2_t, \ldots, C^n_t\}$. For an item modified by transaction $t$ in partition $q$, the version number of the item is identified by commit timestamp $C^q_t$.

A transaction’s updates are applied to the local database and then asynchronously propagated to other sites which store any of the partitions modified by the transaction. The information transmitted to sites in the update propagation message includes a set of vector clocks, called transaction dependency view ($TD$), containing a vector clock corresponding to each partition. The transaction dependency view for a transaction $t$ identifies all the transactions that causally precede $t$. At the remote site, the updates of transaction $t$ are applied only when all the events identified in $TD_t$ have been applied. We describe the details of the update propagation later.
We describe below in details the various steps in the transaction execution protocol.

**Obtaining start snapshot time:** In the case when all partitions to be accessed by the transaction are local, the start snapshot time \( S_t \) for transaction \( t \) is obtained using the partition view values for those partitions. The pseudocode for obtaining start snapshot time is shown in Algorithm 5. Later in Algorithm 13 we generalize this for a transaction accessing partitions from remote sites.

**Algorithm 5 Obtaining start snapshot time**

```
function GetSnapshot
    \( \mathcal{P} \leftarrow \) partitions accessed by the transaction
    [ begin atomic action
    for each \( p \in \mathcal{P} \) do
        \( S_{p}^{t} \leftarrow V_p \)
    end atomic action ]
```

**Algorithm 6 Performing read operations**

```
function read(item x)
    \( p \leftarrow \) partition containing item \( x \)
    /* performed in reverse temporal order of versions */
    for each version \( v \in \) version log of \( x \) do
        if \( S_{p}^{t}[v.siteId] \geq v.seqno \) then return \( v.data \)
```

**Performing read operations:** When a transaction reads a data item \( x \) from a partition \( p \), we determine the latest version for \( x \) that must be visible to the transaction based on the transaction’s start snapshot time \( S_{p}^{t} \) for that partition. The procedure to read a data item version from partition \( p \) according to transaction’s snapshot \( S_{p}^{t} \) is the same as in the CSI model. Algorithm 6 gives the pseudocode for performing read operations. In case of write operations, the writes are buffered locally until the commit time.
Algorithm 7 Update conflict checking performed by a transaction at site $j$

function $\text{CheckConflicts}(\text{writeset})$

```
sites ← conflict resolver sites for items $\in$ writeset

for each $s \in sites$

    itemList ← $\{x \mid x \in$ writeset $\land$ resolver$(x) = s\}$

    send prepare message to $s$ : $(itemList, S)$

if all votes are ‘yes’ then

    perform Commit function as shown in Algorithm 10

    to each $s \in sites$ send commit message: $(itemList, C_t)$

else

    abort transaction and send abort message to each $s \in sites$
```

Update conflict checking: When the transactions reaches its commit point, it performs update conflict checking if it has modified any items. Algorithm 7 shows the protocol for conflict checking. The update conflict checking is performed using a two-phase commit (2PC) protocol with the conflict resolver sites responsible for the items contained in the transaction’s write-set. The procedure to check for update conflicts is same as described in Section 4.1 for the CSI model.
Algorithm 8 Functions executed by the conflict resolver site

/* upon receiving prepare message for transaction $t$ */

function $\text{RecvPrepare}(itemList, S_t)$

for each $x \in itemList$

$\quad p \leftarrow$ partition containing item $x$

$\quad v \leftarrow$ latest version of item $x$

$\quad$ if $S^p_t[v.siteId] \geq v.seqno$ \& \& $x$ is unlocked then

$\quad\quad$ lock $x$

$\quad$ else

$\quad\quad$ return response with ‘no’ vote

if all $x \in itemList$ are locked then send response with ‘yes’ vote

/* upon receiving commit message for transaction $t$ */

function $\text{RecvCommit}(itemList, C_t)$

for each $x \in itemList$

$\quad p \leftarrow$ partition containing item $x$

$\quad$ record version timestamp $C^p_t$ in version log and release lock on $x$

/* upon receiving abort message for transaction $t$ */

function $\text{RecvAbort}(itemList)$

release locks on all $x \in itemList$
Algorithm 9 Computing transaction dependency view for transaction $t$

\begin{algorithm}
\textbf{function} \text{ComputeTransactionDependency} \\
\hspace{1em}$P \leftarrow \text{set of partitions on which } t \text{ performed any read/write operation.}$ \\
\hspace{1em}$E_t \leftarrow \text{set of effective causal snapshot vectors corresponding to partitions in } P$ \\
\hspace{2em}\textbf{for each} $p \in P$ \textbf{do} \\
\hspace{3em}$TD^p_t \leftarrow E^p_t$ \\
\hspace{4em}[ begin atomic action \\
\hspace{5em}\textbf{for each} $p \in P$ \textbf{do} \\
\hspace{6em}\textbf{for each} $D^q_p \in D_p$ \textbf{do} \\
\hspace{7em}$TD^q_t \leftarrow \text{super}(D^q_p, TD^p_t)$ \\
\hspace{6em}$\text{end atomic action } ]$
\end{algorithm}

\begin{algorithm}
\textbf{function} \text{super}(V_1, V_2, \ldots, V_k) \text{ returns } V \\
\hspace{1em}$\forall i, V[i] = \max(V_1[i], V_2[i], \ldots, V_k[i])$
\end{algorithm}

**Determining transaction dependencies:** After successful validation, transaction $t$ computes its transaction dependency view $TD_t$. $TD_t$ is a set of vector clocks $\{TD^1_t, TD^2_t, \ldots, TD^9_t, \ldots, TD^n_t\}$, in which an element $TD^q_t$ identifies the transactions performed on partition $q$ which causally precede $t$. Algorithm 9 shows the pseudocode for computing $TD_t$. Transaction $t$ computes its effective causal snapshot vector $E_t$, which is a set containing vector clocks for each partition accessed by $t$. Element $E^p_t$ in this set is the effective causal snapshot corresponding to partition $p$ and is computed as discussed in Section 4.1, considering the items read/written by $t$ from partition $p$. $E_t$ captures the causal dependencies for transaction $t$, solely based on items read or written by $t$. To capture transitive dependencies, we include in $TD_t$ the partition dependency view vectors of each partition accessed by $t$. If any two partitions $p_1$ and $p_2$ accessed by $t$ each have in their dependency views an element for some other partition $q$, i.e. $\exists q \text{ s.t. } D^q_{p_1} \in D_{p_1} \land D^q_{p_2} \in D_{p_2}$, then we take element-wise max value from $D^q_{p_1}$ and $D^q_{p_2}$ using the ‘super’ function shown in Algorithm 9.

**Commit phase:** Algorithm 10 shows the commit protocol for transaction. A commit timestamp
vector $C_t$ is assigned to the transaction by obtaining a sequence number of each partition modified by $t$. The updates made by transactions are written to the local database and transaction dependency view set $TD_t$ is computed. The partition views $V$ and dependency views $D$ of all updated partitions are advanced using $TD_t$ and $C_t$, as shown in the function ‘AdvanceVectorClocks’. If the committed transaction involves modifying items in multiple partitions, then the above procedure ensures that the partition dependency view $D$ for each modified partition is updated using the $C_t$ value to capture all events in the atomic set of the transaction. The above procedure is done as a single atomic action to ensure that the transaction’s updates are made visible atomically. The updates, along with $TD_t$ and $C_t$ values, are propagated to every site that stores any of the partitions modified by $t$. The update propagation can be started asynchronously once the $TD_t$ and $C_t$ have been computed.
Algorithm 10 Commit Protocol for transaction at site $j$

**function** COMMIT($writeset$)

- $\mathcal{P} \leftarrow$ partitions pertaining to $writeset$.

  **for each** $p \in \mathcal{P}$ **do**

  - $ctr_p \leftarrow$ local sequence counter for partition $p$
  - $C_t^p.seq \leftarrow ctr_p++$

  ApplyUpdates($writeset, C_t$)

  /* compute dependencies as shown in Algorithm 9 */

- $TD_t \leftarrow$ ComputeTransactionDependency()

- AdvanceVectorClocks($TD_t, C_t$) /* advance local vector clocks */

  propagate to every site that stores any partition $p \in \mathcal{P}$

  ($TD_t, writeset, C_t$)

**function** APPLYUPDATES($writeset, C_t$)

- $\mathcal{P} \leftarrow$ partitions pertaining to $writeset$.

  **for each** $p \in \mathcal{P}$ **do**

    **for each** item $x$ in $writeset$ pertaining to $p$ **do**

    - write the new version of $x$ to the local database.
    - record version timestamp $C_t^p$ in version log
Algorithm 11 Function to update vector clocks for partitions

function ADVANCEVECTORCLOCKS(TD\textsubscript{t}, C\textsubscript{t})

\[ \mathcal{P} \leftarrow \text{partitions pertaining to writeset} \]

[ begin atomic region

for each \( p \in \mathcal{P} \) do

\[
\text{for each } TD\textsubscript{t}^q \in TD\textsubscript{t} \text{ s.t. } q \neq p \text{ do }
\]

\[ D\textsubscript{p} \leftarrow \text{super}(TD\textsubscript{t}^q, D\textsubscript{p}) \]

/* Advance \( D \) to capture the \( t \)'s update events in other partitions */

\[
\text{for each } C\textsubscript{t}^q \in C\textsubscript{t} \text{ s.t. } q \neq p \text{ do }
\]

\[ D\textsubscript{p}[C\textsubscript{t}^q,\text{siteId}] \leftarrow C\textsubscript{t}^q,\text{seq} \]

\[ V\textsubscript{p}[C\textsubscript{t}^p,\text{siteId}] \leftarrow C\textsubscript{t}^p,\text{seq} \]

end atomic region ]

Applying updates at remote sites: When a remote site \( k \) receives update propagation for \( t \), it checks if it has applied, to its local partitions, updates of all transactions that causally precede \( t \) and modified any of its local partitions. Thus, for every partition \( p \) specified in \( TD\textsubscript{t} \), if \( p \) is stored at site \( k \), then site checks if its partition view \( V\textsubscript{p} \) for that partition is advanced up to \( TD\textsubscript{p} \). Moreover, for each of the modified partitions \( p \) for which the remote site stores a replica of \( p \), the site checks if \( V\textsubscript{p} \) of the replica contains all the events preceding the sequence number value present in \( C\textsubscript{p} \). If this check fails the site defers the updates locally and enters a synchronization phase which includes either pulling the required updates or delaying the application of updates until the vector clocks of the local partitions advance enough. If this check is successful, the site applies the updates to the corresponding local partitions. Updates of \( t \) corresponding to any non-local partitions are ignored. The partition views at site \( k \) are advanced as shown in procedure ‘AdvanceVectorClock’ in Algorithm 10.
Algorithm 12 Applying updates at a remote site \( k \)

function \( \text{RecvUpdatePropagation}(\mathcal{T}D_t, \text{writeset}, \mathcal{C}_t) \)

/*check if the site is up to date with respect to \( \mathcal{T}D_t \) */

for each \( \mathcal{T}D^p_t \in \mathcal{T}D_t \) do

if \((p \text{ is local partition}) \land V^p < \mathcal{T}D^p_t \) then

   synchronize phase: buffer the updates locally and either pull the required causally preceding

   updates or wait till the vector clock advances enough.

for each \( \mathcal{C}^p_t \in \mathcal{C}_t \) do

   if \((p \text{ is local partition}) \land \mathcal{V}^p[\mathcal{C}^p_t, \text{siteId}] < \mathcal{C}^p_t, \text{seq}-1 \) then synchronize phase as shown above

   ApplyLocalUpdates(writeset, \mathcal{C}_t) /* apply updates to local partitions at site \( k \) */

AdvanceVectorClocks(\mathcal{T}D_t, \mathcal{C}_t) /* advance vector clocks of site \( k \) */

4.4.4 Execution of Multi-site Transactions

It is possible that a site executes a transaction that accesses some partitions not stored at that site. This requires reading/writing items from remote site(s). One important requirement while performing a multi-site transaction is to ensure that the transaction observes a consistent global snapshot.

We describe how the start snapshot vector is determined. The Algorithm 13 shows the modified ‘GetSnapshot’ function. Note that at a given site the partition dependency view of any partition reflects a consistent global snapshot. One can thus simply take the \( D \) vector set of any one of the local partitions to be accessed by the transaction. However, it is possible that such a set may not contain a vector corresponding to some partition to be accessed by the transaction. We present below a procedure to obtain a snapshot for all partitions to be accessed by the transaction.

We can form a consistent global snapshot by combining the partition dependency views of all the local partitions. If two local partitions contain in their \( D \) sets a vector for some partition \( p \), then we can take ‘super’ of these two vectors as the snapshot for \( p \). We follow this rule for each
partition to be accessed across all local partition dependency views to form a global snapshot. Such a snapshot is consistent because the causal and atomic set dependencies of all the local partitions are collectively captured in this snapshot.

It is still possible that this set may not have a snapshot vector for some partition to be accessed by the transaction. For each such partition \(q\), we then need to follow a procedure to obtain a snapshot from some remote site containing that partition. We read the partition view \(V_q\) of the remote site and consider it as the snapshot for \(q\) provided that its causal dependencies as indicated by the \(D_q\) set at the remote site have been seen by the local site. The function ‘GetRemoteSnapshot’ performs this step.

Algorithm 13 Obtaining snapshot for a multi-site transaction \(t\) at site \(i\)

function GETSNAPSHOT

\[
\begin{align*}
\mathcal{L} & \leftarrow \text{local partitions accessed by } t \\
\mathcal{R} & \leftarrow \text{non-local partitions accessed by } t \\
\text{for each } p \in \mathcal{L} & \text{ do} \\
\quad S_{t}^{p} & \leftarrow V_{p} \\
\text{for each } q \in \mathcal{R} & \text{ do} \\
\quad \text{for each } p \in \text{partitions}(i) & \text{ do} \\
\quad\quad \text{if } D_{q}^{p} \in D_{p} & \text{ then} \\
\quad\quad\quad S_{t}^{q} & \leftarrow \text{super}(D_{p}^{q}, S_{t}^{q}) \\
\text{for each } q \in \mathcal{R} & \text{ such that } S_{t}^{q} \notin S_{t} \text{ do} \\
\quad S_{t}^{q} & \leftarrow \text{GetRemoteSnapshot}(S_{t}) \\
\text{if } S_{t}^{q} & \text{ is null then repeat above step using some other replica site for partition } q
\end{align*}
\]

/* Function executed at remote site \(j\) to obtain snapshot for \(t\) for partition \(q\) */

function GETREMOTESNAPSHOT\((S_{t}, \text{partition } q)\)

\[
\begin{align*}
\text{for each } r & \text{ such that } D_{q}^{r} \in D_{q} \land S_{t}^{r} \in S_{t} \text{ do} \\
\quad \text{if } S_{t}^{r} \prec D_{q}^{r} & \text{ then} \\
\quad\quad \text{return null} \\
\text{return } V_{q}\n\end{align*}
\]
After obtaining the start snapshot time, transaction $t$ performs local reads as shown in Algorithm 6. For a remote read, site $i$ contacts the remote site $j$ which then performs a local read and returns the version. Before performing the read operation, site $j$ checks if it is advanced up to the transaction's snapshot for that partition. This check is needed only in the case when the transaction did not need to contact the remote site when it constructed its start snapshot time.

If a transaction involves updating any remote partition, it must make sure to obtain a snapshot vector for that partition at the start of the transaction, as described above. In the commit phase it contacts that site to obtain a sequence number for that partition. The rest of the commit protocol is performed as shown in Algorithm 10. The updates to local partitions are applied first and remote updates are sent to the remote site using the update propagation mechanism described above. Even though there is a delay in applying updates to the remote partition, the atomicity guarantee in obtaining a snapshot is still guaranteed because the $D$ vector set of the local partitions would force the use of the updated view of the remote partition.

4.4.5 Discussion of Protocol Correctness

We now discuss the correctness of the P-CSI protocol. The property central to the correctness of the protocol is that the partition dependency view $D$ of any partition at a site reflects a consistent global snapshot across all partitions. The partition dependency view is updated whenever a transaction is executed modifying that partition. Initially, $D$ sets for all partitions are empty, and therefore this property holds trivially.

We show that this property holds by using induction on the length of causal sequence of transactions executed at a site. Assuming that this property holds for transaction sequences up to length $n$, we show that this property is preserved when a new causally succeeding transaction $t$ is executed at the site extending the length of a causal sequence to $n + 1$.

As shown in Algorithm 10, in the commit phase the transaction $t$ updates some partitions and updates their $D$ vector sets and partition views. The first step involves obtaining commit timestamps for each partition to be modified. It then inserts new versions of the modified items in these partitions. However, these versions are not visible to any transaction since the partition
views are yet not advanced. Next, the commit protocol computes the transaction dependency view $\mathcal{T}\mathcal{D}_t$. The procedure shown in Algorithm 9, constructs a vector clock for each partition $r$ on which transaction $t$ is causally dependent to capture all the causally preceding events in partition $r$. This is denoted by $\mathcal{T}\mathcal{D}_t^r$. These causal dependencies arise because of partitions accessed by $t$ which happen to be dependent on events in $r$. Suppose that $t$ accesses partitions $\{P_1, P_2, \ldots, P_k\}$, then $\mathcal{T}\mathcal{D}_t^r$ has the following property, where $\mathcal{E}_t^r$ is the effective causal snapshot of partition $r$ if it is accessed by $t$.

$$\mathcal{T}\mathcal{D}_t^r = \text{super}(\mathcal{D}_{P_1}^r, \mathcal{D}_{P_2}^r, \ldots, \mathcal{D}_{P_k}^r, \mathcal{E}_t^r) \quad (4.1)$$

The above expression means that $\mathcal{T}\mathcal{D}_t^r$ includes all causal dependencies of transaction $t$ on partition $r$.

Next the transaction updates the $\mathcal{D}$ vector sets of all the modified partitions using the $\mathcal{T}\mathcal{D}_t$ vector. For each such partition $p$ and each $r$ in $\mathcal{T}\mathcal{D}_t$, it sets $\mathcal{D}_p^r$ equal to $\mathcal{T}\mathcal{D}_t^r$, thus updating the dependency view of a modified partition $p$ to include its dependency on events in $r$. Moreover, for each modified partition $p$, $\mathcal{D}_p^q$ for each other modified partition $q$, $q \neq p$, is modified using the $\mathcal{C}_t^q$ vector so that $\mathcal{D}_p$ includes in its view the update event of $t$ on partition $q$. This ensures that the partition dependency view of each modified partition captures all events in the atomic set of transaction $t$. The steps of modifying $\mathcal{D}$ vectors are performed as a single atomic action to ensure that any other concurrent transaction would always observe a consistent snapshot. The modified $\mathcal{D}$ vector sets ensure the consistency property mentioned above.

Propagation of transaction $t$’s updates to remote sites includes $\mathcal{T}\mathcal{D}_t$, $\mathcal{C}_t$ and the write-set. Before applying updates of transaction $t$, each remote site ensures that for each partition $r$ stored at the remote site, it has applied all the causally preceding events implied by $\mathcal{T}\mathcal{D}_t^r$. When the updates of $t$ are applied, the procedure followed for updating $\mathcal{D}$ vector set and partition views of the partitions located at the remote site is the same as that described above for the execution site. Thus the modified $\mathcal{D}$ vector sets at the remote sites also ensure the consistency property mentioned above.
4.4.6 Example

We illustrate the working of the protocol with an example. Consider a replicated system with 4 sites S1, S2, S3, and S4 and 4 partitions p1, p2, p3, and p4. Site S1 stores partition p1 and p4, Site S2 stores partition p2 and p4, Site S3 stores partition p2 and p3, and Site S3 stores partition p1 and p3. Table 4.1 shows the initial database state and the vector clock values for all the sites. Data item x, y, z, and w belong to partitions p1, p2, p3, and p4, respectively. The vector clock values are shown as <siteId:seqno, ⋯> indicating the events seen by a site from other sites. Thus, for example, the partition vector clock value for partition p1 at site S4, which is <S1:100, S2:0>, indicates that S4 has seen events modifying the state of partition p1 from site S1 up to sequence number 100. Furthermore, S4 has not executed any transaction that modifies any items in partition p4. Suppose a transaction t1 is performed at site S2 which reads item x from partition p1 and modifies item y from partition p2. We illustrate in Algorithm 14 the various steps performed at different sites during the execution and propagation of t1. We also show in this pseudo-code how the values of various vector timestamps such as start snapshot timestamp, effective causal snapshot timestamp and transaction dependency vector are computed.

Before starting the execution t1 at site S2 a start snapshot timestamp S_{t1} is obtained for partition p1 and p2. S_{t1} is set to the current value of partition view V_{p2} at site S2. For partition p1, since S2 does not have a vector clock value for p1 in any of its partition dependency views, a snapshot timestamp for p1 is obtained from remote site S1. S1 returns the current value of V_{p1} as the start snapshot for p1. After obtaining start snapshot, the read/write operations of t1 are executed which involves remote read from S1 for item x. The effective causal snapshot is calculated as follow. Since t1 accessed the version <S1:100> of x from partition p1, E_{t1}^{p1} is set to

<table>
<thead>
<tr>
<th>Database State</th>
<th>Site 1</th>
<th>Site 2</th>
<th>Site 3</th>
<th>Site 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: x <a href="">S1:100</a></td>
<td>p2: y <a href="">S2:199</a></td>
<td>p3: z <a href="">S3:299</a></td>
<td>p4: w <a href="">S4:0</a></td>
<td></td>
</tr>
<tr>
<td>p2: y <a href="">S2:199</a></td>
<td>p3: z <a href="">S3:299</a></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p3: z <a href="">S3:299</a></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p4: w <a href="">S4:0</a></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Initial State of Sites

<table>
<thead>
<tr>
<th>Vector Clocks</th>
<th>Site 1</th>
<th>Site 2</th>
<th>Site 3</th>
<th>Site 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>V_{p1} = &lt;S1:100, S4:0&gt;</td>
<td>V_{p2} = &lt;S2:199, S3:0&gt;</td>
<td>V_{p3} = &lt;S3:299, S4:0&gt;</td>
<td>V_{p4} = <a href="">S4:0</a></td>
<td></td>
</tr>
<tr>
<td>D_{p1} = ∅</td>
<td>D_{p2} = ∅</td>
<td>D_{p3} = ∅</td>
<td>D_{p4} = ∅</td>
<td></td>
</tr>
<tr>
<td>V_{p4} = &lt;S1:50, S2:0&gt;</td>
<td>V_{p4} = &lt;S1:50, S2:0&gt;</td>
<td>V_{p4} = &lt;S1:50, S2:0&gt;</td>
<td>V_{p4} = &lt;S1:50, S2:0&gt;</td>
<td></td>
</tr>
<tr>
<td>D_{p4} = ∅</td>
<td>D_{p4} = ∅</td>
<td>D_{p4} = ∅</td>
<td>D_{p4} = ∅</td>
<td></td>
</tr>
</tbody>
</table>
For item \( y \) in partition \( p_2 \) to be modified by \( t_1 \), the latest version of \( y \) visible to \( t_1 \) is \( <S2:199> \). Therefore, \( E_{t_1}^{p_2} \) is set to \( <S2:199, S3:0> \). The transaction then proceeds to commit phase. Upon successful validation, \( t_1 \) is assigned a commit timestamp for partition \( p_2 \) using the local sequence counter for \( p_2 \) at site \( S2 \). \( t_1 \)'s updates are applied to local database. The transaction dependency vector \( T\mathcal{D}_{t_1} \) is computed as follows. First, \( T\mathcal{D}_{t_1}^{p_1} \) and \( T\mathcal{D}_{t_1}^{p_2} \) are set to \( E_{t_1}^{p_1} \) and \( E_{t_1}^{p_2} \), respectively. Then partition dependency views of \( p_1 \) and \( p_2 \), i.e. \( D_{p_1} \) (obtained from Site \( S1 \)) and \( D_{p_2} \), are examined for capturing transitive causal dependencies. Since both the vector sets are empty, no further updates are made to \( T\mathcal{D} \) vector. The updates of \( t_1 \) are then propagated to Site \( S3 \) since \( S3 \) stores a replica of \( p_2 \). When site \( S3 \) receives the propagation message for \( t_1 \) from \( S2 \) it first checks if it has applied all the events causally preceding \( t_1 \) which modifies any partition stored at \( S3 \). Thus \( S3 \) verifies that its current partition view for \( p_2 \), \( V_{p_2} \), is advanced enough, i.e. \( V_{p_1} \geq T\mathcal{D}_{t_1}^{p_2} \). Furthermore it checks if it has applied all the events from site \( S2 \) modifying \( p_2 \) that precede \( t_1 \). It then applies \( t_1 \)'s updates and advances the vector clock values accordingly.
**Algorithm 14 Executing Transaction $t_1: read(x), write(y)$**

**Executing Transaction at $t_1$ Site 2**

- Get snapshot for $p_1$ from Site 1
- Set $S_{t_1}^{p_1} = \langle S1:100, S4:0 \rangle$
  
  \[ S_{t_1}^{p_2} = \langle S2:199, S3:0 \rangle \]

- Execute read/write operations
- Determine effective causal snapshot $E_{t_1}$:
  
  \[ E_{p_1}^{t_1} = \langle S1:100, S4:0 \rangle \]
  
  \[ E_{p_2}^{t_1} = \langle S2:199, S3:0 \rangle \]

- **After successful validation of $t_1$**

  - Obtain commit timestamp $C_{p_2}^{t_1} = \langle S2:200 \rangle$ and apply updates
  
  - Compute transaction dependency $TD_{t_1}$:
    
    \[ TD_{p_1}^{t_1} = \langle S1:100, S4:0 \rangle \]
    
    \[ TD_{p_2}^{t_1} = \langle S2:199, S3:0 \rangle \]

  - Advance local vector clocks:
    
    \[ V_{p_2} = \langle S2:200, S3:0 \rangle \]
    
    \[ D_{p_1}^{p_2} = \langle S1:100, S4:0 \rangle \]

  - Propagate $t_1$’s update to Site 3 with $TD$ and commit timestamp $C_{t_1}$

**After receiving $t_1$’s propagation message at Site 3:**

- Check $V_{p_2} \geq TD_{p_2}^{t_1} \land V_{p_2}[S1] = C_{p_2}^{t_1}$
- Apply $t_1$’s update
- Advance local vector clocks:
  
  \[ V_{p_2} = \langle S2:200, S3:0 \rangle \]
  
  \[ D_{p_2}^{p_1} = \langle S1:100, S4:0 \rangle \]

### 4.4.7 Implementation Details

We implemented a prototype system implementing the P-CSI model. In our prototype, we implemented an in-memory key-value store to serve as the local database for a site. Each
site also maintains a ‘commit-log’ in secondary storage. During the commitment of an update transaction, the updates are written to the ‘commit log’. Committed update transactions are propagated by the execution site at periodic intervals, set to 1 second. During the update synchronization phase at the remote site (refer Algorithm 12), if the updates cannot be applied, then the site buffers the updates locally and delays their application until the corresponding vector clock values have been advanced enough.

![Diagram](image)

**Figure 4.5: Transaction Throughput Scalability**

Figure 4.5 shows the architecture of the prototype implementing a database site. Each database partition is managed by a `PartitionManager`, which also hosts a `StorageManager` for maintaining the data for that partition. The `StorageManager` component provides a get/put interface to the clients and stores data in the underlying storage system. It also implements the version management logic described above. The underlying storage can be implemented in various ways. It can be a local in-memory storage, a distributed key-value system or a relational database system. Our prototype system includes two types of storage manager implementations - `InMemoryStorageManager` and `HBaseStorageManager`. The `InMemoryStorageManager` implements an in-memory key-value store and uses `write-ahead log` maintained in the network.
file system for data durability. The HBaseStorageManager uses HBase [4] for data storage. In this implementation, the storage manager ignores the version timestamps assigned by the HBase system and instead records for each data version the $<$siteId,seqno$>$ information in HBase to implement the version management scheme of P-CSI.

A PartitionManager also maintains vector clocks for partition view and partition dependency view for that partition. The transactions are managed by the TransactionManager component. TransactionManager creates a transaction handler thread for each transaction, which executes the read/write operations on the database. The functions of propagating transaction updates to remote site and receiving and applying updates from remote sites are handled by the Replication-Controller component. A site also includes a ConflictResolver component which is responsible for checking write-write conflicts.

4.5 Evaluations

We present below the results of our evaluation of the P-CSI model. In these evaluations, we were interested in evaluating the following aspects: (1) scalability of the P-CSI model, (2) advantages of partial replication using P-CSI over full replication, (3) impact of locality in transaction execution, and (4) cost of ensuring causality. The primary performance measures used were: (1) transaction throughput measured as committed transactions per second, (2) transaction latency, measured in milliseconds, (3) visibility latency, measured as described below and (4) cost of update propagation, measured as number of propagation messages sent per transaction.

Visibility Latency: An important measure in determining the cost of ensuring causality is the visibility latency of an update, which is the time since an update is committed till the time it is applied at all the replicas storing the updated items. This depends on three factors: network latency, delays in propagating updates at the execution site because of the lazy propagation model, and delays in applying updates at the remote site due to causal dependencies. In our experiments the update propagation interval was set to 1 second. Visibility latency indicates the inconsistency window i.e. the time for which the data copies at different replicas are inconsistent with respect to a given update.
4.5.1 Experiment Setup

System Environments

We performed the evaluations using two types of system environments. The first environment is a local cluster of 30 nodes using the resources provided by Minnesota Supercomputing Institute (MSI). In this cluster, each node had 8 CPU cores with 2.8 GHz Intel X5560 "Nehalem EP" processors, and 22 GB main memory. Each node in the cluster served as a single database site in our experiments. The other system environment uses Amazon EC2 cloud service [3] to create a system of geographically distributed sites. In this environment, we use total of 8 sites situated in geographically distributed cloud datacenters provided by Amazon EC2. Each site is implemented using a single instance provided by EC2. Each instance is of EC2 ‘Extra-Large’ instance type which had total of 8 cores with 1.2 GHz CPU capacity and 15 GB main memory. The average RTT between any two sites was found to be 214 ms.

Replication configuration

We performed experiments for different numbers of sites. The number of partitions was set equal to the number of sites. Thus, for 10 sites, the system database consisted of 10 partitions, whereas for 20 sites the database was scaled accordingly to contain 20 partitions. In the experiments using MSI cluster we set the replication degree to 3, i.e. each partition was replicated on three sites. In case of experiments using Amazon EC2, due to relatively small number of geographic sites, we set the replication degree to 2. Each partition contained 100,000 items of 100 bytes each. For a partition, we designated one of its replica sites as the conflict resolver for items in that partition.

Transaction Workload

We synthesized a transaction workload consisting of two types of transactions: local transactions which accessed only local partitions, and non-local transactions which accessed some remote partitions from a randomly selected site. In the transaction workload, each transaction read from 2 partitions and modified 1 partition. In case of local transactions, the partitions to read/modify were randomly selected from the local partitions. For each accessed partition, the
transaction read/modified 4 randomly selected items. In subsection 4.5.4, we varied the number of modified partitions to evaluate its impact, as described later.

4.5.2 Scalability of P-CSI

We evaluated the P-CSI model for various system sizes, i.e. number of sites, to demonstrate its scalability under the scale-out model. For each system size, we measured the maximum transaction throughput and the average transaction latency. We performed this evaluation over both the local area MSI cluster as well as the wide-area cluster using Amazon EC2. In the evaluations using the MSI cluster, we evaluated the system with both the in-memory storage and HBase storage. In case of evaluations using EC2, we used the in-memory storage.

We also compare the performance of partial replication with the full replication scheme. For full replication scheme, the database contained equal number of items as that of the partial replication configuration, however each item was replicated on all sites, i.e. the replication degree was set the number of total sites. Note that this configuration corresponds to the basic CSI/PSI model. Thus, this evaluation also presents the comparative evaluation of CSI/PSI protocol and the P-CSI protocol.

Figures 4.6 and 4.7 show the transaction throughput data for systems using in-memory storage and HBase storage, respectively. This data corresponds the evaluation performed using the MSI cluster. Table 4.2 shows the throughput scalability data for experiments performed on Amazon EC2 using in-memory storage. For the figures we can see that P-CSI provides near-linear scalability; the maximum throughput achieved scales almost linearly with increase in number of sites. The throughput provided by full replication is much smaller compared to the throughput provided by partial replication. Moreover, in contrast to partial replication, full replication offers poor throughput scalability. For example, if we double the number of sites from 10 to 20, in full replication throughput increases only by a factor of roughly 1.29 for in memory storage and roughly 1.39 with HBase storage. The same for partial replication is 1.95 and 1.94, respectively. The throughput gain achieved in full replication diminishes even further if we scale the system to 30 sites.

We observe that the throughput provided by partial replication is higher than the throughput
provided by full replication approximately by a factor ranging between 2 to 4. This factor tends to be higher with increase in system size mainly due to poor scalability offered by full replication. One would expect that this factor to be close the ratio of replication degree of partial replication to the replication degree of full replication. For example, in case of 30 sites we would expect the throughput of partial replication with replication degree of 3 to be 10 times higher than the the throughput of full replication, which has replication degree of 30. However, the throughput is affected due to various other factors, such as two-phase commit overhead, overhead of executing read/write operations, etc., which induce same amount of overhead in both full and partial replication. Therefore, the relative throughput gain we observe in partial replication is lower than expected.
Table 4.2: System Performance on Amazon EC2 (Using InMemory Storage)

<table>
<thead>
<tr>
<th>Num. of Sites</th>
<th>Max Throughput (txns/sec)</th>
<th>Avg. Response Time</th>
<th>Avg. Visibility Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partial Replication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1938</td>
<td>198 ms</td>
<td>5.41 sec</td>
</tr>
<tr>
<td>8</td>
<td>3205</td>
<td>220 ms</td>
<td>5.46 sec</td>
</tr>
<tr>
<td>Full Replication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>768</td>
<td>318 ms</td>
<td>8.2 sec</td>
</tr>
<tr>
<td>8</td>
<td>938</td>
<td>370 ms</td>
<td>21.1 sec</td>
</tr>
</tbody>
</table>

Figure 4.8 shows the average transaction response times for partial and full replication schemes with in-memory and HBase storage. Table 4.2 shows the same for experiments performed on Amazon EC2 using in-memory storage. In case of partial replication, the response times remain roughly constant with increase in number of sites. The response times with HBase storage are higher than the response times with in-memory storage. The response times observed in case of Amazon EC2 experiments are higher due to higher communication latencies of wide-area network. The partial replication configuration provides lower response times compared to full replication. Moreover, in case of full replication, the response times typically increase with increase in the number of sites, indicating that full replication does not provide response time scalability.

**Impact of scale-out on Visibility Latency**

Another important question is whether increasing the system size, i.e number of sites in the system, has an impact on the amount of update delay and visibility latency. One of the advantages of the PCSI protocol is that the updates need to be propagated only to the sites containing the update partitions. This allows the system to be scaled to obtain almost-linear increase in transaction throughput, as demonstrated above. However, if scaling the system causes increase in the visibility latency, it can nullify the advantage of scaling the system. Thus, ideally one would expect that visibility latencies remain unaffected under scaling.

Figure 4.9 shows the average visibility latencies for above mentioned configurations. In full replication, visibility latency increases significantly with increase in system size due to the
increase in update propagation overhead. The same trend can be observed in geographic-scale replication using EC2, as shown by the data presented in Table 4.2. In contrast, the visibility latency remains roughly constant in partial replication for both cluster-based as well geo-scale replication. This is because the P-CSI protocol requires propagating updates only to the sites containing the modified partitions, and therefore, the visibility latency depends only on the workload characteristics and not on the system size. In summary, with partial replication one can achieve near-linear scalability without increase in the visibility latency.

4.5.3 Impact of Non-local Transactions

To evaluate the impact of locality in transaction execution, we induced non-local transactions, i.e. transactions with remote partition access. We varied the percentage of the non-local transactions to observe the impact on transaction latencies. Figure 4.10 shows the results of this evaluation.

We show in this figure the average latencies for all transactions as well as average latencies for non-local transactions. We observe only a slight increase in the overall latencies due to non-local transactions, however, these latencies can be higher in wide-area environments.
4.5.4 Impact of Transaction Size

As noted earlier, in partial replication, the cost of update propagation, i.e. the number of update propagation messages sent for a transaction, depends on the number of modified partitions $m$. Thus, if $n$ is the largest degree of replication for a partition, then the number of update propagation messages for a transaction modifying $m$ partitions is at most $m \cdot (n - 1)$. In Table 4.3 we show the number of update propagation messages per transaction observed in our experiments for different values of $m$. We also show in this table the average transaction latencies. The average transaction latency increases with $m$ mainly due to the increase in the latencies imposed by the 2PC protocol, since a transaction would need to coordinate with more number of sites with increase in $m$. This evaluation was conducted using 20 sites and 40 partitions on MSI cluster with in-memory storage.

4.5.5 Cost of Ensuring Causality

The primary cost of ensuring causality is associated with the delays that incur when applying a transaction’s update at a site that has not applied all the causally dependent transactions. We refer to these delays as update delay. In other words, the update delay corresponds to the
delay in making a transaction update visible at a site. Thus, increase in update delay at a site increases the visibility latency. We use these two metrics - update delay and visibility latency - to assess the cost of providing causal consistency.

Various factors have impact on the amount of update delay. One primary factor is the average number of causal dependencies for a transaction. The transaction that has large number of causal dependencies is likely to be delayed at a site for a longer time compared to transaction that causally depends on only few events. The number of causal dependencies for a transaction depends on the size of read/write set, i.e. the number of items read and modified by the transaction. Another factor that is likely to have impact on update delay is the eagerness of the update propagation mechanism, or more specifically the frequency with each updates are propagated. If updates are propagated lazily, i.e with low frequency, the sites are likely to be out of synch for longer time and the vector clock differences between sites are likely to be higher. This increases the likelihood that a given site is not advanced enough to apply updates of a
remote transaction. Furthermore, with low update propagation frequency, the rate at which the site's view of other site is advanced is also low, thereby increasing the delay in application of remote transactions’ updates. Note that, the increase in the update propagation interval also results in increase in visibility latency.

We experimentally evaluated the impact of various parameters on the update delay and visibility latency. These evaluations were performed using system size of 20 sites on MSI cluster with in-memory storage. For this evaluation, a moderate transaction load (60% the max throughput) was induced. Table 4.4 shows the impact of read and write sets size on the amount of update delay and visibility latency. This data shows that increase in read/write set size causes increase in update delay and visibility latency. This is mainly because increasing the number of items read and/or modified by a transaction increases the number of causal dependencies, thereby increasing the delays in application of transaction’s updates at remote sites.

We experimentally evaluated the impact of various parameters on the update delay and visibility latency. These evaluations were performed using system size of 20 sites on MSI cluster with in-memory storage. For this evaluation, a moderate transaction load (60% the max throughput) was induced. Table 4.4 shows the impact of read and write sets size on the amount of update delay and visibility latency. This data shows that increase in read/write set size causes increase in update delay and visibility latency. This is mainly because increasing the number of items read and/or modified by a transaction increases the number of causal dependencies, thereby increasing the delays in application of transaction’s updates at remote sites.

Table 4.4: Impact of Read/Write Set Size

<table>
<thead>
<tr>
<th>Read-Set Size</th>
<th>Write-Set Size</th>
<th>Average Update Delay (ms)</th>
<th>Average Visibility Latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>360</td>
<td>949</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>383</td>
<td>985</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>486</td>
<td>1110</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>577</td>
<td>1255</td>
</tr>
</tbody>
</table>

Table 4.5: Impact of Propagation Frequency

<table>
<thead>
<tr>
<th>Propagation Frequency</th>
<th>Average Update Delay (ms)</th>
<th>Average Visibility Latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 sec</td>
<td>383</td>
<td>985</td>
</tr>
<tr>
<td>2 sec</td>
<td>479</td>
<td>1980</td>
</tr>
<tr>
<td>5 sec</td>
<td>603</td>
<td>4305</td>
</tr>
</tbody>
</table>

Table 4.5 shows the impact of the update propagation frequency on the amount of update delay and visibility latency. In this table, the ‘Propagation Frequency’ column indicates the periodic interval at which updates are propagated to remote sites. As expected, both the update delay and visibility latency increase with the decrease in update propagation frequency.
(i.e. increase in the periodic interval). The increase in update delay is caused because with increase in update propagation interval the sites are out of sync with each other for a longer time, as explained above. The increase in visibility latency is due to two factors. One factor is the increase in update delay. The second factor is the delay in propagation of updates from the execution site.

### 4.6 Session Consistency Model

We discuss here various session level consistency guarantees and mechanisms to support them. A session is defined as a sequence of transactions executed by a user (or application client). During a session the user can move geographically and may connect to different sites. For supporting mobility, the session model provides primitives for suspending a session at a site and resuming it later at a different site.

Our session consistency model supports the following four session guarantees, which were first proposed in [52].

- **Read-Your-Writes (RYW):** A read operation on a data item must read a version that is equal to or newer than the version created by the most recent write operation on that item performed in the session.

- **Monotonic-Read (MR):** A read operation on a data item must never see a version of that item which is older than the version seen by any previous read operation on that item performed in the session.

- **Monotonic-Writes (MW):** A write operation on a data item follows any preceding writes on that item performed in the session, i.e. all the writes performed on a particular data item in the session are serialized.

- **Write-Follows-Reads (WFR):** A write operation on a data item is performed on a version that is equal to or newer than the versions seen by the preceding read operations performed on that item.

The MW and WFR guarantees are supported, not only on a session level but across the entire
system, through the \textit{ww} conflict checking protocol of the basic SI model. If a transaction attempts to write a data item and if the latest version of that item is not visible in its snapshot then the transaction is not allowed to commit. Thus, a committed transaction updating an item always sees the latest version of that item. The RYW and MR guarantees are supported automatically when the user is connected to a single site during the entire session. However, these guarantees can be violated if the user connects to a different site and the updates seen at the previous site have not been propagated there. This is particularly the case for mobile users, since such users may connect to different sites while they are travelling. Even in case of a non-mobile user, he/she may be redirected to a different site if the local site is unavailable or overloaded. We discuss below how the RYW and MR guarantees can be supported in such cases.

Our basic model for supporting session guarantees is as follow. For a session, we maintain the \textit{session state} as the events (i.e. transactions) seen by the session. In other words, the session state indicates the updates that must be visible at a site in order to support the session guarantees at that site. An important question here is how to capture this session state efficiently. One approach is to maintain, for each read/write operation performed in the session, the information about the data item versions read/written. However, this may require maintaining a lot of information as the session may read/write large number of items. A more efficient way is to maintain the session state in form of a vector clock, called \textit{session vector clock} in which each element corresponds to a site and identifies the sequence number of the latest event corresponding to that site that must be visible in order to support the required session guarantees. Recall that, our update propagation model guarantees that if a site has applied a particular transaction from another site it must have applied all the preceding transactions from that site.

A simple approach to calculate the session state vector is to consider the current vector clock value of the site where the session is executing. The vector clock value of the session’s current site would capture all the events seen by the session. However, this can also induce a lot of false dependencies, i.e. the events not seen by the session. A more precise approach is to consider only the events seen by the session. This includes updates seen by the read operations and updates performed during the session. Basically, the session vector clock value can be computed in the same way as the effective causal snapshot but considering all the data versions seen by
the entire session and not just a single transaction. The session vector clock value is maintained
for each partition accessed during the session.

When a user migrates to some other site the vector clocks values at that site are examined
to check whether the site has advanced enough, i.e. it has applied the events identified in the
session vector clock. If the site is not enough up to date, then we need additional mechanisms
to provide session guarantees. A simple approach is to forward all read/write requests to the
previous site. We call this tunneling. This approach is obviously expensive as each operation
would incur significant delays. Another simple approach is to redirect the client to another
site whose vector clock is advanced enough to support the session guarantees. However, this
requires sites to have the knowledge of the vector clock values of other sites. For this purpose,
sites periodically exchange their current vector clock values. For efficiency, this information is
piggy-backed with the update propagation. Using this information, the new site determines a
candidate site for redirection. It may be possible that the new site could not find any candidate
site for redirection. This can happen either because the site has stale view of other sites' vector
clock values or there is no site (other than the original site of the session) which is advanced
enough. Another problem with redirection approach is that it may lose certain benefits such as
site proximity.

A more ideal solution is to let a site pull the required updates to advance itself up to the
session vector clock, so that the session can be resumed at that site. To do this, the site first
determines the updates that need to be pulled, and then contacts the corresponding sites and
pulls the required updates. While pulling the updates, the site may also need to pull any
causally preceding updates if they are not already received by the site. The delay incurred in
resuming the session depends on the number of updates to be pulled and latencies between sites.
For minimizing such delays, we also provide a primitive for the user to specify one ore more
preferred sites while suspending the session. In this case, the original site eagerly propagates
updates to the preferred sites while the user is migrating.
4.7 Summary

In this chapter, we have presented a transaction management model for geo-replicated databases. Since providing strong consistency for geo-replicated data is not practical, we have provided a weaker yet useful form of Snapshot Isolation, called \textit{Causal Snapshot Isolation (CSI)}. The CSI model is targeted for databases using full replication and provides causal ordering of transactions across geo-replicated sites. Our protocol for ensuring causal consistency avoids \textit{false causal dependencies} by considering only the items read/written by a transaction when determining its causal dependencies. Building upon the CSI model, we have presented a model for partially replicated databases, called \textit{Partitioned-CSI (P-CSI)}. The P-CSI protocol addresses the unique issues raised due to partial replication in supporting causal consistency and snapshot isolation guarantees. The P-CSI protocol efficiently supports both these guarantees and requires communicating updates only to the sites storing copies of the updated items. Through experimental evaluation we have demonstrated the effectiveness of this protocol. Additionally, we also discuss the mechanisms for supporting \textit{session consistency} guarantees in this transaction model.
Chapter 5

Conclusion and Future Directions

In this thesis, we have addressed the problem of providing general transaction support and appropriate consistency models on NoSQL data management systems. Transaction primitives with ACID guarantees are required in many cloud applications. The lack of appropriate transaction and data consistency models in NoSQL systems has been an obstacle in the adoption of these systems. Furthermore, it also puts a burden on the application programmers to reason about consistency of their applications and to program the applications such that they behave in a consistent manner in the absence of any useful consistency guarantees by the underlying data storage system. In this regard, the goal of this thesis has been to develop transaction management models with appropriate consistency guarantees that can be useful for cloud applications.

We have presented in this thesis efficient transaction management models and techniques for NoSQL systems deployed in cluster environments as well geo-replicated environments. The snapshot isolation (SI) model of transaction execution forms the basis of our work. Based on the SI model, we provide a spectrum of consistency models that includes basic snapshot isolation, serializable snapshot isolation and causal snapshot isolation.

The NoSQL system have dropped support for transactions as it was deemed impractical due to scalability and availability considerations. Contrary to conventional understanding, we demonstrated that providing transaction support for NoSQL systems without compromising the scalability and availability properties is possible. The crucial aspect of our approach was the
notion of decoupling transactions management functions from the storage system and integrate with the application layer. This provides a natural way of scaling the transaction workload by scaling-out the application layer without affecting the scalability or availability of the storage layer. We have presented two architecture models based on this notion - a fully decentralized architecture and a service-based architecture where conflict detection functions are performed by a dedicated service. To ensure the availability and scalability of this service, we also presented a design for its replication and fault-tolerance. Through experimental evaluations, we demonstrate that both these architecture are scalable under the scale-out model. We show the scalability over a large cluster (up to 100 nodes). Conceptually, the decentralized architecture does not have any scalability bottleneck as it does not rely on any centralized component. In terms of transaction throughput and response times, the service-based architecture performs better than the decentralized approach, however the saturation throughput of the system mainly depends on the saturation throughput of the conflict detection service. For TPC-C benchmark, with system size of close to 100 nodes and a single conflict detection server, we could achieve throughput of roughly 5000 requests per second without the service becoming the bottleneck. We have presented a design for replication of this service for its scalability and fault-tolerance. With the replicated service using 8 conflict detection servers, we could achieve throughput of roughly 24,000 requests per second. Thus, with 8 servers the replicated conflict detection service can support the workload requirement of a transaction processing system of roughly 500 nodes. This indicates that only a small number of conflict detection servers are required to support the workload requirement of a large transaction processing system, or in other words, only a small percentage of cluster resources are required to be dedicated to the conflict detection service.

Since the basic SI model does not guarantee serializability, we investigated two approaches for ensuring serializability. The first approach called as cycle prevention was based on preventing certain conditions that lead to serialization anomalies. The second approach called cycle detection was based on explicitly detecting cycles and aborting a transaction to break the cycle. The cycle detection approach aborts only the transaction leading to serialization anomalies, whereas The cycle prevention approach can sometimes lead to unnecessary aborts. Through experimental evaluations, we found that the cycle prevention approach is more efficient than the cycle detection approach, both in terms of throughput and response times. This result suggests
that with respect to concurrency control approaches, an approach that is simpler and incurs less overhead is generally better.

In case of geo-replication, providing serializable transactions with strong data consistency is not practical because of high communication latencies in wide-area networks. Therefore, our goal was to develop a weaker but useful transaction consistency model that can be supported in wide-area environment without compromising the requirements of high performance. Towards this goal, we developed a model called as *Causal Snapshot Isolation* for transaction management in geo-replicated systems. This model provides SI based transactions with causal consistency for database replicated across multiple geographically distributed sites or datacenters. We further extended this model to develop a model, called as Partitioned-CSI (P-CSI), to support transactions for partially replicated databases. Partial replication raises unique challenges with respect to supporting snapshot isolation and causal consistency. Addressing these challenges, we developed an efficient transaction management protocol that requires communicating the transaction updates only to the sites storing the updated items. We also presented mechanisms to support session consistency guarantees under this model. We experimentally demonstrated the scalability and performance efficiency of the CSI model. The important contribution of this work is that it provides a transaction management model that does not require synchronous replication and yet provides a useful consistency model for geo-replicated applications.

There are various interesting future research directions based on the work presented in this thesis. In case of CSI model, an interesting direction for further research is to use operation commutativity to minimize remote coordination and reduce transaction aborts due to *write-write* conflicts. The CSI model requires communication with conflict resolver sites to check for *write-write* conflicts. In certain cases, two operations updating the same data item can commute. For example, two addition operations adding a certain amount to a numerical data item value are commutative. Any order of application of these two operations results in the same final value of the data item and therefore they can be applied in any order. Thus, operation commutativity can be exploited to reduce transaction aborts since two transactions executing commutative operations on same data items do not conflict with each other. This can also help in minimizing the remote coordination required for checking *write-write* conflicts. However, this approach requires further investigation to examine various issues that can arise in exploiting
operation commutativity. One issue is related to multi-version data management. If commutative operations are applied in different order at different sites storing the copy of the data item, then this can lead to different sites having different timelines for data item versions. This factor needs to be taken into account when managing versions and applying data updates from remote sites. This also raises the issue with respect to observing a correct and consistent snapshot for transaction execution. Another issue is related to concurrent execution of commutative operations with non-commutative operations. In such scenarios, the sites need to agree upon the correct order of non-commutative operations.

In this thesis, we developed different consistency models providing varying levels of consistency guarantees. These models provide different trade-offs between consistency guarantees and system performance properties such as scalability or operation latency. In this regard, an interesting problem for future work is how to simultaneously support different consistency models. Such a mixed mode consistency model can be beneficial in various cases so that applications can efficiently trade-off between consistency and performance. In certain cases, an application may require strong consistency guarantees only for certain critical data items and it can tolerate a weaker consistency model for other data items. For example, an online shopping application may require strong consistency for financial data whereas it could tolerate weaker consistency for product catalogs or inventory information. A mixed mode consistency model can also be used when different applications or user groups using the same database require different consistency levels. One way in which such a mixed mode consistency model can be supported is by partitioning data into different partitions based on their consistency requirements. However, this approach would require further investigation to develop an efficient transaction execution model to support such mixed mode consistency model.
References


isolation under the hood,” in IEEE ICDE’11, april 2011, pp. 625 –636.


A. Fikes, and R. E. Gruber, “Bigtable: A Distributed Storage System for Structured Data,”

sen, N. Puz, D. Weaver, and R. Yerneni, “Pnuts: Yahoo!’s hosted data serving platform,”

A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd,
C. Taylor, R. Wang, and D. Woodford, “Spanner: Google’s Globally-Distributed Database,”


163–174.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Siva-
subramanian, P. Vosshall, and W. Vogels, “Dynamo: Amazon’s highly available key-value

shot isolation,” in 24th IEEE Symposium on Reliable Distributed Systems, 2005, pp. 73 –
84.


