Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions

by

Atul Adya

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

March 1999

Author . . . .
Department of Electrical Engineering and Computer Science
March 18, 1999

Certified by . . . .
Barbara H. Liskov
Ford Professor of Engineering
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions

by

Atul Adya

Submitted to the Department of Electrical Engineering and Computer Science
on March 18, 1999, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Current commercial databases allow application programmers to trade off consistency for performance. However, existing definitions of weak consistency levels are either imprecise or they disallow efficient implementation techniques such as optimism. Ruling out these techniques is especially unfortunate because commercial databases support optimistic mechanisms. Furthermore, optimism is likely to be the implementation technique of choice in the geographically distributed and mobile systems of the future.

This thesis presents the first implementation-independent specifications of existing ANSI isolation levels and a number of levels that are widely used in commercial systems, e.g., Cursor Stability, Snapshot Isolation. It also specifies a variety of guarantees for predicate-based operations in an implementation-independent manner. Two new levels are defined that provide useful consistency guarantees to application writers; one is the weakest level that ensures consistent reads, while the other captures some useful consistency properties provided by pessimistic implementations. We use a graph-based approach to define different isolation levels in a simple and intuitive manner.

The thesis describes new implementation techniques for supporting different weak consistency levels in distributed client-server environments. The mechanisms are based on optimism and make use of multipart timestamps. A new technique is presented that allows multipart timestamps to scale well with the number of clients and servers in our system; the technique takes advantage of loosely synchronized clocks for removing old information in multipart timestamps.

This thesis also presents the results of a simulation study to evaluate the performance of our optimistic schemes in data-shipping client-server systems The results show that the cost of providing serializability relative to mechanisms that provide lower consistency guarantees is negligible for low-contention workloads; furthermore, even for workloads with moderate to high-contention workloads, the cost of serializability is low. The simulation study also shows that our mechanisms based on multipart timestamps impose very low CPU, memory, and network costs while providing strong consistency guarantees to read-only and executing transactions.

Thesis Supervisor: Barbara H. Liskov
Title: Ford Professor of Engineering

Acknowledgments

I would like to thank my research advisor, Barbara Liskov, for her constant support and counsel during my stay as a graduate student. I have learned many principles on performing good research from her, especially about combining theoretical aspects of system design with practical implementations.

My thesis committee members made several excellent suggestions for improving the content and presentation of this work. John Guttag and John Chapin gave helpful suggestions for clarifying different parts of this thesis. Jim Gray provided extremely useful feedback about current database systems that helped me address a number of important issues for defining weak consistency levels.

A number of people contributed in making my graduate life at MIT an enjoyable experience. If I am an improved computer scientist and a better person, it is because of the companionship and advice of my friends and colleagues. They have had a positive influence on me and enriched my life in many ways.

My colleagues at the Programming Methodology Group provided a stimulating environment for technical discussions. I have learned many interesting aspects of computer systems from my interactions with Andrew Myers and Miguel Castro. My discussions with Phillip Bogle, Chandrasekar Boyapati, Mark Day, Robert Gruber, Sanjay Ghemawat, Umesh Maheshwari, and Quinton Zondervan have also been beneficial for me. Dorothy Curtis and Paul Johnson have helped me on numerous occasions with equipment and software. Kavita Bala, Radhika Nagpal, and every group member was always willing to attend my practice talks and provide feedback for improving my presentations.

I will always cherish my pleasant and light-hearted experiences with the PM Group members. I have enjoyed planning movie-outings and solving puzzles with Andrew Myers, talking about Star Wars with Jason Hunter, exchanging good-natured jabs with Arvind Parthasarathi and Doug Wyatt, and having numerous intellectual conversations with Chandrasekar Boyapati, Umesh Maheshwari, Quinton Zondervan, and others in the group.

Phil Bogle and Sudhendu Rai have been a constant source of encouragement and inspiration for me. Sudhendu's advice on research and philosophy have played a significant role in improving my outlook towards life. Ujjwal Sinha and Ananda Sen Gupta have always been present to help me whenever I needed support. They were wonderful companions and I have had fun watching movies, talking, and going out for short trips with them. Aman Rustagi, Sreeram, and Sandeep Gupta are valuable friends who have always wished well and I have enjoyed the time that I spent with them.

Words cannot even come close to expressing my gratitude to my parents who deserve the credit for whatever positive that I have achieved in my life. They have always supported me in all my endeavors and have patiently waited for the completion of my doctorate. Last but not the least, my wife, Vandana, has been very supportive and encouraging while I have been trying to finish up my thesis.

Contents

Chapter 1

Introduction

This thesis is concerned with providing good performance along with strong semantic guarantees for atomic transactions in a database system. Databases use transactions to ensure that computations transform the system from one consistent state to another in spite of concurrency and failures. To allow programmers to reason about their code in the presence of concurrency, the notion of serializability is provided by many databases, i.e., even though transactions run simultaneously, it seems as if they execute in some sequential order. However, over the years there has been a great deal of interest in weak consistency levels that provide guarantees weaker than serializability to applications. Many weak consistency levels have been proposed in the database literature and all commercial databases allow transactions to run with weaker consistency guarantees; in fact, some systems do not even support serializability.

This thesis makes a number of contributions in the area of weak consistency:

- It presents the first implementation-independent definitions of weak consistency levels that are widely used in commercial database systems. Our definitions also handle predicate-based operations correctly at all consistency levels in an implementation-independent manner. The previous definitions are either incorrect and allow bad behaviors, or are not sufficiently flexible and rule out correct behaviors that might be produced by real concurrency control implementations such as optimism [AGLM95, BOS91, BBG$^+$95, KR81, ABGS87, FCL97]. Our specifications overcome these difficulties and are flexible enough to allow a wide range of concurrency control techniques.

- It specifies two new levels that provide useful consistency guarantees to application writers. One of the new levels, PL-2+, is the *weakest* level that ensures consistent reads; the other level, PL-2L, captures a useful monotonicity property.

- It also presents implementation-independent specifications of a number of consistency levels that are commonly used in commercial databases, e.g., Cursor Stability, Snapshot Isolation [Ora95]. Earlier definitions were either informal or based on locking implementations.

- It presents new implementation techniques for supporting different weak consistency levels in client-server distributed environments. Our protocols are based on optimism and take advantage of the system structure to offload work from servers to clients thereby making the system more scalable. Some of our schemes make use of multipart timestamps; we describe a new technique that allows multipart timestamps to scale well with the number of clients and servers in our system.

- It also evaluates the performance of our consistency schemes via simulation. Our results show that implementations that provide strong consistency guarantees such as serializability need not have high performance penalties compared to schemes that provide weak guarantees.

1.1 Why Weak Consistency Levels are Useful

Weak consistency levels have been of interest over the years for two reasons: they are useful for certain applications, and they can be implemented more efficiently than stronger levels thereby allowing applications to achieve a higher throughput.

Implementations of weak levels have been considered primarily in centralized systems using pessimistic approaches such as locking. Weak consistency levels are desirable in such systems because transactions either acquire fewer locks, or hold them for shorter periods of time. In either case, the result is less delay, since fewer transactions attempt to access the same objects in conflicting modes; additionally, the possibility of deadlock is reduced. We also expect these performance benefits to be important in the wide-area distributed systems and in mobile systems of the future. In these environments, optimistic concurrency control mechanisms appear to be better than pessimistic ones. However, irrespective of the type of concurrency control mechanism used, weaker levels are advantageous in many circumstances. Again, the advantage of weaker consistency levels is that there are fewer conflicts; this can lead to reduced communication, fewer aborts (in an optimistic system), or fewer delays (in a pessimistic system). Thus, it is desirable to allow application programmers to take advantage of weaker levels (when this makes sense) and trade off consistency for better performance.

Applications that can be executed at lower degrees of consistency must be written taking program semantics into account. There are two kinds of applications that can run below serializability. In the first kind, the program writer is aware that certain kinds of conflicts will not occur. Such programs can be executed at a consistency level that need not provide guarantees with respect to these kinds of conflicts. Thus, even though the transaction is executed at a level below serializability, it is still serializable. Here is an example.

Suppose that a brokerage firm provides advice to its preferred clients about whether the prices of stocks of a few companies are expected to rise or fall. An analyzer transaction T reads the stock data of these companies for the past few weeks, performs an analysis of the data, writes the results to a report log, and sends email to the clients. In this case, the brokerage firm knows that the stock data will not be modified by any application and transaction T updates a private part of the database (only T generates the report and has exclusive access to this region). Thus, T can be executed at a consistency level below serializability; T will still be serializable because of its access patterns. If the overheads of providing serializability are higher than the overheads of a weaker consistency level implementation, the analyzer transaction can achieve better performance by executing at a lower level.

In the second class of applications that can be executed below serializability, the program writer handles inconsistent reads and writes in the code itself. Here are a few examples of applications that follow this style:

- **Programs that read approximate or non-serializable information:** In this class of applications, reading an approximate (and inconsistent) state of the database is sufficient. For example, a manager in a grocery store may browse the database to determine the approximate earnings achieved since the morning.

- **Programs based on weak invariants:** Some applications only need to observe a consistent state of the database; they are programmed such that they require fewer constraints between multiple objects. For example, suppose that a bank considers a customer to be in good standing if the allowed credit is less than the customer's current bank balance. Suppose that a transaction T_i increases a customer's credit limit to the current bank balance. In a later transaction T_j, the customer deposits money in his account, i.e., T_i is ordered before T_j. If a bank transaction T_b observes the new balance and the old credit limit, the customer will be considered in good standing. Since T_b reads T_j's updates but does not observe T_i's modifications, it must be ordered before T_i and after T_j, which is not possible. Thus, in this scenario, transactions T_i, T_j, and T_b cannot be serialized even though T_b observes a consistent database state. Thus, the bank application could be written based on requirements weaker than serializability and still function correctly.

- **Programs that check for violated invariants:** Some programs observe broken invariants but the application programmer writes the code to take these inconsistencies into account. For example, in a producer-consumer scenario, suppose that a consumer transaction S removes all elements from a list containing 10 elements. Transaction S reads the number of entries, processes each entry and removes it. In the meanwhile, another consumer transaction T removes 3 elements from the list. After iterating over 7 elements, transaction S receives an exception that the list is empty. If S has been coded to terminate early, it can be executed at a lower consistency level. However, if it is written assuming that the list contains exactly 10 elements, serializability would be needed.

- **False conflicts:** False conflicts occur due to grouping of information. Different transactions may read/write different fields of an object and not conflict at all; such transactions are not serializable when we consider concurrency control based on objects but are serializable if we treat the fields as separate objects.

  Such grouping of information may be done for a variety of reasons. First, keeping track of individual fields may increase concurrency control overheads substantially. In a locking scheme, more locks have to be acquired, released, and kept track of; in an optimistic scheme, more information has to be sent to servers, resulting in higher network and server CPU overheads. Second, grouping of attributes into one object may be done for space considerations. For example, an age attribute may be an integer in a personal record of an employee; making a separate object or tuple for age will increase space overheads. Finally, grouping of objects may be done for better abstraction properties, e.g., keeping complete information about a person's medical information in one tuple/object is preferable compared to spreading the data over many objects.

The above examples demonstrate that weak consistency levels are sufficient for a large class of applications. Thus, if weaker levels can be supported more efficiently than higher levels, it will be worthwhile to execute applications below serializability. However, an important drawback of weak consistency guarantees is that it is much more difficult to reason about an application's correctness: the possibility of destroying database integrity can increase significantly compared to serializability; an application writer has to be fully aware of conflicts of the program code with other transactions. Thus, unless serializability costs are prohibitive, we believe that transactions that modify the database should not be executed at low consistency levels.

1.2 Why New Consistency Definitions are Needed

Any set of definitions for weak consistency levels must satisfy two goals. First, they must be sufficiently restrictive to disallow all behavior that is considered undesirable by application programmers and end-users, e.g., non-serializable histories should be disallowed by the consistency level that provides serializability. Second, they must also be permissive enough to allow all good behavior that is expected by applications, or at least all histories that can occur using some realistic concurrency control technique. In particular, we would like to allow both pessimistic and optimistic concurrency control approaches.

Pessimistic schemes such as 2-phase locking require appropriate permission (e.g., read or write) to be obtained before an object can be accessed. On the other hand, optimistic concurrency control schemes allow immediate access to objects. At the end of a transaction, the database system checks for conflicts and, the transaction is aborted if necessary.

Most systems in the past have used locking to provide concurrency control. However, it is important to have definitions of consistency levels that allow other concurrency control implementations such as optimism and multi-version schemes because such techniques may perform better in some environments. For example, optimism appears to be a good approach for wide-area distributed systems and systems with disconnected nodes [GKLS94, KS91, TTP$^+$95]. Furthermore, it is imperative that consistency definitions allow optimistic mechanisms since commercial databases provide different consistency levels using such schemes. For example, Gemstone [BOS91] provides serializability using a multi-version optimistic concurrency control scheme.

There have been three attempts in the past to specify weak consistency levels. We now discuss why the goals of correctness and flexibility have not been met by them; they are discussed in more detail in Chapter 2.

The notion of weak consistency levels in database systems was first introduced in [GLPT76]. These levels are also referred to as *degrees of isolation* in [GR93]. The degrees are numbered from 0 to 3, where degree 3 is the same as serializability. The levels proposed in [GLPT76] are based on notions of locking rather than being independent of a concurrency control technique. Further refinements to the original isolation levels along with new levels such as Cursor Stability were introduced in [Dat90].

The second set of consistency specifications, the ANSI/ISO SQL-92 definitions [ANS92], had the goal of coming up with an industry standard that was implementation-independent. In particular, the aim was to allow different concurrency control schemes such as locking and optimism. The ANSI specifications are widely used in database products [IBM99, Ora95]. The ANSI definitions were based on the work in [GLPT76] and [Dat90] and are stated as a set of *phenomena* that different isolation levels were intended to exclude.

A subsequent paper [BBG$^+$95] showed that the ANSI-SQL definitions were ambiguous and could be interpreted to allow histories that result in inconsistencies. They proposed a third set of consistency definitions by proposing a different set of phenomena that was correct and precise. However, the new definitions were simply equivalent to locking, as was shown in [BBG$^+$95]. Therefore, the definitions in [BBG$^+$95] failed to meet the goals of ANSI-SQL with respect to implementation independence.

Thus, at present, there is a real need for new specifications of weak consistency levels that are correct and yet flexible enough to allow a variety of concurrency control techniques.

1.3 Contributions: Specifying Degrees of Isolation

Our first set of contributions is in the area of redefining and extending isolation levels to permit a wide range of concurrency control implementations. We present new definitions that capture the intent of the ANSI/SQL properties, yet allow a wide variety of concurrency control techniques including optimistic and multi-version schemes. Our specifications have the following important attributes:

Implementation-Independence: They allow a range of concurrency control mechanisms including locking, optimism and multi-version schemes. Our definitions are complete (they allow all good histories); in particular, they provide conflict-serializability [BHG87]. It is difficult to prove completeness for lower isolation levels, but we show that our definitions are more permissive than those given in [BBG$^+$95].

Correctness: Our definitions for PL-3 rule out all non-serializable histories since they provide conflict-serializability. It is difficult to prove correctness for lower levels since well-defined requirements of these levels have not been specified in the past. However, situations described as undesirable in the literature are disallowed by our conditions.

Intuitive and Backwards Compatible: Our specifications capture the essence of the ANSI specifications and are similar to the existing definitions making it easy for programmers to understand them.

Commercial Applicability: Our specifications characterize a large number of isolation levels that are in common use by commercial DBMS products, e.g., Cursor Stability, Snapshot Isolation used in the Oracle server.

Flexibility for Predicate-based Operations: We specify a variety of guarantees that can be provided to predicate-based operations at weak consistency levels in an implementation-independent manner; a database system can choose the guarantees that it wants to support at each consistency level. Earlier definitions for these operations were either incomplete, ambiguous, or specified in terms of an implementation such as locking or in terms of a particular database language such as SQL.

Our definitions are based on the observation that any set of consistency specifications must not allow transactions to observe violations of *multi-object constraints*; these are invariants of the type $x + y = 10$ that involve multiple objects. The approach suggested in [BBG$^+$95] captures multi-object constraints by disallowing conflicting operations to run concurrently on individual objects, i.e., the conditions are specified in terms of single-object histories. However, optimistic schemes allow conflicting operations to execute simultaneously and still correctly preserve multi-object constraints; the reason is that the consistency checks take *all* objects that were accessed by the committing transaction into account. Thus, any consistency definition that tries to capture such constraints using a fixed number of objects and transactions will be either incorrect or overly restrictive and disallow valid histories, i.e., consistency levels must be specified by considering all accesses of a transaction.

We have achieved this goal by using a combination of constraints on object histories and serialization graphs [BHG87]. These graphs provide a simple way of capturing multi-object constraints. Each node in this graph corresponds to some committed transaction T_i and edges are added between nodes corresponding to every read and write performed by T_i. Some of our conditions are specified in terms of the different types of cycles (based on different types of conflicts) that must be disallowed in these graphs, e.g., serializability disallows all types of cycles whereas the lowest isolation level disallows only cycles involving updates (and not reads). We also use graphs for defining correctness conditions in mixed systems in which different transactions may commit at different isolation levels.

Specifying consistency conditions using graphs and different types of conflicts/dependencies is a well-known technique and has been used in the literature for specifying serializability [BHG87, KSS97, GR93], semantics-based correctness criteria [AAS93], and for defining extended transaction models [CR94]. Our approach is the first that applies these techniques to defining ANSI and commercial isolation levels in an implementation-independent manner.

In [GLPT76], it has been shown that locking-based definitions of weak consistency levels can also be expressed in terms of constraints on graphs. Since these graph-based definitions are intended to be equivalent to locking, they are not as flexible as our specifications. Furthermore, those definitions do not consider predicate-based operations and also lack some conditions that are required for correctness.

Another property of our isolation definitions is that they allow an application to request different isolation guarantees for committed and running transactions. This characteristic provides more flexibility to system builders and allows efficient implementations for providing various consistency levels. We show our approach in the form of a graph in Figure 1-1. The graph shows guarantees provided for committed transactions on the Y-axis and guarantees for running transactions on the X-axis. Our specifications permit any scheme in the X-Y plane. The approach in [BBG$^+$95] allows only schemes that are on the diagonal because their specifications require that a concurrency control scheme provide the same guarantees for running and committed transactions (a lock-based implementation does indeed have this property). Thus, we have extended the isolation specification space from a single line to a two-dimensional grid. This flexibility ensures that a wide range of concurrency control mechanisms are permitted by our isolation specifications. For example, CLOCC [AGLM95, Gru97] and CLOCC with EPL-2+ (discussed in Chapter 5) are disallowed by the definitions in [BBG$^+$95] (since they do not lie on the diagonal) but are permitted by our specifications.

1.3.1 Definitions of Existing Isolation Levels

We have arrived at our new specifications for weak consistency levels by understanding the motivation of the original definitions [GLPT76] and the problems that were addressed in [BBG$^+$95]. Our isolation conditions capture the essence of the ANSI specifications [ANS92]; for each ANSI-

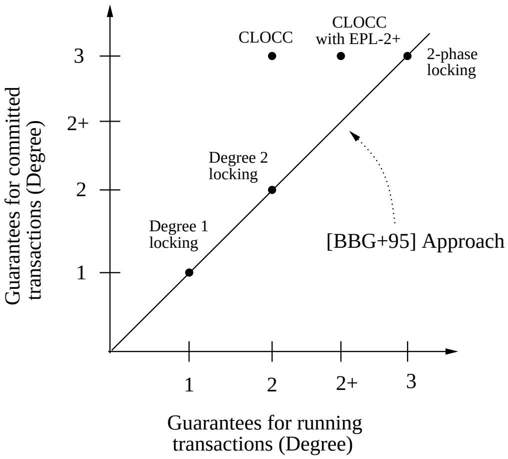


Figure 1-1: Separate guarantees for running and committed transactions as provided by our isolation definitions allows schemes such as CLOCC.

SQL degree, we have developed a corresponding *portable isolation level* that is precise and implementation-independent. Our levels for committed transactions are called PL-1, PL-2, and PL-3 where PL-3 is the same as serializability.

Along with the levels specified by ANSI, we present definitions of existing commercial levels such as Cursor Stability [Dat90] Snapshot Isolation [BBG$^+$95], and Oracle's Read Consistency [Ora95]. Unlike earlier definitions, our specifications are implementation-independent. We specify these levels by extending the graphs used for defining the ANSI levels; different types of nodes and edges are added to capture the constraints relevant to each level. These definitions demonstrate that the graph-based approach for specifying isolation levels is flexible; as new degrees are developed in the future, graphs can be used to specify them.

1.3.2 New Isolation Levels

There is a wide gap between degree 2 (which does not provide consistent reads or writes) and degree 3 (which provides serializability). We have developed two new and useful levels between degree 2 and degree 3 and related them to existing commercial consistency guarantees.

Our first level, PL-2+, is the *weakest* level that ensures that transactions do not observe violated multi-object constraints. However, it allows transactions to update the database in an inconsistent manner. Thus, PL-2+ lies "halfway" between degrees 2 and 3 since it ensures consistent reads but allows inconsistent writes. Level PL-2+ ensures that a transaction is placed after all transactions that causally affect it, i.e., it provides a notion of "causal consistency". This level disallows all phenomena that Snapshot Isolation was intended to disallow. Since PL-2+ is weaker than Snapshot Isolation, it has the potential of being implemented more efficiently, especially in a distributed client-server system (an efficient optimistic scheme for providing PL-2+ in such environments is presented in Chapter 5). Thus, PL-2+ may be preferable to Snapshot Isolation.

Our second new level, PL-2L, captures useful properties of a lock-based implementation of degree 2. It ensures that a transaction observes a monotonically increasing prefix of the database history as it executes, e.g., in an online auction system, if a transaction closes the auction to sell a product and a user transaction observes this closure and then the value of the product, PL-2L will ensure that the user observes the final value of the product. PL-2L can be useful for legacy applications that execute at degree 2 and assume such monotonicity properties are provided by a locking implementation; when the system is changed from locking to a different concurrency control mechanism, PL-2L can be used to ensure that these applications continue to run correctly. An interesting observation about PL-2L is that it is similar to the Read Consistency guarantees provided by the Oracle server [Ora95]. Level PL-2L disallows phenomena that Read Consistency was intended to disallow. Since PL-2L is weaker than Read Consistency, it may be less expensive to provide PL-2L than Read Consistency, especially in distributed client-server systems.

1.4 Contributions: New Implementation Techniques

Our second set of contributions addresses the problem of providing high-performance consistency mechanisms in distributed client-server systems. In such systems, objects are stored at servers and clients execute transactions on cached copies of their machines. This architecture is desirable because a large fraction of the application's work can be offloaded from servers to clients, thereby enhancing the scalability of the system. It is important to develop efficient consistency mechanisms for such systems since they are expected to be common in the future.

We have designed optimistic schemes to support new and existing isolation levels in distributed client-server systems. Optimistic mechanisms are appealing in client-caching systems since they allow clients to execute transactions using cached information without extra communication with the servers; lock-based schemes may require such communication when an object has to be modified. This intuition has been borne out by Gruber's work [Gru97], which shows that an optimistic scheme called CLOCC or *Clock-based Lazy Optimistic Concurrency Control* (CLOCC was earlier referred to as AOCC) outperforms the best-known locking implementation for client-server systems across a wide range of workloads and system parameters: this research also shows that CLOCC scales well with the number of clients. An additional advantage of optimism is that it can be easily applied to disconnected and mobile environments [KS91, TTP$^+$95, GKLS94].

However, Gruber's work studied only serializability in client-server systems. Efficient and scalable optimistic schemes for lower consistency levels in such environments do not exist. Thus, the first challenge is to design efficient consistency techniques for these levels with low communication and server overheads. Furthermore, high-performance optimistic schemes such as CLOCC provide strong serializability guarantees for committed transactions but provide very weak guarantees as transactions execute. These guarantees are important since application writers may expect certain integrity constraints to be valid while a transaction executes; otherwise, the application may behave in an unexpected manner (e.g., crash or go into an infinite loop). Optimistic schemes such as O2PL [FCL97] provide strong guarantees for running transactions but they do so at a high communication cost (to the extent that they lose the advantages of optimism over locking). Thus, another challenge for client-server systems is to provide strong consistency guarantees to transactions as they execute yet ensure that the cost of these mechanisms is low.

Multistamp-based Mechanisms and Multistamp Truncation

We have developed efficient implementations for providing different isolation guarantees to transactions in distributed client-server systems. We have taken advantage of the system structure and also utilized characteristics of real systems to optimize our mechanisms, e.g., we use loosely synchronized clocks to truncate some of our data structures.

Our isolation levels have been defined in terms of different types of conflicts and we use multipart timestamps or *multistamps* to capture these consistency constraints. Multistamps are propagated to clients to warn them of potential violations of consistency; if a client does not have information as indicated by the multistamp, it communicates with the relevant servers. Clients act on the multistamp information only if it might affect the current transaction. Being lazy buys time so that the needed consistency information is highly likely to be present by the time it is needed. Furthermore, it allows us to piggyback most of the information on existing messages in the system thereby reducing the overheads of our schemes.

A negative aspect of multistamps is that they do not scale well with a large number of clients and servers; mechanisms that send multistamps incrementally [BSS91] still require complete multistamps to be stored at different servers. Instead, we have devised a novel mechanism called *multistamp truncation* that keeps them small; this technique takes advantage of the fact that our multistamps contain real time clock values. Based on the stored time values, the mechanism determines the consistency constraints that are old and replaces them with approximate information. In our system, this approximation may lead to extra messages sent by clients called *consistency stalls*. However, since the removed multistamp constraints are old, it is likely that the consistency information has been received by the relevant clients from the servers. Thus, discarding it has little impact on system performance; our simulation study shows that multistamps impose negligible space and time overheads.

This technique assumes that clocks are loosely synchronized; this assumption is realistic in today's environment where protocols such as the Network Time Protocol [Mil92] are able to achieve low synchronization even in wide-area networks [Mil96]. The correctness of our schemes is not affected if the multistamps are truncated too early or if the clock synchronization is poor.

We also use our multistamp-based mechanisms for providing consistent views to transactions as they execute and for efficiently committing read-only transactions. Read-only transactions are common in transaction processing workloads and improving their performance can significantly improve overall system performance. Our techniques help in reducing latency of read-only transactions since they can commit these transactions without communicating with the servers in most cases; CLOCC requires a message roundtrip for committing all read-only transactions. They also make the system more scalable since there are fewer network messages and most of the work for committing read-only transactions is offloaded to clients. Furthermore, the server need not validate (i.e., check for serializability) read-only transactions; it needs to validate update transactions only with other update transactions. Even though we present our mechanisms in conjunction with CLOCC, they can be used for committing read-only transactions in hybrid systems, i.e., where update transactions are committed using optimism or locking and read-only transactions are committed optimistically using our mechanisms; such a system was presented in [CG85].

1.5 Contributions: Experimental Evaluation

This thesis also evaluates the relative performance of implementations of different isolation levels in a distributed client-server system in which clients cache objects and the transaction code is executed at client machines (i.e., a data-shipping system architecture). To our knowledge, this is the first published study that compares implementations of different isolation levels in such systems. We used a simulator to evaluate different isolation mechanisms using workloads with low to high contention in LAN and WAN environments.

In our study, we assume that clients can cache the accessed objects for the duration of the transaction; this assumption is valid for a large class of applications because we have developed an efficient object-caching mechanism [CALM97]. We also expect our results to hold for some applications in which a transaction's accessed data cannot be cached at a client machine; in many such applications, contention rarely occurs and our simulation results for low-contention workloads can be applied to such cases as well.

We wanted to understand the performance gains offered by committing update transactions at weaker isolation levels since these gains have a high productivity cost: a database programmer must carefully analyze the application code and ensure that it does not corrupt the database when executed at a low isolation level. We were also interested in determining the overheads of high isolation level implementations such as PL-2+ and serializability for read-only and executing transactions since providing stronger consistency guarantees for these transactions further reduces the burden on a database programmer by making it relatively easier to reason about correctness of the application code.

Our results show that providing strong consistency guarantees to update, read-only, and running transactions in client-server systems is not expensive:

- The cost of providing strong consistency guarantees such as serializability to update transactions is negligible for low-contention workloads. The reason is that CLOCC (which we use for providing serializability) has low CPU and communication overheads when there are few conflicts in the workload. This is an important result since many applications exhibit low-contention and such workloads are one of the main environments recommended for using weaker isolation levels; this recommendation is based on the assumption that a higher isolation level mechanism imposes unnecessary performance penalties for such an application.

- For workloads with moderate to high contention, the cost of providing serializability to update transactions is more but it is still not high. At higher contention, the number of aborts is higher in any optimistic scheme including CLOCC. However, in CLOCC, the performance degradation due to serializability is not proportional to the abort rate. The reason is that CLOCC has low costs for restarted transactions and it prevents excessive wasted work by aborting a transaction T early during T's execution. Thus, very high abort rates do not necessarily result in a corresponding performance penalty. For example, even at very high contention (with a high abort rate of more than 100%), the throughput degradation due to serializability was observed to be approximately 10%. In general, we observed that the performance degradation due to serializability is significant only when contention is high *and* the cost of restarting a transaction is high.
- The cost of providing strong consistency guarantees such as PL-2+ using our multistamp-based mechanisms to read-only transactions is low in all workloads; a performance penalty of 2-10% is incurred in such an implementation compared to a system that provides serializability only for update transactions. Unlike CLOCC, these schemes are able to avoid sending a commit message to the servers for most read-only transactions. The results show that the CPU, memory, and network costs imposed by multistamps are very low: the multistamp truncation technique is effective and multistamps smaller than 100 bytes are sufficient for ensuring that few extra messages are sent in the system for maintaining consistency information. The results also show that our multistamp-based mechanisms can be used for providing strong consistency guarantees to executing transactions at very low costs.

1.6 Thesis Outline

This thesis is organized as follows.

Chapter 2 analyses the definitions for different isolation degrees that have been presented in the literature; it shows why the current definitions are inadequate and motivates the need for our work.

Chapter 3 presents our new specifications for the existing ANSI degrees of isolation and proves that they are more flexible than the existing definitions; we provide definitions for both committed and executing transactions. This chapter also discusses how various levels interact with each other.

Chapter 4 presents definitions of our two new isolation levels, PL-2+ and PL-2L, for committed transactions. It describes the implementation-independent specifications of Cursor Stability, and Oracle's Snapshot Isolation and compares it with PL-2+; it also presents a level that captures the essence of Oracle's Read Consistency and compares it with PL-2L.

Chapter 5 presents our mechanisms for providing different isolation guarantees to committed and running transactions. It describes CLOCC and our multistamp-based techniques for efficiently committing transactions and for providing strong guarantees to running transactions.

Chapter 6 discusses our experimental framework. It describes our simulator environment and the workloads that we use for the performance study.

Chapter 7 presents the simulation results for comparing implementations of various isolation levels; it also evaluates the cost of multistamps and the cost of committing read-only transactions.

Chapter 8 concludes the thesis with a summary of our work and suggestions for future research.

Chapter 2

Existing Definitions

This chapter discusses the specifications for different consistency levels that have been presented earlier in the literature [GLPT76, ANS92, BBG$^+$95] and motivates the need for our work.

The chapter is organized in the following manner. Section 2.1 presents the original definitions of consistency [GLPT76] and Section 2.2 describes the ANSI/ISO SQL-92 definitions that were later developed [ANS92]. Section 2.3 summarizes the discussion from [BBG$^+$95]. In Section 2.4, we demonstrate why the solution suggested in [BBG$^+$95] is inadequate.

2.1 Degrees of Isolation

The concept of weak consistency levels was first introduced in [GLPT76] under the name *Degrees of Consistency*, with the aim of providing improved concurrency (and hence better performance) for some workloads by sacrificing the guarantees of full serializability. The degrees were also referred to as *Levels of Consistency* in a companion paper [A$^+$76] and *Degrees of Isolation* in [Dat90, GR93]. The definitions were based on notions of locking: weaker isolation was achieved by reducing the duration (long to short) for which read or write-locks were held. Long-term locks are held until the transaction taking them commits; short-term locks are released immediately after the transaction completes the desired read or write that triggers the lock attempt.

Four degrees of consistency were defined in [GLPT76]. At degree 0, only short write-locks are acquired by a transaction (no read-locks); at degree 1, there are only long write-locks; at degree 2, there are short read-locks and long write-locks; and at degree 3, there are long read and write-locks.

Degree 3 provides serializability since a history acting under 2-phase locking protocol is serializable [BHG87, GR93]. Early release of locks permits histories that are not serializable. Degree 2 requires transactions performing reads to take (at least) short read-locks, which implies that these transactions are unable to read uncommitted updates of transaction acting under degree 1 or higher (due to long write-locks). Transactions at all levels acquire long-term write-locks to prevent concurrent transactions from overwriting each others' changes. Transactions at degree 0 and 1 are not normally expected to perform any writes, but only to get an approximate idea of the database state.

The word “isolation” was chosen to emphasize the fact that different isolation degrees provide varying levels of non-interference of various transactions against each other. Serializability offers the highest degree of isolation since a programmer can write the transaction code with the assumption that the operations of other transactions will not interfere with his/her transaction’s execution.

The work in [GLPT76] also suggests a promising approach that defines isolation levels in terms of graphs. However, since the authors wanted to give an alternate definition of locking behavior using constraints on cycles, their definitions disallow many histories that are permitted by us; such behavior can occur with optimistic or multi-version schemes. Furthermore, those definitions do not consider predicate-based operations and also lack some conditions that are required for correctness.

2.2 ANSI/ISO SQL-92 Definitions

Further refinements to the original isolation levels along with new levels such as Cursor Stability were introduced in [Dat90]. The work in [GLPT76] and [Dat90] set the stage for the ANSI/ISO SQL-92 definitions [ANS92] for isolation levels. The ANSI levels were *informally* defined in terms of English statements that proscribed certain types of behavior or *phenomena* for each isolation level; a completely serializable system disallowed all these situations whereas lower levels of consistency prevented some situations but allow others.

ANSI/ISO SQL-92 [ANS92] defines the phenomena in English as follows:

> **Dirty Read —** Transaction T_1 modifies *x*. Another transaction T_2 then reads *x* before T_1 commits or aborts. If T_1 then aborts, T_2 has read a data item that was never committed and so never really existed.
>
> **Fuzzy or Non-repeatable Read —** Transaction T_1 reads *x* and then T_2 modifies or deletes *x* and commits. If T_1 then attempts to reread *x*, it receives a modified value or discovers that the data item has been deleted.
>
> **Phantom —** Transaction T_1 reads a set of data items satisfying some `<search condition>`. Transaction T_2 then creates data items that satisfy T_1’s `<search condition>` and commits. If T_1 then repeats its read with the same `<search condition>`, it gets a set of data items different from the first read.

The phenomena have not been stated in terms of any particular concurrency control scheme. An important goal of the ANSI/ISO isolation designers was to have flexible definitions that permit a variety of concurrency control mechanisms. Furthermore, the designers also wanted a higher isolation level to be obtained from a lower one by simply disallowing more phenomena or bad situations (i.e., an “additive” kind of property). They defined successively more restrictive isolation levels: READ COMMITTED, REPEATABLE READ, and SERIALIZABLE, which were intended to correspond to degrees 1-3 of [GLPT76] respectively. The ANSI levels are defined as follows: READ COMMITTED disallows Dirty Read, REPEATABLE READ disallows Fuzzy Read as well, and SERIALIZABLE prevents all the above phenomena; the intent of the SERIALIZABLE isolation level is that disallowing all bad situations should provide the normally accepted notion of serializability, i.e., even though transactions execute concurrently, it seems as if they ran in some serial order. There was also a lowest isolation level, READ UNCOMMITTED, that proscribed none of the phenomena. However, READ UNCOMMITTED was not allowed to operate in SQL except in the read-only access mode.

2.3 Preventative Phenomena Approach

The authors of [BBG$^+$95] analyzed the ANSI-SQL standard and demonstrated several problems in its isolation level definitions: some definitions were ambiguous, while others were missing entirely. They showed that the ANSI/ISO definitions have at least two possible interpretations. The first interpretation follows the written description closely and disallows the described situation; we will call this interpretation as the *anomaly interpretation.* The other interpretation prevents any execution sequence that *may* lead to undesirable behavior; we shall call this interpretation the *preventative interpretation* (the work in [ABJ97] presents a different way of stating the preventative interpretation). Thus, the anomaly interpretation allows more histories than the preventative interpretation.

[BBG$^+$95] shows that the anomaly interpretation is incorrect since disallowing all phenomena does not necessarily disallow all non-serializable histories, i.e., a database system that provides isolation guarantees using this interpretation can cause applications to behave incorrectly. Thus, the authors suggest that the preventative interpretation is the correct interpretation of the ANSI definitions. We now summarize this discussion.

We have shown the two different interpretations of the ANSI levels in Figure 2-1. The anomaly interpretation is prefixed by **A** and the preventative interpretation is prefixed by **P**. Reads and writes are denoted by “r” and “w” respectively; “a” and “c” denote abort and commit. An operation “$r_1(x, v)$” indicates that transaction T_1 has read object *x* and the read value is *v*. Similarly, “$w_1(x, v)$” indicates that transaction T_1 has modified object *x*’s value to be *v*. Each entry in the table shows an undesirable situation that must be disallowed.

For dirty reads, proscribing A1 ensures that if T_2 reads from an uncommitted transaction T_1 that aborts, T_2 must not be allowed to commit. The preventative interpretation, P1, is more restrictive and requires that a transaction T_2 must not read any object from an uncommitted transaction T_1 (there is no abort or commit action for T_1 between $w_1(x)$ and $r_2(x)$); the lack of an abort or commit

Phenomenon	Anomaly Interpretation	Preventative Interpretation
Dirty Write	None	**P0:** $w_1(x) \ldots w_2(x)$
Dirty Read	**A1:** $w_1(x) \ldots r_2(x) \ldots$ (a_1 and c_2 in any order)	**P1:** $w_1(x) \ldots r_2(x)$
Fuzzy Read	**A2:** $r_1(x) \ldots w_2(x) \ldots c_2 \ldots r_1(x) \ldots c_1$	**P2:** $r_1(x) \ldots w_2(x)$
Phantom	**A3:** $r_1(P) \ldots w_2(y \text{ in } P) \ldots c_2 \ldots r_1(P) \ldots c_1$	**P3:** $r_1(P) \ldots w_2(y \text{ in } P)$

Figure 2-1: Anomaly and Preventative Interpretations of ANSI levels

action in the condition simply means that all combinations of aborts and commits after $r_2(x)$ are disallowed.

For fuzzy reads, proscribing A2 prevents a transaction T_1 from committing if T_1 reads an object *x* twice and another transaction T_2 overwrites *x* between the two reads by T_1. Phenomenon P2 simply rules out the overwriting of an object being read by an uncommitted transaction.

In databases, queries and updates may be performed on a set of objects if a certain condition called the *predicate* is satisfied. Phenomena A3 and P3 deal with inconsistencies involving predicates and are similar to A2 and P2. The notation "$r_1(P)$" means that transaction T_1 has read objects based on a predicate P and "$w_2(y \text{ in } P)$" says that an object which satisfies P has been modified by T_2. Proscribing P3 requires that T_2 cannot modify a predicate P by inserting, updating, or deleting a row if an uncommitted transaction T_1 has observed objects based on P.

However, phenomenon P3 does not prevent all problems with phantoms. For example, a phenomenon similar to A1 can occur with respect to predicate reads; phenomenon P1 does not prevent such a scenario as well.

An additional property, not included in ANSI/ISO SQL-92, is introduced in [BBG$^+$95]:

> **Dirty Write —** Suppose T_1 modifies *x* and T_2 further modifies *x* before T_1 commits or aborts. If either T_1 or T_2 aborts, it is unclear what the real value of *x* should be.

The preventative interpretation of dirty write, P0, says that a transaction cannot overwrite the changes made by an uncommitted transaction. There is no anomaly interpretation given in [BBG$^+$95]; the authors assume P0 to be a basic requirement that should be included in all levels of consistency.

The paper noted that proscribing these phenomena to define isolation levels is simply a disguised form of imposing a locking protocol on a history. Proscribing P0 is simply a disguised locking definition, requiring T_1 and T_2 to acquire long write-locks. Similarly, proscribing P1 requires T_1 to acquire a long write-lock and T_2 to acquire (at least) a short-term read-lock, and proscribing P2 requires the use of long read and write-locks; disallowing P3 requires acquisition of long phantom read-locks [GR93].

The ANSI levels are redefined in [BBG$^+$95] as follows: READ UNCOMMITTED disallows P0, READ COMMITTED disallows P0 and P1, REPEATABLE READ disallows P0 - P2, and SERIALIZABLE disallows P0 - P3. Figure 2-2 relates isolation levels, preventative phenomena, and the use of locks.

We now present two examples from [BBG$^+$95] that differentiate the preventative and anomaly interpretations. Since P0, P1, P2, and P3 together are equivalent to two-phase locking, all histories permitted by them are serializable. However, the same is not true for *anomaly serializable* histories, which disallow P0, A1, A2, and A3.

The examples concern a transfer from *x* to *y*, where the invariant is that $x + y = 100$. History H_1 differentiates P1 from A1; it shows that A1 is not sufficiently strong to prevent anomalous behavior:

H_1: $r_1(x, 50)$ $w_1(x, 10)$ $r_2(x, 10)$ $r_2(y, 50)$ c_2 $r_1(y, 50)$ $w_1(y, 90)$ c_1

Locking Isolation Level	Proscribed Phenomena	Read-Locks on Data Items and Phantoms (same unless noted)	Write-Locks on Data Items and Phantoms (always the same)
Degree 0	none	none	Short write-locks
Degree 1 = READ UNCOMMITTED	P0	none	Long write-locks
Degree 2 = READ COMMITTED	P0, P1	Short read-locks	Long write-locks
REPEATABLE READ	P0, P1, P2	Long data-item read-locks, Short phantom read-locks	Long write-locks
Degree 3 = SERIALIZABLE	P0, P1, P2, P3	Long read-locks	Long write-locks

Figure 2-2: ANSI isolation levels based on locking (preventative interpretation)

Transaction T_1 does the transfer properly yet transaction T_2 observes the total to be only $60. H_1 is anomaly serializable: phenomenon P0 does not occur because there are no concurrent writes in the system, A1 does not happen since both transactions commit, and A2 does not occur since no transaction reads the same data item twice. However, H_1 is not serializable because T_2 observes an inconsistent state of the database and commits. On the other hand, the preventative interpretation disallows this history since P1 has been violated (T_2 performs a dirty read of object *x*).

History H_2 differentiates A2 and P2 (a similar history can be used for A3 and P3):

H_2: $r_2(x, 50)$ $r_1(x, 50)$ $w_1(x, 10)$ $r_1(y, 50)$ $w_1(y, 90)$ c_1 $r_2(y, 90)$ c_2

With the anomaly interpretation, A2 is not violated since T_2 never reads *x* again after T_1 has modified it (P0 and A1 are also not violated). Thus, history H_2 is anomaly serializable but it is not serializable since T_2 observes the sum of *x* and *y* to be $140 and commits. The preventative interpretation disallows H_2 because condition P2 has been violated (due to $r_2(x)$ and $w_1(x)$).

Thus, the authors conclude that the ANSI/ISO SQL-92 isolation definitions should be interpreted using the preventative interpretation and not the anomaly interpretation.

2.4 Analysis of Preventative Definitions

The ANSI definitions are imprecise because they allow at least two interpretations; furthermore, the anomaly interpretation is definitely incorrect. The preventative interpretation is correct in the sense that it rules out undesirable (i.e., non-serializable) histories. However, this interpretation is overly restrictive since it also rules out correct behavior that does not lead to inconsistencies and can occur in a real system. Thus, any system that allows such histories is disallowed by this interpretation, e.g., databases based on optimistic mechanisms [AGLM95, BOS91, BBG$^+$95, KR81, ABGS87, FCL97].

We first show that the preventative interpretation is overly restrictive since it rules out serializable histories. Next we briefly discuss how optimistic schemes deal with good and bad histories; finally, we show why the preventative interpretation disallows optimistic and multi-version mechanisms.

2.4.1 Restrictiveness

This section illustrates the restrictiveness of the preventative interpretation by giving examples of serializable behavior that it rules out. First, consider the non-serializable history H_1, which was presented in the previous section:

H_1: $r_1(x, 50)$ $w_1(x, 10)$ $r_2(x, 10)$ $r_2(y, 50)$ c_2 $r_1(y, 50)$ $w_1(y, 90)$ c_1

H_1 is clearly not serializable and ought not to be permitted by any concurrency control scheme. However, suppose that transaction T_2 reads the new values of *x* and *y* as installed by T_1:

$H_{1'}$: $r_1(x, 50)$ $w_1(x, 10)$ $r_1(y, 50)$ $w_1(y, 90)$ $r_2(x, 10)$ $r_2(y, 90)$ c_1 c_2

In this case, T_2's reads happen after T_1's writes have occurred but before T_1 commits. History $H_{1'}$ is serializable but is not permitted by the preventative interpretation because it violates P1. Now consider history H_2 which is not serializable since T_2 reads *x*'s old value and *y*'s new value:

H_2: $r_2(x, 50)$ $r_1(x, 50)$ $w_1(x, 10)$ $r_1(y, 50)$ $w_1(y, 90)$ c_1 $r_2(y, 90)$ c_2

Instead of reading the new value of *y*, suppose that T_2 reads the old values of *x* and *y*. The resulting history, $H_{2'}$, is serializable:

$H_{2'}$: $r_2(x, 50)$ $r_1(x, 50)$ $w_1(x, 10)$ $r_1(y, 50)$ $r_2(y, 50)$ $w_1(y, 90)$ c_2 c_1

However, $H_{2'}$ is disallowed by the preventative interpretation because it violates P2 (T_1 overwrites objects *x* and *y* that have been read by an uncommitted transaction T_2).

It is not surprising that the preventative interpretation rules out histories like $H_{1'}$ and $H_{2'}$. This interpretation *prevents* conflicting operations from executing concurrently; it disallows all histories that would not occur in a lock-based implementation. Thus, even though the operations in $H_{1'}$ and $H_{2'}$ have been scheduled such that these histories are serializable, the preventative interpretation disallows them because they allow conflicting reads and writes to run simultaneously.

The real problem with the preventative approach is that the phenomena are expressed in terms of single-object histories. However, the properties of interest are often multi-object constraints. To avoid problems with such constraints, the phenomena need to restrict what can be done with individual objects more than is necessary. Our approach avoids this difficulty by using specifications that capture constraints on multiple objects directly.

2.4.2 Optimism

The fact that some legal histories are ruled out by the preventative interpretation would not be important if those histories did not arise in real implementations. But in fact both histories $H_{1'}$ and $H_{2'}$ are allowed by optimistic and multi-version mechanisms. Since an important goal of the ANSI/ISO SQL-92 isolation levels is to permit non-locking implementations while providing useful guarantees to a database programmer, providing a definition of these levels that precludes all but lock-based implementations is undesirable.

Before we describe how optimistic schemes deal with histories such as H_2 and $H_{2'}$, we briefly discuss the characteristics of these schemes. Unlike a locking scheme, optimistic implementations allow conflicting operations by concurrent transactions and abort some transactions if necessary. In these schemes, modifications are not made to the database directly; instead they are made to volatile copies. These copies are installed in the database at commit time if the transaction *validates* successfully. During the validation process, the database system checks if a committing transaction T can be serialized by comparing T's reads/writes with the reads/writes of other transactions. If validation fails, T is aborted.

Validation is done in two different ways — forward and backward validation [Hae84]. *Forward validation* compares the transaction with all uncommitted transactions; if a committing transaction T has modified any object that has been read by an uncommitted transaction S, T is aborted. Thus, when transaction S commits, only its writes will have to be validated; its reads are certain to be valid. (Forward validation schemes are similar to "optimistic locking" implementations [FCL97].) *Backward validation* checks the committing transaction T against all previously committed transactions; if T has read any object *x* that has been modified by a committed transaction since T read *x*, T is aborted. Since backward validation does not consider uncommitted transactions, the commit of a transaction may cause an uncommitted transaction to abort later; furthermore, unlike forward validation, a preparing transaction's reads must be validated as well.

Disallowing Bad Histories

Both forward and backward schemes will reject H_1 and H_2. History H_1 is not permitted by most optimistic implementations since these schemes operate on local copies and disallow dirty reads. Consider a prefix of history H_2:

H_2 (prefix): $r_2(x, 50)$ $r_1(x, 50)$ $w_1(x, 10)$ $r_1(y, 50)$ $w_1(y, 90)$ A

If T_1 tries to commit at point *A*, a forward validation scheme will abort T_1 since T_1's modifications conflict with the reads of an uncommitted transaction T_2. A backward validation scheme will allow T_1 to commit. However, when T_2 tries to commit, it will be aborted because a committed transaction T_1 has overwritten an object *x* that was read by T_2.

Accepting Good Histories

We now discuss why phenomena P0, P1, and P2 rule out optimistic schemes when each phenomenon is considered individually.

Phenomenon P0 can occur in optimistic implementations since there can be many uncommitted transactions modifying local copies of the same object concurrently; if necessary, some of them will be forced to abort so that serializability can be provided. Thus, disallowing P0 can rule out optimistic implementations, e.g., the following serializable history that is allowed by many optimistic schemes is ruled out by P0:

$H_{0'}$: $w_1(x, 10)$ $w_1(y, 90)$ $w_2(x, 50)$ $w_2(y, 50)$ c_1 a_2

Proscribing P1 disallows transactions from reading updates by uncommitted transactions. Such reads are disallowed by many optimistic schemes, but they are desirable in mobile environments, where commits may take a long time if clients are disconnected from the servers [GHOS96, GKLS94]. For example, history $H_{1'}$ can occur in a mobile system, but P1 disallows it. In such a system, commits can be assumed to have happened "tentatively" at client machines; later transactions may observe modifications of those tentative transactions. When the client reconnects with the servers, its work is checked to determine if consistency has been violated and the relevant transactions are aborted. Of course, if dirty reads are allowed, cascading aborts can occur, e.g., in history $H_{1'}$, T_2 must abort if T_1 aborts; this problem can be alleviated by using compensating actions [KS91, TTP+95, KSS97]. Another environment where reads from uncommitted transactions may be desirable are high traffic hotspots [O'N86]; disallowing P1 rules out mechanisms designed for these situations.

Proscribing P2 disallows a transaction to modify an object that has been read by another uncommitted transaction (P3 rules out a similar situation with respect to predicates). As with P0, uncommitted transactions may read/write the same object concurrently in an optimistic implementation. There is no harm in allowing phenomenon P2 if transactions commit in the right order. For example, history $H_{2'}$ is accepted by both forward and backward validation schemes. Consider a prefix of history $H_{2'}$:

$H_{2'}$ (prefix): $r_2(x, 50)$ $r_1(x, 50)$ $w_1(x, 10)$ $r_1(y, 50)$ $r_2(y, 50)$ $w_1(y, 90)$ *A*

If T_2 and T_1 try to commit at point *A* (in the order T_2 followed by T_1), both validation schemes will allow the commits; T_1 will be serialized after T_2. When T_2 tries to commit, it succeeds validation with a forward validation scheme since it has not modified any object. When T_1 tries to commit later, it succeeds since its modifications do not conflict with any uncommitted transaction's reads/writes. With a backward scheme, T_2 commits successfully since it does not conflict with an already committed transaction; similarly, T_1 is also allowed to commit later.

2.4.3 Multi-version schemes

Multi-version schemes [Ree78, BHG87, BBG+95, Wei87] allow multiple versions of the same object to exist in the database state. Thus, in a history such as H_2, when transaction T_2 tries to read object *y*, it can be provided with an old version of *y* (and not the latest version) resulting in a serializable history. However, the conditions in the ANSI definitions (and the preventative interpretation) are specified in terms of single-version object histories rather than multi-version histories. Thus, they are inadequate for analyzing multi-version schemes. In fact, the authors in [BBG$^+$95] point out that in order to place multi-version schemes such as Snapshot Isolation in the isolation hierarchy, the designer of the scheme must first map the histories of such schemes to single-version histories and then apply the consistency conditions. We address his problem directly by specifying our conditions in terms of multi-version histories.

2.5 Summary

This chapter has analyzed the existing definitions for various degrees of isolation. The original definitions presented in [GLPT76] were inspired by a lock-based interpretation. That work also presented informal English statements that formed the basis of the ANSI/ISO SQL-92 definitions, an industry standard [ANS92]. However, as shown in [BBG$^+$95], these definitions are ambiguous and at least one interpretation can permit histories that lead to inconsistencies. The authors in [BBG$^+$95] suggest another interpretation called the preventative interpretation that is essentially a disguised form of locking. Our analysis showed that this interpretation is overly restrictive since it disallows histories that can occur in realistic implementations, especially systems that use optimistic or multi-version mechanisms. The preventative interpretation only permits implementations that prevent conflicting operations from running concurrently. On the other hand, optimistic schemes allow such operations to execute simultaneously and then abort the relevant transactions.

Chapter 3

Proposed Specifications for Existing Isolation Levels

This chapter presents new specifications for the existing ANSI isolation levels. Our definitions allow optimistic and multi-version implementations; we also specify a variety of guarantees for predicate-based operations at weak consistency levels in an implementation-independent manner. Our specifications address the issues of multi-object constraints directly by using complete read and write sets of transactions. We specify different isolation levels using a combination of constraints on object histories and graphs; we proscribe different types of cycles in a serialization graph at each isolation level. Apart from using graphs to define isolation levels for committed and transactions, we also use them to specify interactions among various isolation levels.

Our graphs are similar to those that have been used before for specifying serializability [BHG87, KSS97, GR93], semantics-based correctness criteria [AAS93], and for defining extended transaction models [CR94]. Our approach is the first that applies these techniques to defining ANSI and commercial isolation levels.

The rest of this chapter is organized as follows. Section 3.1 presents the system model and terminology that we use for our specifications. Section 3.2 presents our specifications of the existing isolation levels for committed transactions. In Section 3.3, we discuss how various levels interact with each other. In Section 3.4, we prove that our specifications allow more histories and are strictly less restrictive than the preventative interpretation. In Section 3.5, we discuss how our specifications can be extended to provide consistency guarantees to transactions as they execute.

3.1 System Model and Terminology

We now present the system model and terminology that will be used for specifying our isolation conditions. We use a multi-version model similar to what has been presented in the literature [BHG87]. However, unlike earlier work, our model incorporates predicates and handles them in a correct and flexible manner at all isolation levels.

3.1.1 Database Model

The database consists of objects that can be read or written by transactions. Each transaction reads and writes objects and indicates a total order in which these operations occur; thus, our transactions are sequential in nature. An object has one or more versions. Transactions interact with the database only in terms of objects; the system maps each operation on an object to a specific version of that object. A transaction may read versions created by committed, uncommitted, or even aborted transactions; constraints imposed by some isolation levels will prevent certain types of reads, e.g., reading versions created by aborted transactions.

When a transaction writes an object *x*, it creates a new version of *x*. A transaction T_i can modify an object multiple times; its first update of object *x* is denoted by $\mathrm{x}_{i.1}$, the second by $\mathrm{x}_{i.2}$, and so on. Version x_i denotes the final modification of *x* performed by T_i before it commits or aborts. That is,

$$\mathrm{x}_i \equiv \mathrm{x}_{i.n} \text{ where n = max } \{\mathrm{j} \mid \mathrm{x}_{i.j} \text{ exists}\}$$

The last operation of a transaction is a *commit* or *abort* operation to indicate whether the transaction's execution was successful or not; there is at most one commit or abort operation for a transaction.

An event is added to a transaction's sequence of events after it has been registered by the database, e.g., for a read event of version x_i, it could mean that the database has added x_i to the transaction's read set and returned the relevant value. Similarly, in an optimistic system, a commit may be requested for transaction T_i but T_i may abort; in this case, we register an abort event for T_i and not a commit event.

The database state refers to the versions of objects that have been created by committed *and* uncommitted transactions. The *committed state of the database* reflects only the modifications of committed transactions. When transaction T_i commits, each version x_i created by T_i becomes a part of the committed state and we say that T_i *installs* x_i. If T_i aborts, x_i does not become part of the committed state. Thus, the system needs to prevent modifications made by uncommitted and aborted transactions from affecting the committed database state.

Conceptually, the committed state comes into existence as a result of running a special initialization transaction, T_{init}. Transaction T_{init} creates all objects that will ever exist in the database; at this point, each object *x* has an initial version, x_{init}, called the *unborn* version. When an application transaction inserts an object *x* (e.g., inserts a tuple in a relation), we model it as the creation of a *visible* version for *x*. When a transaction T_i deletes an object *x* (e.g., by deleting a tuple from some relation), we model it as the creation of a special *dead* version, i.e., in this case, x_i (also called x_{dead}) is a dead version. Thus, object versions can be of three kinds — unborn, visible, and dead; the ordering relationship between these versions is discussed in Section 3.1.2.

All objects in the database have a unique identity that is not based on field values. Suppose transaction T_i deletes *x* (i.e., x_i is a dead version) and a later transaction T_j checks if this tuple exists and inserts a new tuple. Transaction T_j's insert operation overwrites the unborn version of an object *y* that has not been used before and creates a visible version of *y*, i.e., the deleted and inserted objects are different.

When a transaction T_i performs an insert operation, the system selects a *unique* object *x* that has never been selected for insertion before and T_i creates a visible version of *x* if T_i commits. If two transactions try to insert a tuple with the same field values, the system selects two distinct objects for insertion. The decision whether both tuples can be inserted is left to the application and the database system, i.e., our model does not require that the database contains unique tuples.

We assume object versions exist forever in the committed state to simplify handling of inserts and deletes. An implementation only needs to maintain visible versions of objects, and a single-version implementation can maintain just one visible version at a time. Furthermore, application transactions in a real system access only visible versions. Well-formed systems enforce this constraint in different ways. For example, in an object-oriented database system, an application might be unable to access deleted objects since the system provides garbage collection. In a relational database with explicit deletion, a transaction might look up an object using its primary key after that object has been deleted. The system can inform the transaction that no such object exists, or, if keys are reused, it could return a different object. As discussed above, in our model, these objects will be considered to be distinct.

3.1.2 Transaction Histories

We capture what happens in an execution of a database system by a history. A *history H* over a set of transactions consists of two parts — a partial order of events *E* that reflects the operations (e.g., read, write, abort, commit) of those transactions, and a version order, $\ll$, that is a total order on committed object versions.

Each event in a history corresponds to an event of some transaction, i.e., read, write, commit or abort. A write operation on object *x* by transaction T_i is denoted by $w_i(x_i)$ ($w_i(x_{i.m})$ for the m^{th} modification to *x*); if the value *v* is written into x_i, we use the notation, $w_i(x_i, v)$. When a transaction T_j reads a data item *x*, it reads some version of *x* that was written by a transaction T_i (T_i could be the same as T_j); we denote this as $r_j(x_i)$ (or $r_j(x_{i.m})$ if T_j reads an intermediate version). To indicate that T_j has read x_i's value to be *v*, we use the notation $r_j(x_i, v)$. Note that version x_i is not necessarily the most recently installed version in the committed state. (The subscript of "w" in the write operation is always the same as the version of the modified object; this redundancy exists so that the notation for writes is similar to that for reads.)

The partial order of events *E* in a history obeys the following constraints:

- It preserves the order of all events within a transaction including the commit and abort events.
- If an event $r_j(x_{i.m})$ exists in *E*, it is preceded by $w_i(x_{i.m})$ in *E*, i.e., a transaction T_j cannot read version x_i of object *x* before it has been produced by T_i.
- If an event $w_i(x_{i.m})$ is followed by $r_i(x_j)$ without an intervening event $w_i(x_{i.n})$ in *E*, x_j must

be $x_{i.m}$. This condition ensures that if a transaction modifies object *x* and later reads *x*, it will observe its last update to *x*.

The partial order in [BHG87] places more constraints than the ones given above since their theory has been developed for defining serializability. In our case, we add those extra conditions as they are needed at each lower isolation level; our specification of serializability captures all conditions presented in [BHG87].

For convenience, we will present history events in our examples as a total order (from left to right) that is consistent with the partial order. Furthermore, wherever possible in our examples, we make this total order be consistent with the real-time ordering of events in a database system; a similar approach was adopted in [GR93].

The second part of a history H is the version order, $\ll$, that specifies a total order on object versions created by *committed* transactions in H; there is no constraint on versions due to uncommitted or aborted transactions. We refer to versions due to committed transactions in H as *committed versions* and impose two constraints on H's version order for different kinds of committed versions:

- the version order of each object *x* contains exactly one initial version, x_{init}, and at most one dead version, x_{dead}.
- x_{init} is *x*'s first version in its version order and x_{dead} is its last version (if it exists); all visible versions are placed between x_{init} and x_{dead}.

We additionally constrain the system to allow reads only of visible versions:

- if $r_j(x_i)$ occurs in a history, then x_i is a visible version.

For convenience, we will typically only show the version order for visible versions in our example histories; in cases where unborn or dead versions help in illustrating an issue, we will show some of these versions as well.

The version order in a history H can be different from the order of write or commit events in H. This flexibility is needed to allow certain optimistic and multi-version implementations where it is possible that a version x_i is placed before version x_j in the version order ($x_i \ll x_j$) even though x_i is installed in the committed state *after* x_j is installed. For example, consider history $H_{write-order}$:

$H_{write-order}$: $w_1(x_1)$ $w_2(x_2)$ $w_2(y_2)$ c_1 c_2 $r_3(x_1)$ $w_3(x_3)$ $w_4(y_4)$ a_4 $[x_2 \ll x_1]$

In this history, the database system chooses the version order $x_2 \ll x_1$ even though T_1 commits before T_2. The fact that the write of x_1 occurs before x_2 in the history does not determine the version order either; the system chooses the version order for each object. Furthermore, there are no constraints on x_3 (yet) or y_4 since these versions correspond to uncommitted and aborted transactions, respectively.

In our examples, the subscripts used for labeling transactions are used purely as identifiers and are not supposed to imply any ordering between transactions; all orderings are specified by the history and the version order. Thus, in a history, events of T_1 may or may not precede events of T_2.

3.1.3 Predicates

We now discuss how we handle predicates in our model. In databases, queries and updates may be performed on a set of objects if a certain condition called the *predicate* [GR93] is satisfied. For example, a transaction can execute an SQL statement that updates the phone numbers of all employees whose place of residence is Cambridge; in this case, the predicate is the condition "place of stay = Cambridge" over the relevant relations.

In our model, we assume that predicates are used with relations in a relational database system. There are two modification operations possible on the structure of relations: insertion or deletion of tuples (of course, tuples can be read or updated as discussed earlier; our notation for reading via predicates is discussed below).

We extend our database model in the following way. We divide the database into relations and each object (with all its versions) exists in some relation. As before, unborn and dead versions exist for an object before the object's insertion and after its deletion. A point to note here is that an object's relation is known in our model when the database is initialized by T_{init}, i.e., *before* the object is inserted by an application transaction. Of course, this assumption is needed only at a conceptual level. In an implementation, the system knows the relation of an object *x* when *x* is inserted in a relation.

A predicate P identifies a boolean condition and the relations on which the condition has to be applied; one or more relations can be specified in P. All objects that match this condition are read or modified depending on whether a predicate-based read or write is being considered. We do not make any assumptions about the language used for specifying predicates.

Definition 1 : Version set of a predicate-based operation. When a transaction executes a read or write based on a predicate P, the system selects a version for *each* object in P's relations. The set of selected versions is called the *Version set* of this predicate-based operation and is denoted by Vset(P).

The version set defines the state that is observed to evaluate a predicate P. Since we select a version for all possible objects in P's relations, this set will be very large since it includes unborn and possibly dead versions of some objects. For convenience, we will only show visible versions in a version set; to better explain some examples, we will sometimes also show some unborn and dead versions.

We use the following scenario for explaining various issues regarding predicates. We consider a database in which an Employee relation contains visible versions of objects, *x* and *y*. In many of our examples, *x* is a tuple for an employee in the Sales department and *y* corresponds to an employee in the Legal department; in some cases, we will also consider object *z* for which only an unborn version exists (there will be many other such objects in the relation but we will not need them in our examples).

Predicate-based Reads

If a transaction T_i performs reads based on a predicate P (e.g., in a SQL statement), we represent the event in a history as r_i(P: Vset(P)). To execute such a read, the system (conceptually) reads all versions in Vset(P). Then, the system determines which objects *match* predicate P by evaluating P's boolean condition on the versions in Vset(P); objects whose unborn and dead versions were selected in the previous step do not match.

Here is an example that illustrates reads based on predicates using the above scenario. Suppose that transaction T_i executes a query to determine the tuples in the Employee relation for which the predicate "Dept = Sales" is true. This query (conceptually) reads a version of all objects in the Employee relation, e.g., x_1 and y_2, and the unborn/dead versions (such as z_{init}) of other objects in this relation. This predicate-based read could be shown in a history as r_i(Dept=Sales: x_1; y_2); in this case we do not show any unborn or dead versions in the version set. Version x_1 matches the predicate and y_2 does not match; recall that z_{init} cannot match the predicate. After this query, T_i can execute operations on the matched object, e.g., it could read x_1's value. These reads will show up as *separate* events in the history. If T_i does not read x_1, we do not add a read event to the history, e.g., T_i could simply use the fact that one object matched the predicate. Thus, the history only shows reads of versions that were actually observed by transaction T_i.

Predicate-based Modifications

For writes based on a predicate P, we use the following approach. As for predicate-based reads, the system selects versions for all objects specified in P's relations and then determines which object versions match P; as in the case of reads, unborn and dead versions cannot match P. Then, all objects that match the predicate are modified by the transaction, i.e., in the case of the delete operation, dead versions are installed for the deleted objects. We model this behavior as a sequence of events: w_i(P: Vset(P)) $w_i(x_i)$ $w_i(y_i)$ . . ., where *x, y, . . .* are the objects that matched predicate P.

When a transaction modifies objects based on a predicate, it indicates the modification to be performed. This information is used by the database system to generate appropriate write events. Since a history captures what happened in a system, we show the write-events *after* the system has interpreted the predicate-based operation.

Here are some examples to illustrate writes based on predicates. Suppose that transaction T_i executes the following statement for the employee database discussed above: "increment by $10 salaries of all employees where Dept=Sales". Suppose that the system selects versions, x_1, y_2, and z_{init}, for this operation. If x_1 matches the predicate but y_2 and z_{init} do not, the following events are added to the history: w_i(Dept=Sales: x_1; y_2; z_{init}) $w_i(x_i)$.

Similarly, if a transaction T_i deletes all employees from the Sales department in the above scenario, the following events are added to the history: w_i(Dept=Sales: x_1; y_2; z_{init}) $w_i(x_i)$. Note that similar events are added to the history in the case of deletion as well. However, there is a difference: in the deletion example, x_i is a dead version and cannot be modified further whereas in the update case, x_i can be overwritten later.

Note that a predicate-based modification can also be modeled as a predicate-based read followed by a set of writes on the matched objects. Thus, in the example given above, the operations added to the history are: r_i(Dept=Sales: x_1; y_2; z_{init}) $w_i(x_i)$. This technique provides weaker guarantees (than the approach given above) to predicate-based modifications at lower isolation levels and is supported by some commercial databases. It is possible to change our definitions to allow such semantics; we have chosen a different approach since we wanted to provide stronger guarantees for predicate-based modifications at lower isolation levels.

We only treat updates and deletes as writes based on predicates; we do not provide special "insert predicates". Insert operations are modeled as discussed in Section 3.1.1, i.e., if T_i inserts a tuple, the system selects a unique object *x* (which has never been selected for insertion before) and creates a visible version of x_i when T_i commits.

An insert statement in a database language such as SQL specifies the tuples to be added to a relation; these tuples could be specified explicitly or could be evaluated from a query. If a query is used for generating the values, our model treats such an insert operation as a predicate-based read followed by a sequence of write events that correspond to the tuples being added. This approach corresponds to the semantics of some commercial databases such as IBM's DB2 [IBM99]. For example, consider the following statement that copies employees whose commission exceeds 25% of their salary into the BONUS table (this statement is executed by transaction T_1):

T_1: INSERT INTO Bonus SELECT name, sal, comm FROM Emp
WHERE comm $>$ 0.25 * sal

As stated in the DB2 manual [IBM99], the lock-protocol for the Emp relation is governed by the "rules for read-only processing"; for the Bonus relation, the lock-protocol for "change processing" is used. Thus, if the transaction executes at degree 2, read-locks on the Emp relation are released after the insert statement. This allows later transactions to modify the Emp relation while T_1 is still uncommitted.

As stated above, our approach handles T_1's insert in a similar manner. Here is a possible history for T_1's execution in our model:

$H_{query-insert}$: r_1(comm > 0.25 * sal: x_0) $r_1(x_0)$ $w_1(y_1)$ c_1

In this history, since x_0 matches the predicate-based query, it is read by T_1 to generate a tuple that is inserted into the Bonus table. We do not consider the insertion of y_1 as a predicate-based write since the predicate is used to determine what tuples will be inserted in the Bonus relation and not for determining which matched objects need to modified. Modeling an insert operation as a predicate-based read followed by a set of write events allows flexibility and ensures that some commercial systems are not ruled out.

Discussion

Our approach of observing some version of each object in one of the selected relations allows us to handle the phantom problem [GR93] in a simple way (as we will see later). However, this does not constrain implementations to perform these observations, e.g., an implementation could use an index.

To model reads and writes based on predicates, we introduced the notion of relations in a database. A read operation based on predicate P only observes object versions in relations that are specified by P (similarly for writes). We chose this approach rather than having a predicate-based read observe versions of all objects in the database because an object *x* in a relation that is not specified by P does not matter. The only way that *x* can match P is if its relation is changed to be one of P's relations. Since objects do not change relations in our model, we can ignore object *x* for an operation based on predicate P since *x*'s relation is not specified by P. Alternatively, we could say that the system chooses unborn versions of all objects in such relations. This is equivalent to our approach of dividing the database into relations. Our approach has an additional advantage that it models how predicates are used in application code, i.e., a predicate indicates the relations on which it operates.

3.1.4 Conflicts and Serialization Graphs

We first define the different types of read/write conflicts that can occur in a database system and then use them to specify serialization graphs. Our notion of conflicts and graphs is similar to the ones given in [BHG87] but they are not exactly same; we capture conflict-serializability with our definitions whereas the conditions given in [BHG87] define view-serializability. We define three kinds of *direct conflicts* that capture conflicts of two different *committed* transactions on the same object or intersecting predicates — read-dependency, anti-dependency, and write-dependency. The first type, *read dependency*, specifies write-read conflicts; a transaction T_j depends on T_i if it reads T_i's updates. *Anti-dependencies* capture read-write conflicts; T_j anti-depends on T_i if it overwrites an object that T_i has read. Write-dependencies capture write-write conflicts; T_j write-depends on T_i if it overwrites an object that T_i has also modified. We now discuss these conflicts in detail. For convenience, we have separated the definitions of predicate-based conflicts and regular conflicts.

Definition 2 : Directly Read-Depends. We say that T_j directly read-depends on transaction T_i if it directly item-read-depends or directly predicate-read-depends on T_i (denoted by $T_i \xrightarrow{wr} T_j$).

Directly item-read-depends: We say that T_j *directly item-read-depends* on T_i if T_i installs some object version x_i and T_j reads x_i.

Directly predicate-read-depends Transaction T_j *directly predicate-read-depends* on T_i if T_j performs an operation r_j(P: Vset(P)) and $x_i \in$ Vset(P).

Transaction T$_j$ directly predicate-read-depends on the initialization transaction T$_{init}$ since T$_j$ observes the unborn versions of objects that have yet not been inserted in P's relations; if T$_j$ observes a dead version of some object, it directly read-depends on the transaction that deleted that object.

For predicate-read-dependencies, all objects in the version set of a predicate-based read are considered to be read, including objects that do not match the predicate. The versions that are actually accessed by transaction T$_i$ show up as normal read events. Other versions in the version set are essentially *ghost reads*, i.e., their values are not observed by the predicate-based read but read-dependencies are established for them as well.

We now define the notion of overwriting a predicate-based operation that is useful for defining anti-dependencies and write-dependencies.

Definition 3 : Overwriting a predicate-based operation. We say that a transaction T$_j$ overwrites an operation r$_i$(P: Vset(P)) (or w$_i$(P: Vset(P))) based on predicate P if T$_j$ installs x$_j$ such that x$_k$ $\ll$ x$_j$, x$_k$ $\in$ Vset(P) and x$_k$ matches P whereas x$_j$ does not match P or vice-versa. That is, T$_j$ makes a modification that changes the set of objects matched by T$_i$'s predicate-based operation. The notion of a write operation overwriting a predicate-based operation can be defined similarly.

Definition 4 : Directly Anti-Depends. A transaction T$_j$ directly anti-depends on transaction T$_i$ if it directly item-anti-depends or directly predicate-anti-depends on T$_i$ (denoted by $T_i \xrightarrow{rw} T_j$).

Directly item-anti-depends: We say that T$_j$ *directly item-anti-depends* on transaction T$_i$ if T$_i$ reads some object version x$_k$ and T$_j$ installs *x*'s next version (after x$_k$) in the version order. Note that the transaction that wrote the later version directly item-anti-depends on the transaction that read the earlier version.

Directly predicate-anti-depends: We say that T$_j$ *directly predicate-anti-depends* on T$_i$ if T$_j$ overwrites an operation r$_i$(P: Vset(P)). That is, if T$_j$ installs a later version of some object that changes the matches of a predicate-based read performed by T$_i$.

This definition handles inserts and deletes. For example, suppose that transaction T$_j$ inserts a tuple *x* in an Employee relation that corresponds to a person who works in the Sales department. Another transaction T$_i$ searches for all employees in the Sales department but does not find this record, i.e., T$_i$ reads *x*'s unborn version. Transaction T$_j$ directly predicate-anti-depends on T$_i$ since T$_j$ changes the objects matched by T$_i$'s read.

Note that read-dependencies are treated differently from anti-dependencies for predicates. A transaction T$_j$ directly predicate-read-depends on all transactions that produced the versions in Vset(P) However, if transaction T$_i$ performs an operation r$_i$(P: Vset(P)), T$_j$ predicate-anti-depends on T_i only if T_j's modifications change the set of objects that *matched* P; simply overwriting a version in Vset(P) does not cause a predicate-anti-dependency. We avoid extra anti-dependency edges originating from T_i since they can unnecessarily disallow legal histories.

In a two-phase locking implementation (for providing serializability), if a transaction T_1 performs a read based on predicate P and T_2 tries to insert an object *x* covered by P, T_2 is delayed till T_1 finishes. In our model, T_1 reads x_{unborn} and T_2 creates a later version x_1. If T_2 changes the matches by T_1's read, T_2 predicate-anti-depends on T_1. Note that T_1's predicate read-locks delay T_2 even if T_2 does not change the objects matched by P. Our definitions are more flexible and permit implementations that allow T_2 to proceed in such cases, e.g., precision locks and granular locks [GR93].

Here is an example to illustrate anti-dependencies with respect to predicates. Consider the employee database scenario described in Section 3.1.3 that contains visible versions of two objects *x* and *y*. Suppose T_i executes a query that selects all Employees in the Sales department and examines versions x_1 and y_2 along with unborn/dead versions of other objects; it determines that *x* is in Sales and *y* is not. If T_i commits, it will read-depend on T_1 and T_2, T_{init}, and transactions that created dead versions observed by T_i. A later transaction T_j will directly predicate-anti-depend on T_i if T_j adds a new employee to the Sales department, moves *y* to Sales, removes *x* from Sales, or deletes *x* from the database.

Definition 5 : Directly Write-Depends. A transaction T_j directly write-depends on T_i if it directly item-write-depends or directly predicate-write-depends on T_i This situation is denoted by $T_i \xrightarrow{ww} T_j$.

Directly item-write-depends: We say that T_j *directly item-write-depends* on transaction T_i if T_i installs a version x_i and T_j installs *x*'s next version (after x_i) in the version order.

Directly predicate-write-depends: We say that T_j *directly predicate-write-depends* on T_i if either

1. T_j overwrites an operation w_i(P: Vset(P)), or
2. T_j executes an operation w_j(Q: Vset(Q)) and $x_i \in$ Vset(Q).

In other words, T_j predicate-write-depends on T_i if T_j installs a later version of some object that changes the matches of a predicate-based write performed by T_i or if the system selects a version x_i in T_j's predicate-based write.

The definition of predicate-write-dependencies is similar to the definitions of predicate-read-dependencies and predicate-anti-dependencies given above. Part (1) of this definition matches read-dependencies, i.e., all objects in the version set of a predicate-based write are effectively considered to be modified, including objects that do not match the predicate. We refer to such writes of unmatched objects as *ghost writes* since no new versions are created for them. For objects that match the predicate, separate write events are generated.

Part (2) of predicate-write-depends is similar to predicate-anti-depends: in both cases, a transaction overwrites a predicate-based read or write of another transaction. As in the case of anti-dependencies, we do not consider a transaction T$_j$ to predicate-write-depend on T$_i$ if T$_j$ simply overwrites a version in Vset(P) (of an operation w$_i$(P: Vset(P))); instead T$_j$ must change the objects matched by P. Predicate-write-dependencies are chosen in this manner to prevent legal histories from being unnecessarily disallowed.

Note that if predicate-based modifications are modeled as predicate-based reads followed by normal writes, the notion of predicate-write-depends is not needed.

The definition for predicate-write-depends handles inserts and deletes also. For example, suppose T$_i$ updates the salaries of all employees in an Employee relation for which "Dept=Sales" is true. Suppose that a later transaction T$_j$ inserts a tuple (z$_j$) for a new employee in the Sales department. Transaction T$_j$ directly predicate-write-depends on T$_i$ since T$_j$ installs an object version that changes the matches of T$_i$'s predicate-based write (when T$_i$ performs its update, w$_i$(P: Vset(P)), its version set Vset(P) contains z$_{init}$ which does not match P).

Let us consider an example that involves deletes. Suppose that transaction T$_i$ deletes a record *x* that matches a predicate "social security number = SN" and a later transaction T$_j$ inserts a record *y* with the same social security number in the database. In our model, these two objects are distinct: T$_i$ creates the dead version of *x* and T$_j$ creates the first visible version of *y*. T$_j$ predicate-write-depends on T$_i$ since it changes the objects matched by T$_i$'s predicate and installs a later version of *y* (T$_i$'s object set contains version y$_{init}$ that does not match whereas y$_j$ matches the predicate).

The following subsidiary definitions will also be useful:

Definition 6 : Directly depends. We say T$_j$ directly depends on T$_i$ if T$_j$ directly write-depends or directly read-depends on T$_i$.

Definition 7 : Directly conflict-depends. We say that T$_j$ directly conflict-depends on T$_i$ if it directly depends or directly anti-depends on T$_i$.

Now we can define the Direct Serialization Graph or DSG. This graph is called "direct" since it is based on the direct conflicts discussed above.

Definition 8 : Direct Serialization Graph. We define the direct serialization graph arising from a history H, denoted DSG(H), as follows. Each node in DSG(H) corresponds to a *committed* transaction in H and directed edges correspond to different types of direct conflicts. There is a *read/write/anti-dependency* edge from transaction T$_i$ to transaction T$_j$ if T$_j$ *directly read/write/anti-depends* on T$_i$.

There can be at most one edge of a particular kind from node T$_i$ to T$_j$ since the edges do not record the objects that gave rise to the conflict; our conditions have been defined such that we only need to know whether a particular type of edge exists between two nodes.

A DSG does not capture all information in a history and hence it does not replace the history, e.g., a DSG only records information about committed transactions. The history is still available if needed, and in fact, we use the history instead of the DSG for some conditions.

Figure 3-1 reviews the definitions for direct conflicts between transactions and shows the notation used in DSGs. As an example, consider the following history:

$$H_{serializable}\text{:}\;\; w_1(z_1)\;\; w_1(x_1)\;\; w_1(y_1)\;\; w_3(x_3)\;\; c_1\;\; r_2(x_1)\;\; w_2(y_2)\;\; c_2\;\; r_3(y_2)\;\; w_3(z_3)\;\; c_3$$
$$[x_1 \ll x_3, y_1 \ll y_2, z_1 \ll z_3]$$

Figure 3-2 shows the corresponding direct serialization graph for this history. As we can see, these transactions are serializable in the order T_1; T_2; T_3.

Definition 9 : Transitive Conflicts. We also define some transitive conflicts in which the relationship among transactions holds indirectly.

Depends: We say that T_j depends on T_i in H if there is a path from T_i to T_j in DSG(H) consisting of one or more dependency edges.

Conflict-depends: We say that T_j conflict-depends on T_i in H if there is a path from T_i to T_j in DSG(H). That is, conflict-depends is the transitive closure of directly conflict-depends.

Anti-depends: We say that a transaction T_j *anti-depends* on T_i if it conflict-depends on T_i but does not depend on T_i, i.e., the path from T_i to T_j contains at least one anti-dependency edge.

3.2 Isolation Levels for Committed Transactions

We now present our specifications for the existing ANSI isolation levels. We developed our conditions by studying the motivation of the original definitions [GLPT76] and the problems that

Conflicts Name	*Description* (T_j conflicts on T_i)	Notation in DSG
Directly write-depends	T_i installs x_i and T_j installs *x*'s next version (or T_j performs a predicate-based write whose version set includes x_i) or T_i performs a predicate-based write and T_j overwrites this write	$T_i \xrightarrow{ww} T_j$
Directly read-depends	T_i installs x_i, T_j reads x_i or T_j performs a predicate-based read whose version set contains x_i	$T_i \xrightarrow{wr} T_j$
Directly anti-depends	T_i reads x_h and T_j installs *x*'s next version or T_i performs a predicate-based read and T_j overwrites this read	$T_i \overset{rw}{- - - \rightarrow} T_j$

Figure 3-1: Definitions of direct conflicts between transactions.

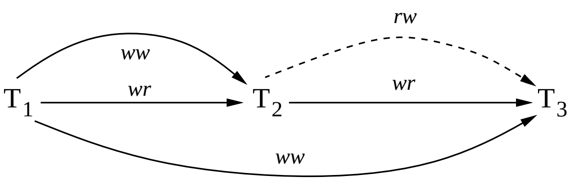


Figure 3-2: Direct serialization graph for history $H_{serializable}$

were addressed by the phenomena in [BBG$^+$95]; this enabled us to develop implementation-independent specifications that capture the essence of the ANSI definitions, i.e., we disallow undesirable situations while allowing histories that are permitted by a variety of implementations. The conditions in [GLPT76] and [BBG$^+$95] were inspired by locking and actually reflect the use of short and long read/write-locks. Therefore, we determined which abstract properties really matter, and which ones are just artifacts of a particular concurrency control technique.

Like the existing approach, we will define each isolation level in terms of phenomena that must be avoided at each level. Our phenomena are prefixed by "G" to denote the fact that they are general enough to allow locking and optimistic implementations; these phenomena are named G0, G1, and so on (by analogy with P0, P1, etc of [ANS92]). We will refer to the new levels as PL levels (where PL stands for "portable level") to avoid the possible confusion with existing names used in the ANSI-SQL definitions.

3.2.1 Isolation Level PL-1

Disallowing phenomenon P0 ensures that writes performed by T_1 are not overwritten by T_2 while T_1 is still uncommitted. There seem to be two reasons why this proscription might be desirable in a database system:

1. It simplifies recovery from aborts. In the absence of this proscription, a system that allowed writes to happen in-place could not recover the pre-states of aborted transactions using a simple undo log approach.
2. It serializes transactions based on their writes alone. That is, if T_2 writes some data item *x* after T_1 writes *x*, there should not be some other data item *y* in which the reverse occurs, i.e., all writes of T_2 must be ordered before or after all writes of T_1.

The first reason does not seem relevant to all systems. Instead, it is based on a particular implementation of recovery. A database system needs a way to prevent aborted transactions from modifying the committed state. Traditionally, this goal has been achieved by performing writes in-place and maintaining an undo log [GR93]; if the transaction aborts or the transaction manager crashes, the database is restored using the before-images stored in the undo log. For example, history $H_{recovery}$ has been presented earlier in the literature [BBG$^+$95, SWY93] and researchers have argued that it must be disallowed because it does not lead to correct recovery for such systems:

$H_{recovery}$: $w_0(x_0)$ c_0 $w_1(x_1)$ $w_2(x_2)$ a_1

In the above history, transactions T_1 and T_2 modify object *x* and then T_1 aborts (the version order is not specified since T_1 and T_2 are not committed). At this point, restoring *x* to x_0 (T_1's before-image) would be incorrect since T_2's update will be lost. Similarly, if T_2 also aborts now, restoring *x* to T_2's before-image (i.e., x_1) would also be wrong. History $H_{recovery}$ is disallowed by P0 since two uncommitted transactions T_1 and T_2 are concurrently modifying object *x*.

However, we believe that $H_{recovery}$ is a valid history and our consistency condition for no-dirty-writes (presented below) allows it. Implementations can handle the aborts of T_1 and T_2 in a variety of ways. For example, when T_1 aborts, the system can let x_2 be the version that is stored in the database. If T_2 also aborts, the system reverts the value of *x* to x_0. Furthermore, some client-server systems such as Thor [LAC$^+$96] do not install modifications into the committed state until the commit point of transactions. In such systems, we do not even have to revert the value of *x* when either transaction aborts.

Note that history $H_{recovery}$ involves blind writes (i.e., a transaction updates an object before reading it) which may be uncommon for a large class of applications. Thus, it may seem that disallowing such histories is not overly restrictive. However, recall that P0 rules out efficient optimistic and multi-version mechanisms by disallowing concurrent writes that conflict. Thus, to allow a range of concurrency control mechanisms, assumptions about a particular recovery implementation must not be made in the consistency specifications and different recovery mechanisms must be permitted.

The second reason for no-dirty-writes seems relevant to all systems and it ensures that conflicting updates are not interleaved, i.e.. transactions are serialized based on writes. This property is captured by phenomenon G0 and we define **PL-1** as the level in which G0 is disallowed:

> **G0: Write Cycles.** A history H exhibits phenomenon G0 if DSG(H) contains a directed cycle consisting entirely of write-dependency edges.

Our PL-1 specification is more permissive than degree 1 of [BBG$^+$95] since G0 allows concurrent transactions to modify the same object whereas P0 does not. Thus, non-serializable interleaving of write operations is possible among uncommitted transactions as long as such interleavings are disallowed among committed transactions (e.g., by aborting some transactions).

The lock-based implementation of PL-1 (long write-locks) disallows G0 since two concurrent transactions, T_i and T_j, cannot modify the same object; therefore, all writes of T_j either precede or follow all writes of T_i.

One might wonder whether there is any point in running a system that only supports the no-write-cycles property. As noted in [GR93], this property may be useful for applications where the modifications reflect information obtained by browsing the database to get an approximate idea of what it contains. In a department store where all customer purchases are logged online, a manager could use level PL-1 to obtain an approximation of the total value of the orders in the purchase log (assuming that uncommitted transactions change the total by a small value). PL-1 is also useful for transactions that do not want their updates to be interleaved with updates by other transactions. However, in general, it seems unwise to run read/write transactions at this level and most database systems do not allow such transactions to be executed with the no-dirty-writes guarantee only.

Discussion

We now consider some examples that show the kind of guarantees that are provided at level PL-1. Consider history H_{wcycle}:

$H_{write-cycle}$: $w_1(x_1, 20)$ $w_2(x_2, 50)$ $w_2(y_2, 50)$ c_2 $w_1(y_1, 80)$ c_1 $[x_1 \ll x_2, y_2 \ll y_1]$

In this history, any serial execution results in $x + y = 100$, but the result of $H_{write-cycle}$ is $x_2 + y_1 = 130$. This history is disallowed by PL-1 because the updates on *x* and *y* occur in opposite orders; Figure 3-3 shows the DSG for this history.

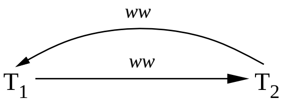


Figure 3-3: Direct serialization graph for history $H_{write-cycle}$

Here are some examples that illustrate the guarantees provided by PL-1 with respect to inserts. Suppose that transaction T_2 increments the salaries of all employees for which “Department = Sales” and another transaction T_1 adds two employees, *x* and *y*, to the Sales department. Suppose the following situation occurs:

$H_{pred-update}$: $w_1(x_1)$ $w_1(y_1)$ $w_2(\text{Dept=Sales: } x_1; y_{init})$ $w_2(x_2)$ c_1 c_2
$[x_{init} \ll x_1 \ll x_2, y_{init} \ll y_1]$

The updates of transactions T_1 and T_2 are interleaved in the above history (*x*’s salary is updated but *y*’s salary is not). There is a write-dependency edge from T_1 to T_2 since x_1 precedes x_2 in the version order. Furthermore, there is a (predicate) write-dependency edge from T_2 to T_1 because T_1 changes the matches due to T_2’s predicate (T_1 creates y_1 which matches the predicate whereas y_{init} does not). Since the DSG contains a write-dependency cycle involving T_1 and T_2, this history is disallowed by our PL-1 definitions.

3.2.2 Isolation Level PL-2

If a system disallows only G0, it places no constraints on reads: a transaction is allowed to read modifications made by committed, uncommitted, or even aborted transactions. Proscribing phenomenon P1 in [ANS92] was meant to ensure that T_1’s updates could not be read by T_2 while T_1 was still uncommitted. There seem to be three reasons why disallowing P1 (in addition to P0) might be useful:

1. It prevents a transaction T_2 from committing if T_2 has read the updates of a transaction that might later abort.
2. It prevents transactions from reading intermediate modifications of other transactions.
3. It serializes committed transactions based on their read/write-dependencies (but not their anti-dependencies). That is, if T_2 depends on T_1, T_1 cannot depend on T_2.

   This property ensures that there is unidirectional flow of information; if T_2 is affected by T_1, T_1 cannot be affected by T_2.

Disallowing P1 (together with P0) captures all three of these issues, but does so by preventing uncommitted transactions from reading or writing objects written by transactions that are still uncommitted. We address the three guarantees due to P1 by the following three phenomena, G1a, G1b, and G1c.

> **G1a: Aborted Reads.** A history H exhibits phenomenon G1a if it contains an aborted transaction T_i and a committed transaction T_j such that T_j has read some object (maybe via a predicate) modified by T_i. Phenomenon G1a can be represented using the following history fragments:
>
> $w_i(x_{i.m})\ \ldots\ r_j(x_{i.m})\ \ldots$ (a_i and c_j in any order)
> $w_i(x_{i.m})\ \ldots\ r_j(\text{P: } x_{i.m}, ...)\ \ldots$ (a_i and c_j in any order)

Proscribing G1a ensures that if T_j reads from T_i and T_i aborts, T_j must also abort; these aborts are also called *cascaded aborts* [BHG87]. In a real implementation, the condition also implies that if T_j reads from an uncommitted transaction T_i, T_j's commit must be delayed until T_i's commit has succeeded [BHG87, GR93].

Condition G1a is needed for recoverability purposes: if the system recovers after a crash and reruns all committed transactions in the same order as they ran earlier, the same committed state should be reached as it existed before the crash. However, if a committed transaction depends on an aborted transaction, this would not be possible. The problem of recoverability due to reads from aborted transactions has been discussed in the literature [BHG87, Dat90, RKS93, YBS91]; some of the suggested solutions are similar to G1a and others are the same as condition P1.

> **G1b: Intermediate Reads**. A history H exhibits phenomenon G1b if it contains a committed transaction T_j that has read a version of object *x* (maybe via a predicate) written by transaction T_i that was not T_i's final modification of *x*. The following history fragments represent this phenomenon:
>
> $w_i(x_{i.m})\ \ldots\ r_j(x_{i.m})\ \ldots\ w_i(x_{i.n})\ \ldots\ c_j$
> $w_i(x_{i.m})\ \ldots\ r_j(\text{P: } x_{i.m}, ...)\ \ldots\ w_i(x_{i.n})\ \ldots\ c_j$

Like our other conditions, G1b does not constrain the behavior of uncommitted transactions. It allows an uncommitted transaction T$_j$ to observe the effects of other uncommitted transactions as long as T$_j$ reads values that are finally installed into the database; if any of these values are intermediate, disallowing G1b ensures that T$_j$ is aborted. Note that proscribing G1a and G1b ensures that a committed transaction only reads object values that existed (or will exist) at some point in the committed state.

Disallowing intermediate and aborted reads can be crucial for many applications. For example, consider a company's web site on the internet where customers fill out online forms for their purchases. A processing agent processes these orders and informs appropriate departments. If the processing agent were allowed to see an order form while it was still being modified by a customer transaction, it might incorrectly process an incomplete order. In this case, it is important to disallow G1a as well; otherwise, the agent may process an order that never existed (i.e., if a customer aborts his/her purchase).

> **G1c: Circular Information Flow.** A history H exhibits phenomenon G1c if DSG(H) contains a directed cycle consisting entirely of dependency edges.

Intuitively, disallowing G1c says that if a transaction T$_j$ is affected by transaction T$_i$, it does not affect T$_i$, i.e., there is a unidirectional flow of information from T$_i$ to T$_j$. Note that G1c includes G0. We could have defined a weaker version of G1c that only concerned cycles having at least one read-dependency edge, but it seems simpler not to do this.

Unidirectional flow of information can be important in many situations. For example, consider an online firm that is selling tickets for a sports event. Each sales agent is structured such that it "backs-off" if it observes a claim request by another agent. Suppose two sales agents A and B make a simultaneous attempt to obtain a batch of 100 tickets. It is possible that agent A observes B's request and vice-versa. As a result, both of them back-off and neither agent obtains the tickets. If unidirectional flow of information was guaranteed, at least one of them would have succeeded in obtaining the tickets; if G1c is disallowed, one agent will observe that there are no other pending requests.

Our condition that captures the essence of no-dirty-reads is G1, which is comprised of G1a, G1b, and G1c. We define isolation level **PL-2** as one in which phenomenon G1 is disallowed. (As mentioned, disallowing G1 also disallows G0.)

Proscribing G1 is clearly weaker than proscribing P1 since G1 allows transactions to read from uncommitted transactions. The lock-based implementation of PL-2 disallows G1 because the combination of long write-locks and short read-locks ensures that if T_i reads a version produced by T_j, T_j must have committed already and therefore it cannot read a version produced by T_i.

Along with disallowing dependency cycles in the DSG, our specifications also include conditions G1a and G1b that are simply based on object histories. We could have expressed these phenomena in terms of graph properties by including aborted transactions in the graph and defining phenomenon G1a as one in which there is a read-dependency edge from an aborted transaction to a committed transaction (G1c would be defined over committed nodes only). Similarly, consider a graph in which edges are labeled by the objects causing the conflict; phenomenon G1b occurs if there exists a cycle consisting of exactly one read-dependency and one anti-dependency edge such that both edges are labeled by the same object. However, these extra nodes, edges, and labels complicate the graph unnecessarily. Thus, for simplicity reasons, conditions G1a and G1b are presented as constraints on object histories rather than being expressed as graph properties.

Consistency Guarantees for Predicate-based Reads at PL-2

The PL-2 definition given in this section treats predicate-based reads like normal reads and provides no extra guarantees for them. However, there are alternative approaches possible. Here are some consistency guarantees that could be provided to such reads at PL-2 (from weakest to strongest). Each predicate-based read is:

1. provided the same guarantees as a normal read.
2. indivisible with respect to any predicate-based write.
3. indivisible with respect to *all* writes of a transaction.

Our definition of PL-2 assumes the first option and disallows only G1a, G1b, and G1c (along with G0) for PL-2 transactions. For example, the following history is allowed by level PL-2:

$H_{pred-noguar}$: w_1(Dept=Sales: x_0; y_0) $w_1(x_1)$ $w_1(y_1)$ r_2(Dept=Sales: x_1; y_0) $r_2(x_1)$ c_1 c_2
[$x_0 \ll x_1$, $y_0 \ll y_1$]

In this history, T_1 updates the salaries of all employees in the Sales department and T_2 reads the salaries of all these employees; T_2 observes only partial effects of T_1's write.

The second guarantee disallows this history and ensures that we can treat all operations (normal/predicate-based reads and writes) as primitive operations. This guarantee essentially says that the versions in the version set of a predicate-based operation must not reflect the partial effects of a predicate-based write, i.e., predicate-based operations are not interleaved with respect to each other. This property can be supported by disallowing phenomenon G1-predA and **PL-2'** can be defined as a level that disallows G1a, G1b, G1c, and G1-predA:

> **G1-predA: Non-atomic Predicate-based Reads**. A history H exhibits phenomenon G1-predA if H contains distinct committed transactions T_i and T_j, and operations w_i(P: Vset(P)), $w_i(x_i)$ ... $w_i(y_i)$ and r_j(Q: Vset(Q)) such that $w_i(x_i)$ and $w_j(x_j)$ are events generated due to w_i(P: Vset(P)), $x_i \in$ Vset(Q), and $w_i(y_i)$ overwrites r_j(Q: Vset(Q)).

Disallowing G1-predA guarantees that if T_i's predicate-based read observes an update by T_j's predicate-based write, it does not see any version older than the ones installed by T_j's write. Thus, disallowing G1-predA ensures that all read/write operations are indivisible with respect to each other. However, level PL-2' still allows a predicate-based read to be interleaved with distinct writes of the same transaction. For example, the following history is allowed by level PL-2':

$H_{atomic-reads}$: w_1(Dept=Sales: x_0; y_0; z_0) $w_1(x_1)$ $w_1(y_1)$ $w_1(z_1)$
r_2(Dept=Sales or Dept=Legal: x_1; y_1; z_0) $r_2(x_1)$ $r_2(y_1)$ c_1 c_2
$[x_0 \ll x_1, y_0 \ll y_1, z_0 \ll z_1]$

This scenario is similar to the one shown for $H_{pred-noguar}$ except that T_1 also updates the record of an employee in the legal department (as a normal write) and T_2 reads an old version of this record as part of its predicate-based read. All operations in this history are indivisible with respect to each other but the predicate-based read is not indivisible with respect to all operations of the transaction.

The third guarantee disallows history $H_{atomic-reads}$ and ensures that a predicate-based read does not observe the partial effects of a transaction, i.e., the values observed by a predicate-based reads are not interleaved with all modifications by another transaction. Phenomenon G1-predB handles partial predicate-based reads:

> **G1-predB: Non-atomic Predicate-based Reads with respect to Transactions**. A history H exhibits phenomenon G1-predB if H contains distinct committed transactions T_i and T_j such that T_i overwrites an operation r_j(P: Vset(P)) and there exists a version x_i in Vset(P).

We define level **PL-2"** to be one that disallows G1a, G1b, G1c, and G1-predB. If predicate-based reads are treated in this manner, each predicate-based operation becomes indivisible with respect to all operations of a transaction (G0 provides this property for predicate-based writes). This level can be useful in many situations such as the one shown in history $H_{atomic-reads}$.

Even stronger guarantees are provided by a lock-based implementation for predicate-based reads. These guarantees are discussed in Section 4.2.2. Furthermore, many commercial systems execute each SQL statement atomically and hence ensure that each predicate-based read observes a database state achieved by the complete execution of some set of transactions; the issue of atomic SQL statements at lower isolation levels is discussed in Section 3.3.2.

3.2.3 Isolation Level PL-3

In a system that proscribes only G1, it is possible for a transaction to read inconsistent data and therefore make inconsistent updates. For example, consider history $H_{lost-update}$:

$H_{lost-update}$: $r_1(x_0, 10)$ $r_2(x_0, 10)$ $w_2(x_2, 15)$ c_2 $w_1(x_1, 14)$ c_1 $[x_0 \ll x_2 \ll x_1]$

This history illustrates the "lost update" problem [GR93], in which T_1 and T_2 attempt to add 4 and 5, respectively, to *x* but T_2's update is "lost" and *x* is incremented by 4 instead of 9. Although phenomenon P2 prevents such histories, it also prevents legal histories such as $H_{2'}$ (presented in Section 2.4.1) and hence, disallows many concurrency control schemes, including optimistic and multi-version concurrency schemes. What we need is to prevent transactions that perform inconsistent reads or writes from committing. This is accomplished by the following condition:

> **G2: Anti-dependency Cycles.** A history H exhibits phenomenon G2 if DSG(H) contains a directed cycle having one or more anti-dependency edges.

We define **PL-3** as an isolation level that proscribes G1 and G2. Thus, all cycles are precluded at this level; since the DSG for history $H_{lost-update}$ contains a cycle (see Figure 3-4), it is disallowed by PL-3. Of course, the lock-based implementation of PL-3 (long read/write-locks) disallows phenomenon G2 also since two-phase locking is known to provide complete serializability.

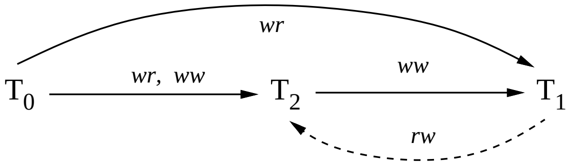


Figure 3-4: Direct serialization graph for history $H_{lost-update}$

Proscribing G2 is weaker than proscribing P2, since we allow a transaction T_j to modify object *x* even after another uncommitted transaction T_i has read *x*. Our PL-3 definition allows histories such as $H_{1'}$ and $H_{2'}$ (presented in Section 2.4.1) that were disallowed by the preventative definitions.

The conditions given in [BHG87] provides view-serializability whereas our specification for PL-3 provides conflict-serializability (this can shown using theorems presented in [GR93, BHG87]). All realistic implementations provide conflict-serializability; thus, our PL-3 conditions essentially provide what is normally considered as serializability.

3.2.4 Isolation Level PL-2.99

The level called REPEATABLE READ or degree 2.99 in [ANS92] provides less than full serializability when predicates are in use. In particular, it uses long locks for all operations except predicate-based reads for which it uses short locks, i.e., it ensures serializability with respect to regular reads and provides guarantees similar to degree 2 for predicate-based reads. Thus, anti-dependency cycles due to predicates can occur at this level. We use the following condition to allow these cycles:

> **G2-item: Item Anti-dependency Cycles.** A history H exhibits phenomenon G2-item if DSG(H) contains a directed cycle having one or more item-anti-dependency edges.

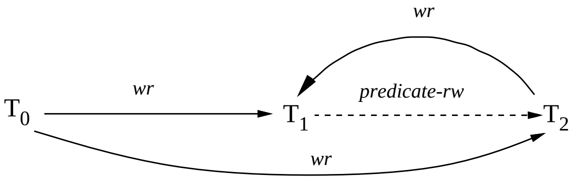


Figure 3-5: Direct serialization graph for history $H_{phantom}$

Level **PL-2.99** is defined as one that proscribes G1 and G2-item. For example, consider the following history:

$H_{phantom}$: $r_1(\text{Dept=Sales: } x_0, 10; y_0, 10)$ $r_2(\text{Sum}_0, 20)$ $w_2(z_2\text{=10 in Dept=Sales})$
$w_2(\text{Sum}_2, 30)$ c_2 $r_1(\text{Sum}_2, 30)$ c_1 $[\text{Sum}_0 \ll \text{Sum}_2, z_{init} \ll z_2]$

When T_1 performs its query there are exactly two employees, *x* and *y*, both in Sales (we show only visible versions in the history). T_1 sums up the salaries of these employees and compares it with the sum-of-salaries maintained for this department. However, before it performs the final check, T_2 inserts a new employee in the Sales department, updates the sum-of-salaries, and commits. Thus, when T_1 reads the new sum-of-salaries value, it finds an inconsistency. This situation is an example of the *read phantom* problem [GR93] where transactions observe an inconsistent state due to predicate-based reads.

The DSG for $H_{phantom}$ is shown in Figure 3-5. This history is ruled out by PL-3 but permitted by PL-2.99 because the DSG contains a cycle only if predicate anti-dependency edges are considered.

Even though a transaction can observe inconsistencies due to predicate-based reads at level PL-2.99, we can still provide strong consistency guarantees to *each* predicate-based read or even each SQL statement; such guarantees are discussed in Sections 3.2.2, 3.3.2, and 4.2.2.

3.2.5 Summary of Isolation Levels

We summarize the isolation levels discussed in this section in Figure 3-6.

Like the existing definitions for isolation levels [GLPT76, BBG$^+$95], each of our consistency conditions captures a different type of conflict. However, instead of preventing those conflicts from

Level	Phenomena disallowed	Informal Description (T_i can commit only if:)
PL-1	G0	T_i's writes are completely isolated from the writes of other transactions
PL-2	G1	T_i has only read the updates of transactions that have committed by the time T_i commits (along with PL-1 guarantees)
PL-2.99	G1, G2-item	T_i is completely isolated from other transactions with respect to data items and has PL-2 guarantees for predicate-based reads
PL-3	G1, G2	T_i is completely isolated from other transactions, i.e., all operations of T_i are before or after all operations of any other transaction

Figure 3-6: Summary of portable ANSI isolation levels

occurring at transaction execution time, our definitions place constraints on the transactions that are allowed to commit; in Section 3.5, we discuss how isolation guarantees can be specified for executing transactions as well. Each isolation level disallows cycles in the DSG corresponding to the conflicts being handled at that consistency level. PL-1 manages write-write conflicts, PL-2 also handles write-read conflicts, and PL-3 (or serializability) takes care of all types of conflicts.

Conditions G1 and G2 provide the well-known definition of serializability, i.e., we can perform a topological sort on the acyclic direct serialization graph and obtain a total order on the committed transactions. If these transactions are executed in this order, the result will be the same as when the transactions executed concurrently. Furthermore, condition G1a also handles recoverability [BHG87, KSS97]. Thus, our definition of serializability is the same as what has been generally accepted in the literature.

Using graphs for capturing multi-object constraints is a well-known technique for serializability purposes [BHG87, GR93, KSS97]; we have extended their use to lower degrees of isolation. Our definitions G0, G1c, and G2 (i.e., the cycle invariants) are similar to the conditions defined towards the end of [GLPT76]. However, since the authors wanted to give an alternate definition of locking behavior using constraints on cycles, their graph contains committed and executing transactions. As a result, their conditions disallow certain histories that are permitted by us; these situations may actually occur when optimistic concurrency control is used. Furthermore, their cycle-based conditions do not handle predicates and do not include properties G1a and G1b.

3.3 Mixing of Isolation Levels

So far, we have only discussed systems in which all transactions are provided the same guarantees. However, in general, applications may run transactions at different levels; Section 3.3.1 discusses how such "mixed" systems are modeled and discusses how these transactions interact with each other. Another form of mixing occurs in real database systems: each SQL statement in a transaction T_i is executed atomically even though T_i has been specified to execute at a lower isolation level than serializability. Issues related to modeling such systems are discussed in Section 3.3.2.

3.3.1 Guarantees to Transactions in Mixed Systems

In a mixed system, each transaction specifies its level when it starts; this information is maintained as part of the history (in our examples, we do not use any notation and simply indicate the levels of different transactions in the text) and used to construct a *mixed serialization graph* or *MSG*. Like a DSG, the MSG contains nodes corresponding to committed transactions and edges corresponding to conflicts; however, only conflicts relevant to a transaction's level or *obligatory* conflicts show up as edges in the graph. Transaction T_i has an obligatory conflict with transaction T_j if T_j is running at a higher level than T_i, T_i conflicts with T_j, and the conflict is relevant at T_j's level. For example, an anti-dependency edge from a PL-3 transaction to a PL-1 transaction is an obligatory edge since overwriting of reads matters at level PL-3.

Edges are added as follows: since write-dependencies are relevant at all levels, we retain all such edges. For a PL-2 or PL-3 node T_i, since reads are important, read-dependencies coming into T_i are added. Similarly, we add all outgoing anti-dependency edges from PL-3 transactions to other nodes.

Now we can define correctness for a mixed history:

Definition 10 : Mixing-Correct. A history H is mixing-correct if MSG(H) is acyclic and phenomena G1a and G1b do not occur for PL-2 and PL-3 transactions.

It is possible to restate the above definition as an analog of the Isolation Theorem [GR93]:

> **Mixing Theorem:** If a history is mixing-correct, each transaction is provided the guarantees that pertain to its level.

The above theorem holds at the level of a history and is independent of how synchronization is implemented. Note that the guarantees provided to each level are with respect to the MSG. The reason is that an MSG considers the presence of transactions at other levels whereas a DSG is simply constructed with all edges. As we discuss below, an MSG is useful for determining correctness if PL-1 and PL-2 transactions "know" what they are doing whereas a DSG ensures correctness without making any assumptions about the operations of lower level transactions.

A mixed system can be implemented using locking (with the standard combination of short and long read/write-locks) and all histories generated by a lock-based system are mixing-correct. Phenomena G1a and G1b cannot occur at PL-2 or PL-3 since such transactions only read the updates of committed transactions. Furthermore, there are no cycles in the MSG because of the following property: If an edge exists from T_i to T_j in the MSG, it implies that T_j commits after T_i in a lock-based implementation. For write-dependency edges, long write-locks ensure that T_j is delayed until T_i commits. A read-dependency edge from T_i to T_j is added for PL-2 and PL-3 transactions only. Since (at least) short read-locks are acquired by such transactions, a read-dependency edge from T_i to T_j implies that T_j is delayed until T_i commits. Similarly, anti-dependency edges are only considered from PL-3 transactions; a PL-3 transaction T_i acquires long read-locks thereby ensuring that if a transaction T_j overwrites T_i's updates, it will commit after T_i. Thus, if a cycle of the form $\langle$ T_1, T_2, ... T_n, T_1 $\rangle$ exists in the MSG, it leads to a contradiction that T_1 commits before T_2, ... T_n commits before T_1. Thus, the MSG must be acyclic, i.e., any history allowed by the lock-based implementation is mixing-correct.

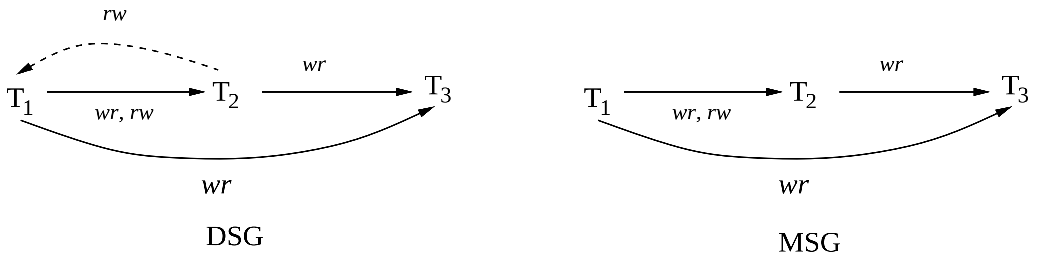
Mixed systems can also be achieved using other concurrency control techniques. For example, an optimistic implementation would attempt to fit each committing transaction into the serial order based on its own requirements (for its level) and its obligations to transactions running at higher levels, and would abort the transaction if this is not possible. An optimistic implementation that is mixing-correct is presented in Chapter 5.

Consistency of Database State

Even if a history H satisfies the mixing theorem, it does not imply that PL-3 transactions in H observe a consistent database state since lower level transactions may have modified the committed state inconsistently. This property is also true for the Isolation Theorem given in [GR93].

For example, suppose that a database contains three objects *x*, *y*, and *z* such that x_0 is 25, y_0 is 20, and z_0's value is 50. The following history is executed:

H_{mixing}: $r_1(x_0, 25)$ $r_1(y_0, 20)$ $r_1(z_0, 50)$ $r_2(x_0, 25)$ $w_1(x_1, 40)$ $w_1(y_1, 10)$ c_1 $r_2(y_1, 10)$
$w_2(z_2, 35)$ c_2 $r_3(x_1, 40)$ $r_3(y_1, 10)$ $r_3(z_2, 35)$ c_3 $[x_0 \ll x_1, y_0 \ll y_1, z_0 \ll z_2]$



DSG MSG

Figure 3-7: DSG and MSG for history H_{mixing}.

In this history, T_1 and T_3 commit at PL-3 whereas T_2 commits at PL-2. This history is possible in a lock-based implementation in which T_2 acquires short read-locks on *x* and *y* whereas long read/write-locks are acquired for all other accesses by T_1, T_2 and T_3. The DSG and MSG of history H_{mixing} is shown in Figure 3-7; for the sake of simplicity, we have not shown T_0 in the graphs. Suppose that an invariant $x + y \leq z$ is being maintained by the transactions. Both T_1 and T_2 update the database to preserve this invariant according to their reads. However, since T_2 makes decisions based on reading the partial effects of transaction T_1, it updates the database inconsistently. Thus, even though T_3 is provided PL-3 isolation, it observes a broken invariant.

As stated in [GR93], to ensure that a PL-3 transaction observes a consistent state, lower level transactions must "know" what they are doing, i.e., they must somehow update the database consistently even if they observe an inconsistent state. Similarly, an acyclic MSG ensures that transactions update and observe a consistent database state if each lower level transaction handles inconsistencies correctly in its code. Thus, even though H_{mixing} is mixing-correct, it does not result in serializable state being observed by T_3 since T_2 commits at PL-2 and destroys the database consistency. On the other hand, the DSG considers all edges and hence results in a cycle for history H_{mixing} (see Figure 3-7).

3.3.2 Guarantees to SQL Statements

In most commercial database systems, a language such as SQL is used to access and modify database objects. These systems ensure certain guarantees with respect to SQL statements executed by a transaction. We now describe how such guarantees can be provided in our framework. Consider the following SQL application code:

```
INSERT INTO Highcom
     SELECT ename, salary, commission FROM Employee
          WHERE commission > 0.25 * salary
     UNION
     SELECT ename, salary, commission FROM Manager
          WHERE dept = Sales
. . . Rest of the transaction code  . . .
```

The INSERT statement first selects employees from the Employee relation whose commission is more than 25% of their salary and all managers in the Sales department; then these selected tuples are inserted into the Highcom relation. There are *two* sub-queries executed by the INSERT statement; in general, an arbitrary number of queries may be executed as part of a SQL statement.

Suppose that there exists a tuple *x* in the Employee relation with visible version, x_0 such that the commission for x_0 is less than 25% of its salary. The Manager relation contains an object *y* corresponding to a manager in the Marketing department. Both relations may contain other objects (with unborn or visible versions) but they are not relevant for this example and we will not consider them. Now suppose that a transaction T_2 executes the SQL code shown above. A concurrent transaction T_1 increases *x*'s commission to be more than 25% of *x*'s salary; T_1 also changes *y* such that y_1 is in the Sales department. Suppose that the following execution occurs:

$H_{sql-insert}$: r_2(commission > 0.25 * salary: x_0) $w_1(x_1)$ $w_1(y_1)$
r_2(dept=Sales: y_1) $r_2(y_1)$ $w_2(z_2)$ c_1 c_2 $[x_0 \ll x_1]$

In this history, T_2's first predicate-based read misses T_1's updates (it reads x_0) but the second query observes T_1's updates (y_1 is in the Sales department). Thus, y_1 matches the predicate and the field values read from y_1 are inserted into the Highcom relation (version z_2 is the new object that is added by T_2). This history is allowed by level PL-2 and even by level PL-2" (described in Section 3.2.2); level PL-2" only ensures indivisibility of each predicate-based read (and not multiple reads) with respect to other transactions. The MSG for this history is given in Figure 3-8(a); both transactions in this history commit at PL-2.

However, most commercial database systems ensure that SQL statements execute atomically even if the transaction executes at degree 2. For example, in a lock-based implementation, read-locks are acquired (conceptually) on the Manager and Employee relation for the duration of the insert operation. Thus, in such systems, either both queries observe T_1's updates or both miss its effects, i.e., these systems will disallow history $H_{sql-insert}$.

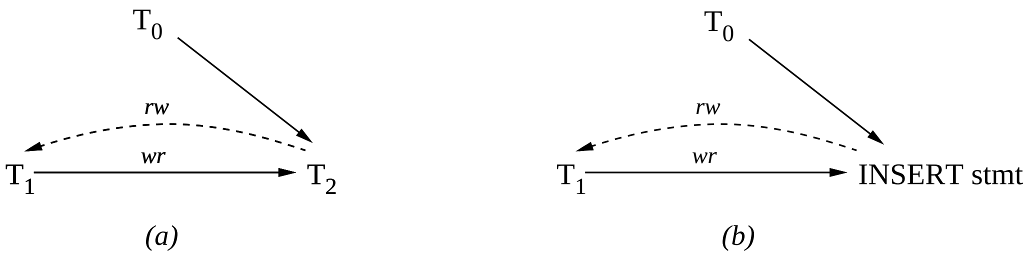


Figure 3-8: MSG and MSSG for history $H_{sql-insert}$ and the INSERT statement.

Atomic execution of SQL statements can be modeled by considering a transaction to be composed of a number of "sub-transactions" that execute at PL-3, i.e., this is a type of mixed system in which different parts of a transaction execute at different isolation levels. If the transaction executes with PL-3 guarantees, no additional constraints are needed to provide the indivisibility property of SQL statements. However, more conditions are needed for lower level transactions to provide such atomicity guarantees. For this purpose, we use a graph called *Mixed Statement Serialization Graph* or *MSSG*. For a history H and for each SQL statement *S* in a lower level transaction T_i, we define MSSG(H, S) that is similar to MSG(H): this graph contains all nodes (except T_i) and edges (except those incident on T_i) in the MSG. Then we add a node corresponding to *S* along with its edges in the graph while treating *S* as a PL-3 transaction. To ensure that all SQL statements are atomic, the following phenomenon must be disallowed for each SQL statement executed in the system (G1a and G1b must not be permitted as well):

> **G-SQL-atomic: Non-atomic SQL statements**. A history H and SQL statement S in history H exhibit phenomenon G-SQL-atomic if MSSG(H, S) contains a cycle.

This phenomena occurs for history $H_{sql-insert}$ and its insert statement; thus, database systems that provide atomicity guarantees to SQL statements will disallow this history (Figure 3-8(b) shows the MSSG).

3.4 Correctness and Flexibility of the New Specifications

Our conditions for PL-3 provide the well-known notion of conflict-serializability [BHG87, GR93]. That is, DSG(H) is acyclic, and G1a, and G1b are satisfied for a history H iff H is conflict-serializable. We can prove the equivalence of our PL-3 conditions with conflict-serializability using the proof given in [GR93]; our DSGs are similar to their graphs. This equivalence can also be proved along the lines of the proof given in [BHG87] for view-serializability.

Proving correctness for lower isolation levels is more difficult since precise guarantees for these levels have not been defined before. Thus, for PL-1 and PL-2, we analyzed behaviors that have been considered undesirable in the literature and ensured that our conditions do allow such histories.

Since our PL-3 conditions are equivalent to conflict-serializability, we know that all conflict-serializable histories are allowed at PL-3. Again, it is more difficult to prove completeness with respect to lower isolation levels. However, we can show that our specifications are strictly less restrictive than the preventative interpretation. We show this result by proving the following theorem (we say that a set of consistency definitions C1 is more *restrictive* than another set C2 if C1 permits fewer histories than C2):

Flexibility Theorem: The preventative interpretation of [BBG$^+$95] is more restrictive than our specifications for the ANSI/SQL isolation levels.

Proof:

There are two parts of the proof. We first show that there exist histories that are allowed by our definitions but are disallowed by the preventative interpretation. We use histories $H_{1'}$ and $H_{2'}$ that were presented in Section 2.4.1. We repeat those histories below for completeness (with the versions):

$H_{1'}$: $r_1(x_0, 50)$ $w_1(x_1, 10)$ $r_1(y_0, 50)$ $w_1(y_1, 90)$ $r_2(x_1, 10)$ $r_2(y_1, 90)$ c_1 c_2 $[x_0 \ll x_1, y_0 \ll y_1]$

$H_{2'}$: $r_2(x_0, 50)$ $r_1(x_0, 50)$ $w_1(x_1, 10)$ $r_1(y_0, 50)$ $r_2(y_0, 50)$ $w_1(y_1, 90)$ c_2 c_1 $[x_0 \ll x_1, y_0 \ll y_1]$

As discussed in that section, $H_{1'}$ is disallowed by condition P1 and history $H_{2'}$ is disallowed by condition P2. Our definitions allow $H_{1'}$ and $H_{2'}$ since there are no cycles formed in the direct serialization graph and conditions G1a, and G1b are not violated.

The second part of the proof requires us to show that all histories allowed by the preventative interpretation are allowed by our definitions as well. This is simple to prove for PL-3 since our PL-3 conditions allow all conflict-serializable histories whereas a lock-based implementation (recall that the preventative interpretation is essentially a disguised form of locking) only allows a subset of these histories.

For PL-1 and PL-2, we show that there exist no histories that are allowed by the preventative interpretation and disallowed by our definitions. We prove this result using contradiction for each isolation level L, i.e., we assume that there exists a history $H_{impossible}$ that is disallowed by our definitions but is allowed by the preventative interpretation at level L. We then show that this assumption leads to a contradiction.

PL-1

Consider a write-dependency edge from T_1 to T_2 in DSG($H_{impossible}$). Such an edge implies that there exists some version x_1 that was installed by T_1 and then overwritten by T_2 (i.e., $x_1 \ll x_2$). Thus, this history contains a segment of the form "$w_1(x_1) \ldots w_2(x_2)$". Since the history does not exhibit phenomenon P0, T_1 must commit before $w_2(x_2)$ occurs, i.e., T_1 commits before T_2 commits.

Since $\mathrm{H}_{impossible}$ is disallowed at PL-1, it exhibits phenomenon G0, i.e., a cycle consisting only of write-dependency edges occurs in DSG($\mathrm{H}_{impossible}$). Suppose that this cycle is of the form $\langle$ T_1, T_2, T_3, ... T_n, T_1 $\rangle$. Since P0 allows $\mathrm{H}_{impossible}$, we know (from above) that T_1 must commit before T_2 does, T_2 must commit before T_3 commits, ..., T_n must commit before T_1 commits. This leads to a contradiction.

PL-2

As in the PL-1 case, we can show that if there exists a read-dependency edge from T_1 to T_2 in DSG($\mathrm{H}_{impossible}$), T_1's commit must precede T_2's commit. Using an argument similar to the one given for PL-1, we can show that there exists no history that is disallowed by G1c at level PL-2 and is accepted by P1.

Furthermore, it is not possible that P1 is disallowed but G1a or G1b occurs in a history. The reason is that if a transaction T_j observes the updates of a committed transaction T_i, it observes the final state of objects (i.e., G1b is disallowed) and T_i cannot abort later (i.e., G1a is disallowed). □

Thus, we have shown that there exists no history that is allowed by the preventative interpretation and is disallowed by our definitions. Furthermore, since there exist histories that are allowed by our specifications but not by the preventative interpretation, our definitions are strictly less restrictive than the latter definitions.

3.5 Consistency Guarantees for Executing Transactions

All definitions presented in this chapter till now provide guarantees to committed transactions only. However, an application may require certain constraints to be valid as its transactions execute; this may be necessary to ensure that the application does not behave in an unexpected manner. In this section, we discuss how the isolation levels presented in this chapter can be extended to provide guarantees to executing transactions. To ensure that there is no confusion regarding isolation levels for committed and executing transactions, we prefix all levels for executing transactions by "E". In Section 3.5.1, we motivate the need for providing different isolation guarantees to executing transactions. Section 3.5.2 discusses the requirements on running transactions that need EPL-1 or EPL-2 and Section 3.5.3 presents the specifications for EPL-3.

3.5.1 Motivation

Suppose that a programmer writes code under the assumption that certain integrity constraints will hold. If these constraints are violated, the transaction will be aborted when it tries to commit. However, before the transaction reaches its commit point, the program may behave in an unexpected manner, e.g., it may crash, go into an infinite loop, or output unexpected results on a user's display. Debugging also becomes more difficult for an application programmer; if the transaction observes a broken invariant, it may be difficult for the programmer to determine whether the invariant was violated due to a code bug or due to weak consistency guarantees provided to executing transactions by the system. Furthermore, interactive applications become more complicated since end users may get confused on reading inconsistent data on the screen. Thus, if strong guarantees are not provided to executing transactions, a programmer must take temporary inconsistencies into account in the application code.

Here is an example in which the program goes into an infinite loop since it observes an inconsistent database state. Suppose that there are two circular lists *p* and *q* stored at servers P and Q respectively. Each node in the list has a *key* and a *data* field. The code maintains the invariant that the keys in both lists are exactly same (they may be ordered differently). Transaction T_1 adds a new key to the beginning of each list and successfully commits. Suppose that transaction T_2 executes the following piece of code:

```
% Invariant: Keys in p and q are the same
% Search circular list q for key = p.key (the first key in  circular list p)
ptr := q
while (p.key $\neq$ ptr.key) do
        ptr := ptr.next
        end
```

The programmer has written T_2's code assuming that T_2 will observe a serializable state while executing; since the invariant states that the key must exist in *q*, the code need not check for its presence. Suppose that T_2 reads the new value of *p* (i.e., p_1) and the old value of *q* (i.e., q_0). Since T_2 observes an inconsistent state (p_1 but not q_1), it will be aborted at the end. However, since the code does not find the new key in *q*, T_2's execution will never finish. The basic problem here is that T_2 has observed the partial effects of T_1; if T_2 was guaranteed to observe a consistent database state as it was executing, it would have avoided the infinite loop. □

Thus, to allow programmers to rely on code invariants, it is important that certain consistency guarantees are provided to transactions as they run. If a transaction T_i requires execution-time isolation guarantee L, the system must ensure that T_i does not detect that it is running below level L at any given instant. Transaction T_i can detect that it is running below level L if it reads objects in a manner that is not allowed by that level. If the system cannot provide degree L guarantees to T_i at some point (e.g., because the needed version has been overwritten), T_i must be aborted, e.g., if a transaction requests for serializability guarantees during execution, it must be aborted *before* it observes a non-serializable database state. (The work on orphan detection [LSWW87] in the Argus distributed system [LCJS87] also ensured that a running transaction is aborted before it observes a non-serializable state.)

Since a transaction can determine whether it is executing below a certain degree only by observing the state of the database, our conditions will provide guarantees *only for reads of uncommitted* *transactions and not for their writes*.

For the purpose of providing consistency guarantees to an executing transaction T$_i$, we consider T$_i$'s predicate-based writes as predicate-based reads. This approach is taken so that appropriate guarantees can be provided for version sets of predicate-based writes performed by T$_i$ (ghost writes performed by T$_i$ are essentially reads).

Our isolation levels for executing transactions are analogous to the levels presented for committing transactions: whatever property was formerly provided at commit time is now ensured as the transaction executes.

We assume that a transaction T$_i$ requests two isolation levels — a level L_c for committing and level L$_e$ while it is executing. We assume that level L_c is at least as strong as level L_e, e.g., a transaction can request EPL-2 while executing and serializability (but not PL-1) for committing. This is a reasonable assumption since providing strong guarantees to running transactions if the database is being updated at a lower isolation level does not make sense. It is possible to consider systems in which execution-time guarantees are stronger than commit-time guarantees. Of course, in such systems, transactions executing at higher isolation levels observe a consistent database state only if lower isolation level transactions update the database in a consistent manner. Consistency conditions for these systems can be developed in a manner similar to the approach used for committed transactions in mixed systems (Section 3.3). For simplicity, we only consider systems where the commit-time guarantees are at least as strong the execution-time guarantees.

3.5.2 Isolation Levels EPL-1 and EPL-2

Isolation level PL-1 provides guarantees with respect to writes. Since our execution time guarantees are provided only for reads, level **EPL-1** does not place *any* constraints on executing transactions.

Isolation level PL-2 guarantees that a transaction is allowed to commit if it observes values that existed in the committed state and there is unidirectional flow of information. To ensure that a transaction T$_j$ (while executing) reads from a transaction T$_i$ that does not later abort or modify the objects read by T$_j$, we disallow reads from uncommitted transactions; furthermore, no dirty reads also ensure unidirectional information flow.

We define **EPL-2** as a level that disallows phenomenon P1. EPL-2 guarantees can be useful for a transaction T$_j$ since an application can display objects on a screen as T$_j$ reads them; the programmer can be sure that the data is based on committed information. As another example, suppose that T$_i$ modifies some object *x* by making a number of incremental changes to it. A transaction running at level EPL-2 is certain to see T$_i$'s complete set of changes, rather than an intermediate state.

3.5.3 Isolation Level EPL-3

If only EPL-2 guarantees are provided, transactions can observe an inconsistent database state:

$H_{incons-view}$: $w_0(x_0)$ $w_0(y_0)$ c_0 $w_1(x_1)$ $w_1(y_1)$ c_1 $r_2(x_0)$ $r_2(y_1)$ $[x_0 \ll x_1, y_0 \ll y_1]$

In this history, T_0 and T_1 are serializable but T_2 is provided EPL-2 guarantees; T_2 observes an inconsistent view since it observes the partial effects of T_1.

To understand what is needed for the EPL-3 conditions, consider the graph for history $H_{incons-view}$ shown in Figure 3-9; note that this graph contains a node for T_2 even though it is not committed. We can see that the graph contains a cycle with a dependency edge and an anti-dependency edge. Our definition for EPL-3 will prevent such cycles and ensure that T_2 observes a serializable database state as it executes.

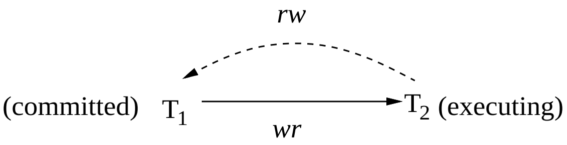


Figure 3-9: Graph for history $H_{incons-view}$ that includes a running transaction T_2 (T_0 is not shown).

To specify EPL-3 for a history H and an executing transaction T_i, we use a new graph called the *Direct Transaction Graph* that is denoted by DTG(H, T_i); recall that a DSG was defined for a history only. The DTG is exactly the same as DSG with one addition: it also contains a node for executing transaction T_i and we add all edges corresponding to the reads of T_i. Recall that we treat all predicate-based writes of T_i as predicate-based reads so that consistency guarantees can be provided for version sets of predicate-based writes. Thus, we get some extra read-dependency and anti-dependency edges due to such "reads", e.g., if executing transaction T_i performs w_i(P: Vset(P)) and Vset(P) contains x_j, we add a read-dependency edge from T_j to T_i in the DTG.

EPL-3 is defined as an isolation level that disallows phenomena P1 and E2:

> **E2: Anti-dependency Cycles at Runtime.** A history H and an executing transaction T_i exhibit phenomenon E2 if DTG(H, T_i) contains a directed cycle involving T_i that consists of dependency edges and 1 or more anti-dependency edges.

Phenomenon E2 is similar to the phenomenon G2 presented earlier for committed transactions; an execution time phenomenon similar to G2-item and a level EPL-2.99 can also be defined that does not provide serializability guarantees with respect to predicate-based reads. Note that we are ignoring anti-dependency edges due to writes by uncommitted transactions. This is in accordance with our goal of providing execution time guarantees for reads only. However, it also turns out that such writes must be ignored so that optimistic schemes can be allowed. We now show why this is the case.

To provide EPL-3, one might be tempted to say that all cycles are disallowed in the direct transaction graph. However, this condition is overly restrictive as shown by the following history:

$H_{antidep-cycle}$: $r_1(x_0)$ $w_2(x_2)$ $r_2(y_0)$ c_2 $w_1(y_1)$ $[x_0 \ll x_2, y_0 \ll y_1]$

In this example, T_1 and T_2 overwrite an object that the other transaction has read. The DTG for history $H_{antidep-cycle}$ is shown in Figure 3-10 if T_2's writes are also considered; we have also extended the version order to include y_1. Transaction T_1 is doomed to abort since a cycle with two anti-dependency edges has been formed. However, since it does not observe a non-serializable database state, there is no harm in executing it. Furthermore, this situation can occur with an optimistic scheme in a client-server distributed system where T_1 may read x_0 and write y_1 in its client's cache before information about T_2's commit arrives at T_1's client.

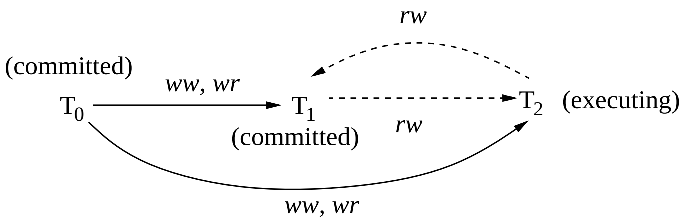


Figure 3-10: DTG for history $H_{antidep-cycle}$ and transaction T_2 if T_2's writes are also considered.

A consistency condition that considers writes and disallows situations such as the one in history $H_{antidep-cycle}$ would be unnecessarily restrictive; hence, our phenomenon E2 does not consider writes by an uncommitted transaction. Note that phenomenon P2 is overly restrictive because it does consider such writes and disallows a transaction from modifying an object if an uncommitted transaction is reading it.

3.6 Summary

In this chapter, we described our new definitions for the existing ANSI-ISO SQL-92 isolation levels. We have separated out the notion of providing guarantees for committed and running transactions so that a wide range of concurrency control implementations can be permitted. We presented correctness conditions for both types of transactions. Unlike previous work, our conditions provide a variety of guarantees for predicate-based operations in an implementation-independent manner.

To capture multi-object constraints, our consistency conditions are specified using invariants based on *all* objects accessed by a transaction. We used a graph-based approach to define different isolation levels in a simple and intuitive manner. Different types of cycles are disallowed at various isolation levels. We also use graphs for defining correctness conditions in mixed systems in which different transactions may commit at different isolation levels. Our conditions for serializability define the well-known notion of conflict-serializability. We showed that our definitions are strictly less restrictive than the preventative interpretation of the existing ANSI specifications because our specifications allow more histories. Our definitions are strong enough to rule out bad histories but are sufficiently flexible to allow a variety of concurrency control mechanisms including locking, optimistic and multi-version schemes.

Chapter 4

Specifications for Intermediate Isolation Levels

There is a wide gap between PL-2, which provides neither consistent reads nor consistent writes, and PL-3, which provides both (this is analogous to the gap between degrees 2 and 3). This chapter presents specifications for isolation levels that lie in this gap; we call these levels *intermediate degrees* of isolation. Researchers in the past have discussed the gap between degrees 2 and 3 and suggested consistency levels, such as Cursor Stability [Dat90], Read Consistency [Ora95], and Snapshot Isolation [BBG$^+$95], in this region.

We have developed two new intermediate degrees. Our first level, PL-2+, is the weakest level that provides consistent reads; it ensures that applications do not commit transactions that have observed an inconsistent state of the database. It is similar to Snapshot Isolation [BBG$^+$95, Ora95] and proscribes all phenomena that Snapshot Isolation was intended to disallow. Since PL-2+ is weaker than Snapshot Isolation, it has the potential of being implemented more efficiently than Snapshot Isolation, especially in a distributed client-server system. Our second new level, PL-2L, captures a useful monotonicity property of a lock-based implementation of PL-2; PL-2L is similar to Oracle's Read Consistency [Ora95]. This level is beneficial for legacy applications that rely on these characteristics; such applications can continue to run correctly when the system is changed to using a different concurrency control mechanism.

Along with the specifications for these two new levels, we also present implementation-independent specifications of some existing intermediate levels that have been described in the database literature or are being used in commercial databases, e.g., Snapshot Isolation, Read Consistency [Ora95], Update Serializability [GW82, HP86] and Cursor Stability [Dat90]. Our specifications are of interest since they are supported in commercial systems, and previous specifications are either given in English (e.g., Snapshot Isolation) or presented in terms of locking (e.g., Cursor Stability). We define these levels based on variations of the direct serialization graph: we add new kinds of nodes and edges as necessary and disallow certain types of cycles for each isolation level.

Figure 4-1 shows the relationships between various levels presented in this chapter and the

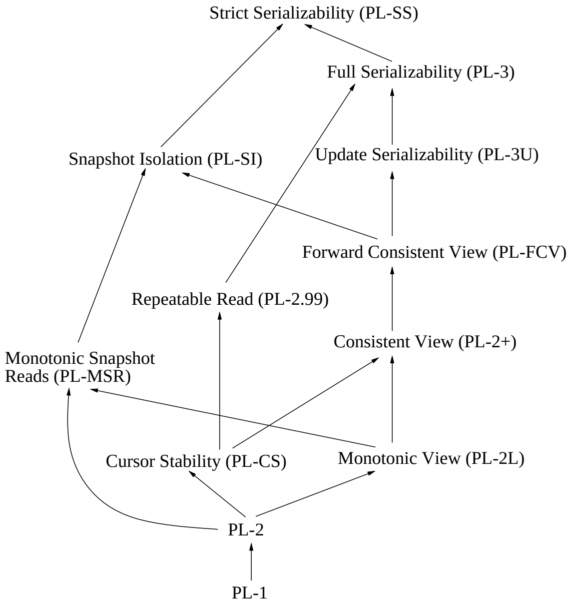

Strict Serializability (PL-SS)
Full Serializability (PL-3)
Snapshot Isolation (PL-SI)
Update Serializability (PL-3U)
Forward Consistent View (PL-FCV)
Repeatable Read (PL-2.99)
Consistent View (PL-2+)
Monotonic Snapshot Reads (PL-MSR)
Cursor Stability (PL-CS)
Monotonic View (PL-2L)
PL-2
PL-1

Figure 4-1: A partial order to relate various isolation levels.

previous chapter. Various levels can be ranked according to their "strength": one level is stronger than another if it allows fewer histories. In the figure, if level *Y* is stronger than level *X*, there is a directed path from *X* to *Y*; if there is no path between two levels, they are unrelated to each other.

For all intermediate levels, we have also developed corresponding guarantees that can be provided to transactions as they execute. As in the previous chapter, the levels defined for running transactions are similar to the corresponding levels for committed transactions.

The rest of this chapter is organized as follows. In Section 4.1, we present our specifications for PL-2+. In Section 4.2, we present definitions for PL-2L. In Section 4.3, we describe specifications of Snapshot Isolation. We discuss a new isolation level called Forward Consistent View in 4.4 that has been inspired by Snapshot Isolation. We describe a level that captures the essence of Oracle's Read Consistency in Section 4.5 and compare it with level PL-2L. Cursor Stability is presented in Section 4.6. Section 4.7 describes update serializability, a consistency guarantee that is useful for read-only transactions, and compares it with PL-2+ and serializability. Finally, in Section 4.8, we extend our definitions for intermediate levels to provide guarantees for executing transactions.

4.1 Isolation Level PL-2+

Isolation level PL-2+ is motivated by the fact that certain applications only need to observe a consistent state of the database and serializability may not be required, e.g., a read-only transaction in an inventory application may simply want to observe a consistent state of the current orders and in-stock items. It is the weakest level that ensures that integrity constraints are not observed as violated as long as update transactions modify the database consistently and are serializable.

Furthermore, the level can be implemented efficiently; an implementation that provides PL-2+ in a distributed client-server system is discussed in Section 5.3.

Consider the following history H$_{broken}$ where transaction T$_1$ observes an inconsistent state of the database:

H_{broken}: r$_1$(x$_0$, -50) r$_2$(x$_0$, -50) r$_2$(y$_0$, 100) w$_2$(x$_2$, 100) w$_2$(y$_2$, -50) c$_2$ r$_1$(y$_2$, -50) c$_1$
[x$_0$ $\ll$ x$_2$, y$_0$ $\ll$ y$_2$]

The consistency constraint, $x + y \geq 0$, is preserved by T$_2$. However, transaction T$_1$ observes a state reflecting both before and after results of T$_2$ and erroneously concludes that the constraint is broken. Figure 4-2(a) shows that history H$_{broken}$ is allowed by PL-2 because the only cycle in DSG(H$_{broken}$) involves an anti-dependency edge (for simplicity, T$_0$ is not shown); however, it is disallowed by PL-3. Note that P0 and P1 also allow H$_{broken}$.

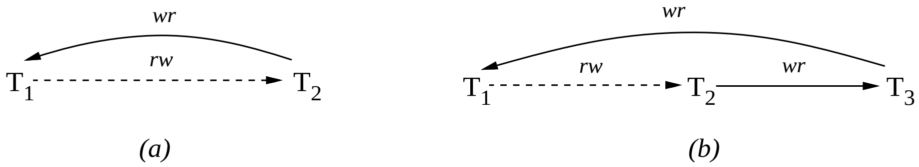


Figure 4-2: Direct serialization graph for histories H$_{broken}$ and H$_{indirect}$.

The basic problem in H$_{broken}$ is that T$_2$ has observed a modification of T$_1$ but has also missed one of T$_1$'s modifications. However, in general the missed modification might not be made by T$_1$ but instead by some earlier transaction that T$_1$ depends on. (Recall that the depends relation given in Definition 9 is the transitive closure of the directly-depends relation). For example, consider history H$_{indirect}$, in which the transactions maintain the invariant, $x \geq y$:

$H_{indirect}$: r$_1$(x$_0$, 50) r$_2$(x$_0$, 50) w$_2$(x$_2$, 100) c$_2$ r$_3$(x$_2$, 100) w$_3$(y$_3$, 75) c$_3$ r$_1$(y$_3$, 75) c$_1$
[x$_0$ $\ll$ x$_2$]

Both T$_2$ and T$_3$ maintain the invariant, but T$_1$ observes the invariant to be broken because it sees T$_3$'s effect but misses the effect of T$_2$, which T$_3$ (and hence T$_1$) depends on; the DSG for this history is shown in Figure 4-2(b) (for simplicity, T$_0$ is not shown).

4.1.1 Specification

Before we define conditions that avoid the problems of histories H$_{broken}$ and H$_{indirect}$, we first present a few assumptions and definitions.

We assume that the committed database state is consistent if the integrity constraints as defined by an application are valid. Furthermore, if an update transaction T$_i$ observes valid integrity constraints and runs alone to completion, we assume that it transforms the committed database state such that the integrity constraints continue to hold after T$_i$ commits.

Definition 11 : Basic-Consistency. A transaction T_j is provided *basic-consistency* if the values read by T_j are the result of a serial execution of some subset of committed update transactions and each update transaction in the serial execution executes the same steps as it did in the concurrent execution.

The above definition was given by Weihl [Wei87]. As discussed in [Wei87], basic-consistency ensures that T_j observes a consistent state of the database because the result of a serial execution of update transactions always results in a consistent state, by our assumption.

In basic-consistency, each update transaction T_j behaves the same (and hence results in the same write operations) in the serial and concurrent execution if it observes the same state in both executions. Thus, T_i cannot read from an arbitrary set of transactions. For example, if T_i observes the updates of T_j, it must not "miss the effects" of a transaction T_k whose updates were observed by T_j, i.e., basic-consistency requires that if the subset of transactions chosen for T_i's observed view includes T_j, it must include T_k as well. Missing the effects of a transaction can be formalized as follows:

Definition 12 : Missing Transaction Updates. A transaction T_j misses the effects of a transaction T_i if T_i installs x_i and an event $r_j(x_k)$ exists such that $x_k \ll x_i$, i.e., T_j reads a version of *x* that is older than the version that was installed by T_i.

According to the above definition, a transaction T_j will not miss the effects of T_i if T_j reads x_i or a later version of *x* (i.e., $i = k$ or $x_i \ll x_k$) or T_i and T_j do not conflict. Inconsistencies observed by transaction T_1 in histories H_{broken} and $H_{indirect}$ can be avoided if the following condition is satisfied:

Definition 13 : No-Depend-Misses. If T_j depends on T_i, it does not miss the effects of T_i.

This property (along with P1) has been shown by Chan and Gray [CG85] to ensure that transaction T_j does not observe violated integrity constraints if update transactions are serializable (i.e., commit at PL-3). We give a synopsis of this proof in Section 4.1.2. In order to ensure that transactions observe a consistent database state, we use the following condition:

> **G-single: Single Anti-dependency Cycles.** A history H exhibits phenomenon G-single if DSG(H) contains a directed cycle with exactly one anti-dependency edge.

Level **PL-2+** proscribes G1 and G-single. Intuitively, PL-2+ provides consistency because cycles with one anti-dependency edge occur exactly when some transaction both observes and misses modifications of another transaction.

We now prove that disallowing G-single is equivalent to no-depend-misses. As we show in Section 4.1.2, this implies that a transaction running at level PL-2+ is certain to see a consistent state, if update transactions are serializable and modify the database consistently.

Theorem 2+: History H does not exhibit phenomena G-single iff it satisfies the no-depend-misses property.

Proof:

Part A: Disallowing G-single implies no-depend-misses.

Suppose that a history H does not exhibit G-single but the no-depend-misses property is violated. Then there exist transactions T_i and T_j such that T_j depends on T_i, yet it misses some effect of T_i; recall that missing T_i's effects implies that history H contains events $r_j(x_k)$ and $w_i(x_i)$ such that $x_k \ll x_i$. This situation is shown in Figure 4-3 where T_u overwrites x_k and hence there is an anti-dependency edge from T_j to T_u; a "*" denotes 0 or more edges and a "+" denotes 1 or more edges. In this figure, T_i could be the same as T_u, i.e., T_i could have overwritten the version that T_j read. Thus, the DSG has a cycle with one anti-dependency edge, which is a contradiction. Therefore, if history H does not exhibit phenomenon G-single, it satisfies the no-depend-misses property.

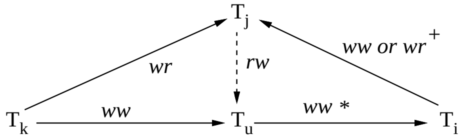


Figure 4-3: A direct serialization graph with a cycle containing one anti-dependency edge.

Part B: No-depend-misses implies that phenomenon G-single cannot occur.

Suppose that a history H satisfies no-depend-misses but exhibits phenomenon G-single. The existence of a cycle with one anti-dependency edge implies that there must be transactions T_i and T_j such that T_i depends on T_j and also T_j directly anti-depends on T_i. Because of the direct anti-dependency, there must be some object *x* such that T_j overwrote the version of *x* that T_i read, i.e., T_i missed T_j's effects even though it depends on T_j. But this contradicts the no-depend-misses property. Therefore, if H satisfies the no-depend-misses property, phenomenon G-single cannot occur in H. □

4.1.2 Relationship between PL-2+ and Basic-Consistency

We now show that PL-2+ is the *weakest* level that ensures basic-consistency provided update transactions modify the database consistently and are serializable. A proof similar to the one given below can be used to show that EPL-2+, the analog of PL-2+ for running transactions (defined in Section 4.8), ensures that application code never sees an inconsistent database state. Since basic-consistency guarantees that no violated integrity constraints are observed, PL-2+ is a useful level for committing transactions, especially read-only transactions.

Theorem: In a history that contains a set of transactions Φ, such that all update transactions in

Φ are serializable, each transaction is provided basic-consistency iff it is committed with at least PL-2+ guarantees.

Proof:

We will prove this theorem with respect to the reads of a transaction Q in Φ; Q's writes are not considered since basic-consistency is only concerned with what a transaction observes. Of course, if Q is an update transaction, it must be serializable (by assumption).

(a) PL-2+ is Sufficient: Given that all update transactions in Φ are serializable and Q is committed at level PL-2+ (at least), Q must be provided basic-consistency.

Suppose that transaction Q depends on a sequence of transactions T$_{i_1}$, T$_{i_2}$, $\ldots$ T$_{i_m}$ in Φ (these transactions are serialized from left to right); we call these transactions Q's *depend-set*. Transaction Q misses the effects of the rest of the transactions in Φ; these transactions are called Q's *missed-set*; Q may anti-depend on some of the transactions in its missed-set.

Consider any transaction T$_{i_p}$ in Q's depend-set; T$_{i_p}$'s depend-set is a subset of Q's depend-set because depends is a transitive relationship. Since T$_{i_p}$ does not depend on any transaction T$_{k_h}$ in Q's missed-set, T$_{k_h}$ has no impact on T$_{i_p}$'s behavior, i.e., T$_{i_p}$'s modifications (which are based on its reads) are unaffected by the presence or absence of any transaction in Q's missed-set. If we consider a history consisting only of T$_{i_1}$, T$_{i_2}$, $\ldots$ T$_{i_m}$, and Q (in this order), there will be no change in the database state observed by Q or by any transaction in its depend-set. Since Q commits with at least PL-2+ guarantees, none of the transactions in its depend-set could have aborted (property G1a); furthermore, G1b cannot occur for Q as well. This implies that Q is provided basic-consistency since the view observed by Q can be obtained by the serial execution of committed transactions T$_{i_1}$, T$_{i_2}$, $\ldots$ T$_{i_m}$ (in the given order) and these transactions execute the same steps in the concurrent and serial executions. □

(b) PL-2+ is Necessary: Given that all update transactions in Φ are serializable (i.e., commit at PL-3), Q is provided basic-consistency only if it is committed with at least PL-2+ guarantees.

We have to prove that Q will not be provided basic-consistency if it is committed below PL-2+. For this purpose, we will show that if any phenomenon of PL-2+ is allowed, it results in violation of basic-consistency.

Phenomena G1a and G1b: The definition of basic-consistency requires a transaction to read from a set of committed transactions. Thus, G1a and G1b must be disallowed for providing basic-consistency.

Phenomenon G1c:

Phenomenon G1c can occur only when update transactions form a cycle consisting of dependency edges. However, since we are given that all update transactions in Φ are serializable and we are only concerned with Q's reads, phenomenon G1c cannot occur in the history.

Phenomenon G-single:

Suppose that an isolation level L allows phenomenon G-single, i.e., a cycle exists in the DSG with one anti-dependency edge and *n-1* dependency edges (n > 1). We will now present a counter-example which shows that basic-consistency can be violated at level L. Consider a history containing *n* update transactions and transaction Q. Suppose that each update transaction T_i doubles the values of two objects *x* and *y*. The initial values of *x* and *y* are 5 and 7, respectively; an invariant maintained by the update transactions is $x < y$. Suppose that Q reads versions y_0 and x_n:

$H_{n-cycle}$: $r_1(x_0, 5)$ $r_2(y_0, 7)$ $w_1(x_1, 10)$ $w_1(y_1, 14)$ $\ldots$ $w_n(x_n, 5 * 2^n)$
$w_n(y_n, 7 * 2^n)$ $r_Q(y_0, 7)$ $r_Q(x_n, 5 * 2^n)$ $[x_0 \ll \ldots x_n, y_0 \ll \ldots y_n]$

Each transaction T_i maintains the invariant $x < y$ but Q observes it as broken; this history exhibits phenomenon G-single. As we can see, there exists no serial ordering of transactions that allows Q to observe the same state as in the concurrent execution. Thus, G-single must be disallowed to ensure that basic-consistency is not violated. □

4.1.3 Discussion

From a programmer's perspective, PL-2+ provides a useful guarantee since it allows an application to rely on invariants without full serializability. For example, consider histories H_1 and H_2 presented in Section 2.2 (we specify these histories along with the version numbers on objects):

H_1: $r_1(x_0, 50)$ $w_1(x_1, 10)$ $r_2(x_1, 10)$ $r_2(y_0, 50)$ c_2 $r_1(y_0, 50)$ $w_1(y_1, 90)$ c_1
$[x_0 \ll x_1, y_0 \ll y_1]$

H_2: $r_2(x_0, 50)$ $r_1(x_0, 50)$ $w_1(x_1, 10)$ $r_1(y_0, 50)$ $w_1(y_1, 90)$ c_1 $r_2(y_2, 90)$ c_2
$[x_0 \ll x_1, y_0 \ll y_1]$

In both histories, T_2 observes the invariant $x + y = 100$ as violated. If these transactions are executed at PL-2+, both histories will be disallowed and T_2 will be aborted because it has observed the partial effects of T_1.

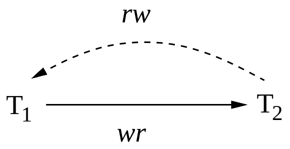


Figure 4-4: DSG for H_1, H_2, and $H_{phantom}$ where T_2 observes an inconsistent database state (for simplicity, T_0 is not shown).

Anomalous situations in both histories cause a cycle with exactly one anti-dependency edge in the DSG (see Figure 4-4). For these histories, PL-2+ is sufficient and PL-3 is not needed to ensure consistent reads. Of course, like other levels below serializability, a transaction committing at level PL-2+ may update the database inconsistently. For example, it allows histories such as H_{wskew}:

H_{wskew}: r$_1$(x$_0$, 1) r$_1$(y$_0$, 5) r$_2$(x$_0$, 1) r$_2$(y$_0$, 5) w$_1$(x$_1$, 4) c$_1$ w$_2$(y$_2$, 8) c$_2$ [x$_0$ $\ll$ x$_1$, y$_0$ $\ll$ y$_2$]

In this history, transactions break the constraint $x + y < 10$ because their operations are interleaved in a non-serializable manner; Figure 4-5 shows that the only cycle in the DSG contains two anti-dependency edges. Nevertheless, PL-2+ is useful for read-only transactions and update transactions where the application programmer knows that the writes will not destroy the consistency of the database, e.g., if the updates are performed to a private part of the database.

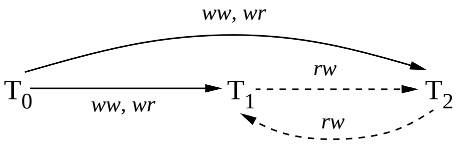


Figure 4-5: DSG for history H$_{wskew}$ that is allowed by PL-2+.

Level PL-2+ also rules out inconsistencies due to phantom reads, since these also give rise to cycles containing one anti-dependency edge. For example, it disallows history H$_{phantom}$ presented in Section 3.2.4 (the DSG for this history is shown in Figure 4-4):

$H_{phantom}$: r$_1$(Dept=Sales: x$_0$, 10; y$_0$, 10) r$_2$(Sum$_0$, 20) w$_2$(z$_2$=10 in Dept=Sales)
w$_2$(Sum$_2$, 30) c$_2$ r$_1$(Sum$_2$, 30) c$_1$ [Sum$_0$ $\ll$ Sum$_2$, z$_{init}$ $\ll$ z$_2$]

Another advantage of PL-2+ over PL-2 is that it avoids the lost updates problem. The following history (presented in Section 3.2.3 and repeated here) is disallowed by PL-2+ because the DSG contains a cycle with one anti-dependency and one write-dependency edge.

$H_{lost-update}$: r$_1$(x$_0$, 10) r$_2$(x$_0$, 10) w$_2$(x$_2$, 15) c$_2$ w$_1$(x$_1$, 14) c$_1$ [x$_0$ $\ll$ x$_2$ $\ll$ x$_1$]

PL-2+ provides a notion of "causal consistency" since it ensures that a transaction is placed after all transactions that causally affect it. Causality [Lam78] or causal consistency has been used earlier in many non-transactional systems. For example, causal memory [ANK$^+$95] provides a processor memory model for reads and writes that is similar to PL-2+; it ensures that a read/write operation *A* is ordered after the operations on which *A* causally depends. Similarly, notions of causality have been used in non-transactional settings for replication [LLSG92], atomic broadcast [BSS91] and mobile systems [KS91, TTP$^+$95].

4.2 Isolation Level PL-2L

A lock-based implementation of degree 2 (i.e., long write-locks and short read-locks) provides stronger guarantees than what is specified by degree 2 (READ COMMITTED), and transaction code in legacy applications may rely on these guarantees. Of course, applications should not make assumptions beyond the specifications of the programming interface. However, if the underlying database is changed to use a different concurrency control scheme (such as optimism), we would still like to ensure that such applications continue to work correctly. Our new isolation level, PL-2L, characterizes one such guarantee, the *lock-monotonicity property*. If a legacy application relies on just the lock-monotonicity property, it will continue to run correctly when moved to a new concurrency control implementation if transactions are executed at PL-2L. The application might be run this way initially; it can be examined later and changed to use PL-2, PL-2+ or PL-3. An efficient optimistic implementation of PL-2L in a distributed client-server system is presented in Appendix B.

4.2.1 Specification

A lock-based implementation of degree 2 provides the following lock-monotonicity property:

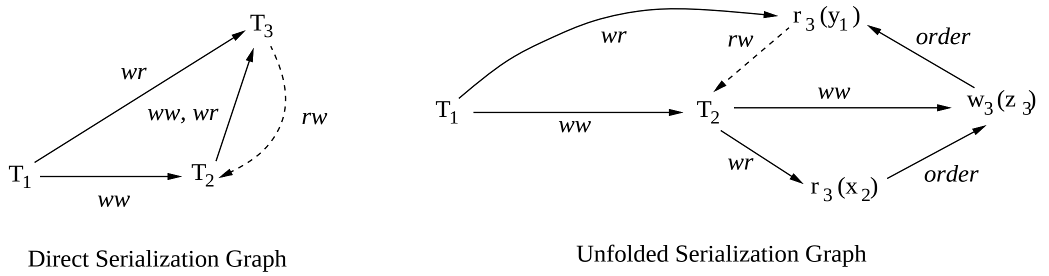
> **Lock-monotonicity.** Suppose that an event $r_i(x_j)$ exists in a history. *After* this point, T_i will not miss the effects of T_j and all transactions that T_j depends on.

This property says that a transaction observes a *monotonically increasing prefix* of the database history as it executes (in accordance with write/read-dependencies). For example, if T_i modifies objects *x* and *y*, and T_j reads x_i and then *y*, this property ensures that T_j observes y_i or a later version of *y after* reading x_i. However, if T_j reads *y* before it reads x_i, it could have read a version of *y* that is older than y_i. Thus, the lock-monotonicity property is weaker than the no-depends-misses property and does not guarantee that T_j observes a consistent database state; recall that the no-depend-misses property ensures that T_j does not miss T_i's effects *irrespective* of when T_j reads x_i.

The lock-monotonicity property is satisfied by a lock-based implementation of degree 2 (i.e., long write-locks and short read-locks) for the following reason. When transaction T_j acquires a short read-lock on object *x* that was last modified by transaction T_i, T_i must have committed by that time since write-locks are held until commit. Furthermore, any transaction T_k that T_i depends on must also have committed. Suppose that T_j reads an object *y* that was also modified by T_i (or T_k). At this point, the database must contain a version of *y* no earlier than y_i (or y_k). Thus, when T_j acquires a read lock on *y*, it will either read y_i (or y_k), or a later version of *y*. But in any case, it will not read a version of *y* produced before T_i (or T_k) committed.

We now define the lock-monotonicity property for a transaction T_i in terms of graph-based conditions. We use a graph called the *Unfolded Serialization Graph* or *USG* that is a variation of the DSG. The USG is specified for the transaction of interest, T_i, and a history, H, and is denoted by USG(H, T_i); recall that a DSG is specified over a history. For the USG, we retain all nodes and edges of the DSG except for T_i and the edges incident on it. Instead, we *split* the node for T_i into multiple nodes — one node for every read/write event in T_i. The edges are now incident on the relevant event of T_i. Here are the details on how USG(H, T_i) is obtained by transforming DSG(H).

For each node *p* ($p \neq T_i$) in DSG(H), we add a node to USG(H, T_i). For each edge from node *p* to node *q* in DSG(H), where *p* and *q* are different from T_i, we draw a corresponding edge in



Direct Serialization Graph

Unfolded Serialization Graph

Figure 4-6: USG and DSG for history H_{non-2L} that is disallowed by level PL-2L.

USG(H, T_i). Now we add a node corresponding to every read and write performed by T_i; these nodes are called *read nodes* and *write nodes* respectively. Any edge that was incident on T_i in the DSG is now incident on the relevant event of T_i in the USG, e.g., if $r_i(x_j)$ (or r_i(P: x_j, ...)) exists in H, a read-dependency edge is added from T_j to $r_i(x_j)$ (or r_i(P: x_j, ...)) in the USG. Finally, consecutive events in T_i are connected by *order edges*, e.g., if an action (e.g., SQL statement) reads object y_j and immediately follows a write on object x in transaction T_i, we add an order-edge from $w_i(x_i)$ to $r_i(y_j)$. These edges are needed to maintain the order of events in a transaction; we denote such order edges between events p and q as $p \xrightarrow{order} q$. Here is a sample history and USG(H_{non-2L}, T_3) is shown in Figure 4-6:

H_{non-2L}: $w_1(x_1)$ $w_1(y_1)$ c_1 $w_2(y_2)$ $w_2(x_2)$ $w_2(z_2)$ $r_3(x_2)$ $w_3(z_3)$ $r_3(y_1)$ c_2 c_3
[$x_1 \ll x_2$, $y_1 \ll y_2$, $z_2 \ll z_3$]

Since the lock-monotonicity property is defined from the perspective of a particular transaction T_i as it executes, we define PL-2L with respect to T_i. Isolation Level **PL-2L** for transaction T_i is defined such that phenomena G1 and G-monotonic are disallowed:

> **G-monotonic: Monotonic Reads.** A history H exhibits phenomenon G-monotonic for transaction T_i if there exists a cycle in USG(H, T_i) containing exactly one anti-dependency edge from a read node $r_i(x_j)$ (or r_i(P: x_j, ...)) to some transaction node T_k (*and* any number of order or dependency edges).

Disallowing phenomenon G-monotonic is identical to the lock-monotonicity property given above and their equivalence can be proved in a way similar to Theorem 2+ where we showed that the no-depend-misses condition and disallowing G-single are identical. Here is a brief argument for one direction: if G-monotonic is disallowed, the lock-monotonicity property must be satisfied. Suppose that the lock-monotonicity property is violated such that after a transaction T_j reads x_i, it misses the effects of T_i or some transaction T_l that T_i depends on, i.e., there is a history subsequence of the form "$r_j(x_i)$ ... $r_j(y_k)$" and $y_k \ll y_i$ (or $y_k \ll y_l$). This situation implies that an anti-dependency edge exists from $r_j(y_k)$ to T_i (or T_l). Since a path containing dependency and order edges from T_i to $r_j(y_k)$ also exists, this history exhibits phenomenon G-monotonic, i.e., a contradiction.

4.2.2 Consistency Guarantees for Predicate-based Reads at PL-2L

In Section 3.2.2, we discussed possible guarantees for predicate-based reads at level PL-2. The following consistency guarantees can also be provided to such reads:

- If a predicate-based read observes the effects of transaction T_i, it observes the complete effects of T_i and all transactions that T_i depends on.
- Each predicate-based read executes as a PL-3 transaction.

The first guarantee is provided by level PL-2L because of the following reason. When a transaction T_i performs a read based on a predicate P, we represent this read event, r_i(P: Vset(P)), in USG(H, T_i) by a single read node; this node captures conflicts with respect to all object versions accessed by r_i(P: Vset(P)). Since G-monotonic ensures that there is no single anti-dependency cycle originating from r_i(P: Vset(P)), PL-2L ensures that this event does not miss the effects of some transaction T_j that it depends on, i.e., r_i(P: Vset(P)) observes a consistent view of the database (assuming that transactions modify the database consistently). In the extreme case, when transaction T_i contains only one action, the whole transaction observes a consistent database state, i.e., we get isolation level PL-2+. Isolation level PL-2+ provides stronger guarantees for normal and predicate-based reads; it ensures that if a read by transaction T_j observes T_i's effects, *all* reads by T_j observe the complete effects of T_i and all transactions that T_i depends on.

The second guarantee is stronger than the first guarantee and is provided by a lock-based implementation since reads based on a predicate are performed after the predicate read-lock has been acquired. This guarantee can be provided by considering each predicate-based read as a sub-transaction that requires PL-3. A correctness condition similar to the one discussed in 3.3.2 for SQL statements can be used for providing atomicity guarantees to predicate-based reads.

We can also provide guarantees to predicate-based writes such that the version set of such operations is consistent. For this purpose, we can treat predicate-based writes as predicate-based reads and add the corresponding edges in the USG for these operations, e.g., if transaction T_i performs w_i(P: Vset(P)) and Vset(P) contains x_j, we can treat w_i(P: Vset(P)) as a predicate-based read, r_i(P: Vset(P)), and add a read-dependency edge from T_j to r_i(P: Vset(P)) in the USG.

4.2.3 Discussion

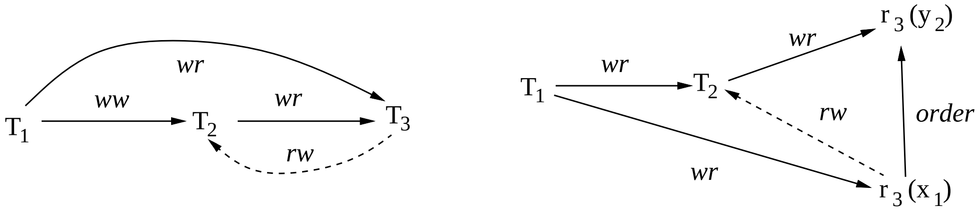
History H_{non-2L} given in Section 4.2.1 is not allowed by PL-2L since there is a cycle with one anti-dependency edge from $r_3(y_1)$ to T_2; T_3 reads x_2 but misses T_2's effects after this point (see Figure 4-6). If T_3 had read y_2, PL-2L would have allowed H_{non-2L}.

Let us consider history $H_{monotonic}$ that is allowed by PL-2L but not by PL-2+:

$$H_{monotonic}\text{:}\ \ w_1(x_1)\ \ c_1\ \ w_2(y_2)\ \ w_2(x_2)\ \ r_3(x_1)\ \ r_3(y_2)\ \ c_2\ \ c_3 \qquad [x_1 \ll x_2]$$

Figure 4-7 shows that phenomenon G-monotonic does not occur but G-single does occur (since there is a cycle in the DSG with one anti-dependency edge). Level PL-2L permits this history since T_3 does not miss the effects of T_2 *after* it reads y_2 (although it has already missed reading x_2); level PL-2+ does not allow T_3 to miss T_2's effects irrespective of when T_3 becomes dependent on T_2.

In Figure 4-1, we have shown that level PL-2L is incomparable with level PL-2.99. The reason is that PL-2L ensures consistent views to each predicate-based read whereas PL-2.99 does not. If required, such guarantees can be provided at level PL-2.99 as well, e.g., we may want each predicate-based read to be treated as a PL-3 transaction at level PL-2.99.



Direct Serialization Graph

Unfolded Serialization Graph

Figure 4-7: DSG and USG for history $H_{monotonic}$ that is accepted at PL-2L but rejected at PL-2+.

PL-2L is useful for more than just legacy code: it allows code to rely on invariants after observing a particular value of an object. For example, consider a stock exchange system where a record is maintained for each stock. There is also a record that stores the status of the market, i.e., open or closed. Suppose that the stock exchange commission runs a transaction that stores the final information about each stock in the stock records and closes the stock market. Now a brokerage firm runs a transaction T_i that observes that the market is closed. Transaction T_i then obtains information about certain stocks, reads some customer records, and generates summary information for the customers. If T_i asks for PL-2L guarantees, the system will ensure that T_i can commit only if it observes today's final stock information; the monotonicity property ensures that T_i does not miss the final stock value *after* observing that the market is closed. PL-2 is insufficient here since it does not provide this guarantee; strong guarantees of PL-2+ may not be needed since the customer records are unrelated to the stock information.

A lock-based implementation of degree 2 (i.e., short read-locks and long write-locks) actually provides stronger guarantees than monotonicity. For example, it ensures that when a transaction reads an object *x*, it observes the latest version of *x* in the committed state. We chose not to specify an isolation level that also provides this guarantee since the latest-version property seems less interesting than monotonicity and we believe that most applications executing at degree 2 do not rely on it. At degree 2, reading the current version does not prevent the object from being modified before the transaction commits; therefore, application code cannot really tell the difference between the current version or a slightly earlier one. Furthermore, more efficient implementations are possible for the lock-monotonicity property than for the latest-version property. There are other guarantees that are provided by the lock-based implementation of degree 2. However, we chose not to specify them since they are complicated and seem less useful than the lock-monotonicity property. Furthermore, we believe that application writers should be discouraged from relying on complicated assumptions about the implementation of the underlying system.

4.3 Snapshot Isolation

Snapshot Isolation was first defined in [BBG$^+$95] and is provided by Oracle [Ora95]. The definition in [BBG$^+$95] is both informal and operational: it comes very close to describing how an implementation would work. It is defined as follows:

Definition 14 : Snapshot Isolation. A transaction T$_1$ executing with Snapshot Isolation always reads data from a *snapshot* of committed data valid as of the (logical) time T$_1$ started, called the *start-timestamp*. (The snapshot could be at the time when T$_1$ started or some point in logical time before it.) Updates of other transactions active after T$_1$ started are not visible to T$_1$. When T$_1$ is ready to commit, it is assigned a *commit-timestamp* and allowed to commit if no other concurrent transaction T$_2$ (i.e., one whose active period [start-timestamp, commit-timestamp] overlaps with that of T$_1$) has already written data that T$_1$ intends to write; this is called the *First-committer-wins* rule to prevent lost updates.

4.3.1 Specification

To specify Snapshot Isolation, we need to add a component to the history, a *time-precedes order*, to capture the notion of how the start of a transaction relates to commits of other transactions. When a transaction T$_i$ starts (e.g., at its first event), the system selects a *start point*, s$_i$, for it and determines the ordering between this start point and the commits of all other transactions. Transaction T$_i$'s start point need not be chosen after the most recent commit when T$_i$ started, but can be selected to be some (convenient) earlier point. For example, the system might keep track of all transactions whose updates must not be observed by T$_i$; in this case, T$_i$'s start point is chosen before the commits of any transaction in this set. The system's decision about choosing the start point is captured in the time-precedes order:

Definition 15 : Time-Precedes Order. The time-precedes order, $\prec_t$, is a partial order specified for history H such that:

1. $s_i \prec_t c_i$, i.e., the start point of a transaction precedes its commit point.
2. for all i and j, if the scheduler chooses T$_j$'s start point after T$_i$'s commit point, we have $c_i \prec_t s_j$; otherwise, we have $s_j \prec_t c_i$.

Definition 16 : Concurrent Transactions. Two transactions T$_i$ and T$_j$ are concurrent if $s_i \prec_t c_j$ *and* $s_j \prec_t c_i$. Thus, concurrent transactions overlap; neither starts after the other one commits.

To capture the system's choice of ordering start and commit events of different transactions, we include the time-precedes order in a history (along with the version order and the partial order of events). For convenience, in our examples we will only show time-precedes constraints of the type $c_i \prec_t s_j$; no relationship is shown for a pair of concurrent transactions T$_a$ and T$_b$, i.e., $s_a \prec_t c_b$ and $s_b \prec_t c_a$ holds for these cases. Here is an example history to illustrate the above definitions:

$H_{si-example}$: w$_1$(x$_1$) c$_1$ w$_3$(z$_3$) c$_3$ r$_2$(x$_1$) r$_2$(z$_0$) c$_2$ [z$_0$ $\ll$ z$_3$; c$_1$ $\prec_t$ s$_2$]

In this history, the time-precedes order shows that T$_3$ is concurrent with T$_1$ and T$_2$ (as stated above, we do not specify such orderings) whereas T$_2$ follows T$_1$. History H$_{si-example}$ demonstrates an important point: the order of events in the history *does not* specify the relationship between commit and start points of transactions. For example, T$_3$'s commit occurs in the history before T$_2$'s first event but s$_2$ is not chosen to be after c$_3$ in the time-precedes order.

Now we can restate the definition of Snapshot Isolation (Definition 14 above) in terms of two properties, Snapshot Read and Snapshot Write, that completely define it. These properties will be used later to prove that our specification is correct.

Definition 17 : Snapshot Read. All reads performed by a transaction T$_i$ occur at its start point. That is, if r$_i$(x$_j$) occurs in history H, then:

1. $c_j \prec_t s_i$, and
2. if w$_k$(x$_k$) also occurs in H (*j* $\neq$ *k*), then either
   (a) $s_i \prec_t c_k$, or
   (b) $c_k \prec_t s_i$ and x$_k$ $\ll$x$_j$

Definition 18 : Snapshot Write. If T$_i$ and T$_j$ are concurrent, they cannot both modify the same object. That is, if w$_i$(x$_i$) and w$_j$(x$_j$) both occur in history H, then either $c_i \prec_t s_j$ or $c_j \prec_t s_i$. This is the first-committer-wins property.

We now define the notion of start-dependency and a *Start-ordered Serialization Graph* or *SSG*.

Definition 19 : Start-Depends. T$_j$ *start-depends* on T$_i$ if $c_i \prec_t s_j$, i.e., if it starts after T$_i$ commits.

Definition 20 : Start-ordered Serialization Graph or SSG. For a history H, SSG(H) contains the same nodes and edges as DSG(H) along with start-dependency edges.

We will represent start-dependency edges in an SSG as solid arrows labeled with "s". Here is an example history that is allowed by Snapshot Isolation; its SSG is given in Figure 4-8:

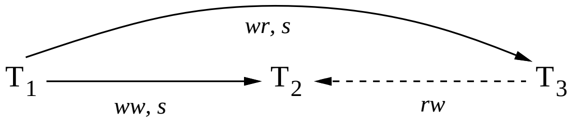


Figure 4-8: Start-ordered serialization graph for history H_{SI}

H_{SI}: $w_1(x_1)$ $w_1(y_1)$ c_1 $w_2(x_2)$ $r_3(x_1)$ $w_2(y_2)$ $r_3(y_1)$ c_2 c_3 $[x_1 \ll x_2, y_1 \ll y_2; c_1 \prec_t s_2, c_1 \prec_t s_3]$

Since the start point for T_3 is before the commit of T_2, Snapshot Isolation ensures that T_3 does not observe T_2's modifications.

Now we define phenomenon G-SI that has two parts. The first part ensures that concurrent transactions do not observe one another's effects and do not modify the same objects:

> **G-SIa: Interference.** A history H exhibits phenomenon G-SIa if SSG(H) contains a read/write-dependency edge from T_i to T_j without there also being a start-dependency edge from T_i to T_j.

This property constrains only concurrent transactions, since start-dependency edges always exist between non-concurrent transactions.

The second property ensures that a transaction T_i observes the complete effects of transactions that committed before it started (i.e., those transactions whose commit point is chosen before T_i's start point):

> **G-SIb: Missed Effects.** A history H exhibits phenomenon G-SIb if SSG(H) contains a directed cycle with exactly one anti-dependency edge.

This property captures the rest of the Snapshot Read requirement since if a transaction T_i does not observe the updates of a transaction that committed before T_i started, it will cause a cycle in the SSG with a single anti-dependency edge and a start-dependency edge. G-SIb is similar to condition G-single but provides stronger guarantees because of the extra start-dependency edges in the SSG.

Like the no-depend-misses and lock-monotonicity properties, G-SIb can be expressed in more "operational" terms, i.e., if a transaction T_j depends on T_i, T_j must not miss the effects of T_i *and* all transactions that committed before T_i started. The equivalence of this operational definition and G-SIb can be shown in a manner similar to Theorem 2+.

Property G-SI consists of G-SIa and G-SIb, and level **PL-SI** proscribes G1 and G-SI. Since G-SIb is strictly stronger than G-single, PL-SI is strictly stronger than PL-2+. Like our earlier definitions, PL-SI is also specified only for committed transactions (see Appendix A for execution time guarantees); furthermore, reads from uncommitted transactions are allowed by this specification also (as long as G1 is satisfied). We now show that our specification for level PL-SI is correct.

Theorem 2: A history H consisting of committed transactions executes under Snapshot Isolation iff G1 and G-SI are disallowed.

Proof:

Part A: If a history H executes under Snapshot Isolation, G1 and G-SI are disallowed.

Suppose G-SIa is allowed, i.e., there is a read/write-dependency from some T_i to T_j without a corresponding start-dependency. If there is no start-dependency, T_i and T_j must be concurrent and neither a read-dependency (by Snapshot Read property) nor a write-dependency (by Snapshot Write property) can exist between T_i and T_j. So we have a contradiction and G-SIa must be disallowed.

Now suppose that G1 or G-SIb is allowed and SSG(H) contains a cycle with 0 or 1 anti-dependency edges. Let this cycle have the form $\langle$ T_1, T_2, T_3, ... T_n, T_1 $\rangle$. We know the following facts about a SSG generated under PL-SI:

1. If there is a (start/read/write) dependency edge from T_i to T_j, $c_i \prec_t s_j$.
2. If there is an anti-dependency edge from T_i to T_j, $s_i \prec_t c_j$. This is because an anti-dependency edge implies that T_i did not see T_j's update. Therefore, we cannot have $c_j \prec_t s_i$; by property (2) of Definition 15, we must have $s_i \prec_t c_j$.

If there is a dependency edge (i.e., start/read/write-dependency) from T_i to T_{i+1}, we must have $c_i \prec_t s_{i+1} \prec_t c_{i+1}$ and hence $c_i \prec_t c_{i+1}$. Thus, if there are no anti-dependency edges in the cycle, we get $c_1 \prec_t c_2 \ldots \prec_t c_1$, which is impossible, and therefore G1 is disallowed. So, there must be at least one anti-dependency edge in the cycle. Without loss of generality, suppose this edge is from T_1 to T_2 and the rest are dependency edges. Then we have: $s_1 \prec_t c_2 \prec_t c_3 \ldots c_n \prec_t s_1$, i.e., $s_1 \prec_t s_1$, which again is impossible, and therefore G-SIb is disallowed.

Therefore, if history H executes under Snapshot Isolation, G1 and G-SI will be disallowed.

Part B: If G1 and G-SI are disallowed for history H, H must have executed under Snapshot Isolation.

We will now show that G1 and G-SI ensure that the Snapshot Read and Snapshot Write properties are satisfied. Suppose Snapshot Write is not satisfied. Then H contains two concurrent transactions T_i and T_j such that one of them, say T_i, overwrites T_j's modification of some object *x*. This implies that a write-dependency edge exists from T_j to T_i, but no start-dependency edge, which violates G-SIa. Therefore, the Snapshot Write property is satisfied.

Now suppose that the Snapshot Read property is violated. Property G-SIa guarantees that a transaction can only observe modifications of transactions that committed before it started (using the same argument as was given above for Snapshot Write); this handles properties (1) and (2a) of Snapshot Read. Now we consider the other part of Snapshot Read concerning missed updates (Snapshot Read 2b). Suppose transaction T_i reads *x* but does not observe the updates to *x* made by transactions that committed before T_i started; assume T_k made the first of these updates. This means that there exists an anti-dependency edge from T_i to T_k. Furthermore, since $c_k \prec_t s_i$, there exists a start-dependency edge from T_k to T_i. Therefore, phenomenon G-SIb exists in the history, which is a contradiction.

Therefore, level PL-SI provides Snapshot Isolation. □

4.3.2 Discussion

PL-2+ disallows all phenomena that Snapshot Isolation was intended to disallow [BBG+95], i.e., consistent reads, including no read phantoms ($H_{phantom}$ on page 73), and no lost updates ($H_{lost-update}$ on page 73).

It is possible that what is really desired from Snapshot Isolation is exactly what PL-2+ provides. Since PL-2+ is weaker than Snapshot Isolation (the latter requires a snapshot to be observed whereas PL-2+ just requires transactions to observe causally consistent views), it has the potential of being implemented more efficiently than Snapshot Isolation, especially in distributed client-server systems, e.g., extra communication may be required to ensure that a client observes the database state as it existed at the same logical time value at different servers. Thus, it may be desirable to use PL-2+ instead of Snapshot Isolation. (An efficient optimistic implementation of PL-2+ in a distributed client-server system is described in Section 5.3.)

Snapshot Isolation is incomparable with PL-3. It both accepts some non-serializable histories, and rejects certain serializable histories. Some serializable histories are rejected because of the Snapshot Write property. This property was introduced to rule out lost updates; however, it also prevents blind writes by concurrent transactions. (Recall that a blind write occurs when a transaction modifies an object without first reading it.) For example, the following history is not permitted by Snapshot Isolation (phenomenon G-SIa occurs) but is allowed by PL-3:

$H_{blind-nonSI}$: $r_1(x_0)$ $r_2(x_0)$ $w_1(z_1)$ $w_2(z_2)$ c_1 c_2 $[z_1 \ll z_2;\ c_0 \prec_t s_1, c_0 \prec_t s_2]$

Since blind writes are rare, history $H_{blind-nonSI}$ is unlikely to occur (but it shows that PL-SI and PL-3 are incomparable). Note that blind writes by non-concurrent transactions are allowed by Snapshot Isolation.

The Snapshot Read property also rules out certain serializable histories. It prohibits a transaction T_j from reading a modification made by a transaction T_i that committed after the start point chosen for T_j or from missing updates of transactions that committed before T_j's start point. For example:

$H_{serial-nonSI}$: $w_1(x_1)$ c_1 $r_2(x_0)$ c_2 $[x_0 \ll x_1;\ c_0 \prec_t s_1, c_1 \prec_t s_2]$

This history is serializable in the order T_0; T_2; T_1 but Snapshot Isolation disallows it since T_2 misses T_1's updates even though $c_1 \prec_t s_2$. Note that the system could have chosen a different time-precedes order (e.g., in consonance with the serial order) and allowed the above history. This is similar to the case with version orders where the database system could choose a version order that allows a history to be serializable. However, our conditions simply check for PL-3 or PL-SI based on the version and time-precedes orders chosen by the system and hence disallow $H_{serial-nonSI}$ under Snapshot Isolation.

Real Time Guarantees

The definition of Snapshot Isolation has a notion of logical time that we have captured with the time-precedes order. If this ordering is consistent with real-time, an application can be provided useful guarantees that are not provided by PL-2+ or even PL-3. For example, if a client runs a query that requests a snapshot of stock values and the system returns the values as of 4:00 pm, Snapshot Isolation would guarantee that the client observes the final values of all stocks at that time. Consider the following history:

$H_{stock-SI}$: $w_1(x_1, \$100)$ c_1 $w_2(y_2, \$50)$ c_2 $r_3(x_1, \$100)$ $r_3(y_2, \$50)$ c_3
$[x_0 \ll x_1;\ c_0 \prec_t s_3,\ c_1 \prec_t s_3, c_2 \prec_t s_3]$

In this case, T_1 updates the database at 3:58 pm, T_2 at 3:59 pm, the stock market closes at 4:00 pm and T_3's start point is 4:00 pm. Transaction T_3 reads the final values of *x* and *y*. The SSG for $H_{stock-SI}$ is in Figure 4-9(a); it shows that this history will be allowed by Snapshot Isolation. Thus, a Snapshot Isolation implementation based on real-time will disallow the following history in which T_3 observes an old value of *x* and a new value of *y*:

$H_{stock-nonSI}$: $w_1(x_1, \$100)$ c_1 $w_2(y_2, \$50)$ c_2 $r_3(x_0, \$99)$ $r_3(y_2, \$50)$ c_3
$[x_0 \ll x_1;\ c_0 \prec_t s_3,\ c_1 \prec_t s_3, c_2 \prec_t s_3]$

It is interesting to note that Snapshot Isolation disallows $H_{stock-nonSI}$ even though it is serializable in the order T_2; T_3; T_1. This history is disallowed because the time-precedes order is chosen according to the occurrence of events in real-time but T_3 does not read the effects of all transactions that committed before T_3's start point in real-time. Figure 4-9(b) shows that the only cycle in the SSG for history $H_{stock-nonSI}$ involves a start-dependency edge. In this example, if the implementation had chosen a different time-precedes order (i.e., not added the constraint $c_1 \prec_t s_3$), the resulting history would been allowed by Snapshot Isolation.

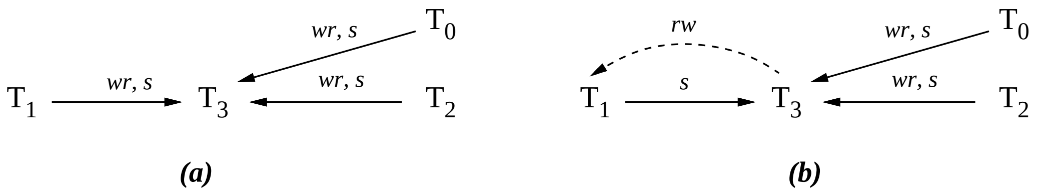


Figure 4-9: The SSGs for histories $H_{stock-SI}$ and $H_{stock-nonSI}$.

Some applications may work correctly with database snapshots in logical time whereas others may require stronger real-time guarantees (as in the example given above). Thus, it would be useful for application programmers if a Snapshot Isolation implementation indicates whether it uses real-time for ordering transactions or not. Of course, like the informal definition given in [BBG+95], our specification of Snapshot Isolation does not include this real-time constraint to provide more flexibility for implementations.

4.4 Forward Consistent View

The stronger requirements of Snapshot Isolation lead to the question of whether everything in the description in [BBG$^+$95] is required. That description is operational and it could easily be the case that some details of what that implementation provides do not matter.

We now discuss how some constraints of Snapshot Isolation can be removed to obtain a weaker isolation level that is still useful. Snapshot Read requires that all reads occur at some point in logical time before a transaction started. Not missing the effects of transactions that committed before a transaction's start point can be a useful property for some applications. We define an isolation level called *Forward Consistent View* or **PL-FCV** that provides this property by precluding G1 and G-SIb. Since G-SIa is not precluded, a transaction T_i is allowed to observe the updates of transactions that commit after it started, i.e., read "forward" beyond the start point. However, these reads are only permitted as long as T_i observes a consistent database state. PL-FCV is strictly stronger than PL-2+ because G-SIb is strictly stronger than G-single (G-SIb is defined on an SSG).

Since PL-FCV is weaker than PL-SI, it can allow more concurrency than PL-SI, which can improve performance. For example, consider:

$$H_{FCV}\text{:}\quad w_1(x_1)\; r_2(x_1)\; r_2(y_0)\;\; w_3(y_3)\; c_3\; r_4(y_3)\; r_4(x_0)\; c_1\;\; c_2\; c_4 \qquad [x_0 \ll x_1,\, y_0 \ll y_3;\, c_3 \prec_t s_4]$$

This history is not allowed under PL-SI since T_2 observes updates of transaction T_1 that is concurrent with T_2 (phenomenon G-SIa occurs). However, this history is permitted under Forward Consistent View because G1 and G-SIb do not occur. The SSG for H_{FCV} is shown in Figure 4-10 (for simplicity, T_0 is not shown).

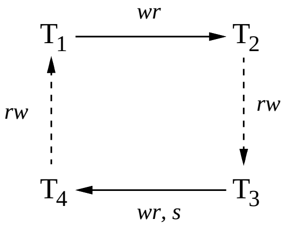


Figure 4-10: The SSG for history H_{FCV} shows that G-SIb is not violated.

4.5 Monotonic Snapshot Reads

We now present implementation-independent specifications of a level called *Monotonic Snapshot Reads* or *PL-MSR* that captures the essence of Oracle's Read Consistency; however, we do not claim that PL-MSR captures all aspects of Read Consistency necessarily.

Oracle's Read Consistency [Ora95] is stronger than degree 2: along with committed reads, this level ensures that each action (e.g., SQL statement) in a transaction T_i observes a snapshot of the database state as it existed before the action started (in logical time). Furthermore, a later action of T_i observes a snapshot that is at least as recent as the snapshot observed by an earlier action of T_i. Read Consistency can provide a different snapshot to every action, whereas Snapshot Isolation provides the same snapshot to all actions in a transaction.

The lock-monotonicity property of PL-2L is similar to the guarantees provided by Read Consistency. We use those ideas for our specifications of Monotonic Snapshot Reads. For a history H and transaction T_i, we construct a graph, *SUSG(H, T_i)* or the *Start-ordered Unfolded Serialization Graph*, that is essentially a combination of the SSG and the USG. In SUSG(H, T_i), there exists a node corresponding to each transaction (except T_i). We also add a read/write node corresponding to every read and write in T_i. We add order-edges between T_i's read/write nodes (see Section 4.2) and start-dependency edges from transaction nodes to other transaction nodes; start-dependency edges from a transaction T_j to T_i in the SSG are now added from T_j to the read/write nodes of T_i in the SUSG. Since we need to know which transactions committed before a read operation of transaction T_i was executed, we specify the time-precedes order for all reads of T_i; the time-precedes order for reads of other transactions is not specified because we are only concerned about guarantees with respect to transaction T_i.

Monotonic reads of snapshots for queries in Read Consistency are specified by a combination of two conditions — G-MSRa and G-MSRb. Both conditions are specified with respect to the reads of a particular transaction T_i; as stated in Section 4.2, we can provide guarantees for version sets of predicate-based writes by treating these operations as predicate-based reads.

> **G-MSRa: Action Interference.** A history H exhibits phenomenon G-MSRa if SUSG(H, T_i) contains a read-dependency edge from a transaction T_j to a read node $r_i(x_j)$ (or $r_i(P: x_j, \ldots)$) without there also being a start-dependency edge from T_j to the same read node.

Disallowing G-MSRa ensures that a read event observes the updates of transactions that have committed before its start point; unlike Snapshot Isolation, this level allows blind writes by concurrent transactions and hence we do not need a constraint with respect to write-dependency edges. The next condition ensures that a read does not miss the effects of all transactions that have committed before the read started:

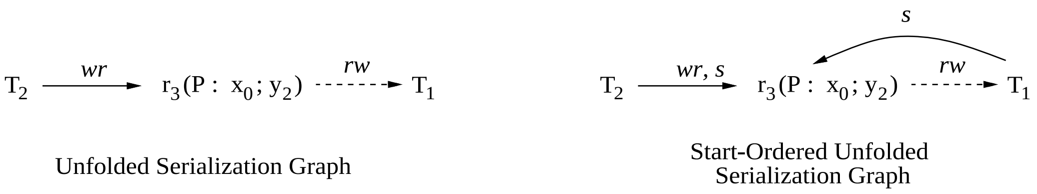
> **G-MSRb: Action Missed Effects.** A history H exhibits phenomenon G-MSRb if SUSG(H, T_i) contains a directed cycle with exactly one anti-dependency edge such that this edge starts from a read node $r_i(x_j)$ (or $r_i(P: x_j, \ldots)$) to a transaction node.

Phenomenon G-MSR is composed of G-MSRa and G-MSRb, and isolation level **PL-MSR** proscribes G1 and G-MSR. This level is stronger than PL-2 and PL-2L. For example, the following history provides PL-2L guarantees but fails to meet G-MSRb:

$H_{non-MSR}$: $w_1(x_1)$ $w_2(y_2)$ c_1 c_2 $r_3(P\!: x_0; y_2)$ c_3

$$[x_0 \ll x_1;\ c_0 \prec_t r_3(P\!: x_0; y_2),\ c_1 \prec_t r_3(P\!: x_0; y_2),\ c_2 \prec_t r_3(P\!: x_0; y_2)]$$

The USG and SUSG for this history is shown in Figure 4-11. In this case, action r_3 reads the updates of T_2 and T_0 but misses the update of T_1 that committed before r_3 had started; objects x_0 and y_2 do not match predicate P. Transaction T_1 changes *x* such that x_1 matches P; this causes T_1 to anti-depend on T_2. Since there is no dependency constraint between T_1 and T_2, PL-2L allows history $H_{non-MSR}$. However, the additional constraint of requiring queries to read database snapshots results in a single anti-dependency cycle in the SUSG; hence, history $H_{non-MSR}$ is disallowed by PL-MSR.



Unfolded Serialization Graph

Start-Ordered Unfolded Serialization Graph

Figure 4-11: History $H_{non-MSR}$ is accepted by isolation level PL-2L but not by PL-MSR.

Isolation level PL-MSR is similar to PL-2L since both levels require that each read in a transaction observes a monotonically increasing view of the database. In Section 4.2.2, we showed that level PL-2L provides each predicate-based read with a consistent view of the database assuming that update transactions maintain consistency (PL-2L places constraints on such reads with respect to dependency conflicts). Level PL-MSR provides stronger guarantees to each predicate-based read: it ensures that a read observe a database snapshot as it existed at some point in logical time. Recall that Snapshot Isolation and PL-2+ also differ from each other in a similar way, i.e., the former provides a database snapshot whereas PL-2+ simply ensures a consistent view based on read and write-dependencies.

Levels PL-MSR and PL-2+ are incomparable even though PL-2+ provides a consistent database to the whole transaction and PL-MSR only guarantees this property for each read. The reason is that PL-2+ does not ensure that each read observes a snapshot of the database whereas PL-MSR does provide this guarantee.

Oracle's Read Consistency provides stronger guarantees to a transaction than provided by our definition of PL-MSR since the Oracle database system executes SQL statements atomically; conditions for providing such guarantees were discussed in Section 3.3.2. To ensure monotonicity of reads with respect to SQL statements (i.e., each SQL statement observes a database state that is at least as recent as the previous one), we can change the SUSG for a transaction T_i to contain action nodes corresponding to T_i's SQL statements rather than its individual reads and writes; we can also define a variant of PL-2L that provides guarantees based on SQL statements rather individual operations.

It seems that the main purpose of Read Consistency is to provide consistent views to each action [Ora95]. PL-2L can achieve the same effect without requiring that each action observe a database snapshot. Thus, PL-2L allows more concurrency than PL-MSR and hence may be preferable to PL-MSR.

4.6 Cursor Stability

Cursor Stability [Dat90] is a well-known consistency guarantee that is supported by many commercial databases. Like some of the other isolation levels, Cursor Stability is also defined in terms of locking in [BBG$^+$95]. We now describe how Cursor Stability can be defined to allow optimistic mechanisms as well; as before, our specifications are presented for committed transactions.

Cursor Stability uses the notion of a *cursor* that refers to a particular (say current) object being accessed by a transaction; there can be multiple cursors in a transaction. When a transaction T_i access an object *x* using a cursor, instead of releasing a read-lock immediately after reading *x* (as in degree 2), T_i retains the lock until the cursor is removed from *x* or T_i commits; if T_i updates the object, the lock is upgraded to a write-lock. This approach prevents the lost update problem, i.e., histories such as $H_{lost-update}$ that was presented in Section 3.2.3:

$H_{lost-update}$: $r_1(x_0, 10)$ $r_2(x_0, 10)$ $w_2(x_2, 15)$ c_2 $w_1(x_1, 14)$ c_1 $[x_0 \ll x_2 \ll x_1]$

In history $H_{lost-update}$, both T_1 and T_2 increment *x* based on *x*'s old value and hence T_2's increment is “lost”. Cursor Stability prevents this scenario by ensuring that T_2 is not allowed to modify *x* while T_1 is uncommitted (if T_1 accesses *x* using a cursor and does not remove the cursor from *x* till it has modified *x*).

Since Cursor Stability is defined in terms of particular objects, we need to annotate the edges in a DSG with object names; this graph is called a *Labeled Direct Serialization Graph* or *LDSG*. Phenomenon G-cursor(*x*) is defined as:

> **G-cursor(x): Labeled Single Anti-dependency Cycles:** A history H exhibits phenomenon G-cursor(*x*) if LDSG(H) contains a cycle with an anti-dependency and one or more write-dependency edges such that all edges are labeled *x*.

Cursor Stability or **PL-CS** is defined to be the level that disallows G1 and G-cursor. PL-CS allows two or more transactions in optimistic schemes to overwrite the same object based on old values as long as only one of the transactions commits, e.g., history $H_{lost-update}$ exhibits phenomenon G-cursor(*x*) and is disallowed (Figure 4-12 shows the LDSG for this history.)

4.7 Update Serializability

Read-only transactions are common in transaction-processing workloads and improving their performance can significantly improve overall system performance. It may be relatively more expensive

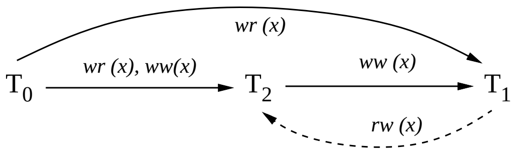


Figure 4-12: Labeled Direct Serialization Graph for history $H_{lost-update}$

to provide serializability than to provide lower isolation guarantees for read-only transactions. Furthermore, read-only transactions may not need strong guarantees such as serializability. For example, if a read-only transaction just needs to observe a consistent state of the database, PL-2+ will be sufficient. In this section, we present an isolation level, PL-3U, for read-only transactions that is stronger than PL-2+ but weaker than serializability. This consistency guarantee has been presented earlier in the literature [GW82, HP86]; we present it using our terminology and describe an optimistic scheme in Section 5.4 that provides this isolation level.

Isolation level PL-3U ensures that a transaction is serializable with all the committed update transactions. The following condition is sufficient to ensure PL-3U for a transaction T_i (T_i could be a read-only transaction):

> **No-update-conflict-misses:** If T_i depends on T_j, it must not miss the effects of T_j and all update transactions that T_j depends or anti-depends on.

Note that the no-update-conflict-misses condition imposes stronger constraints than the no-depend-misses condition since it takes anti-dependencies into account as well. We express the no-update-conflict-misses condition using a DSG that contains all update transactions and transaction T_i (the transaction of interest):

> **G-update: Single Anti-Dependency Cycles with Update Transactions.** A history H and transaction T_i show phenomenon G-update if a DSG containing all update transactions of H and transaction T_i contains a cycle with 1 or more anti-dependency edges.

A transaction T_i is provided **PL-3U** if phenomena G1 and G-update are disallowed; we can prove the equivalence of G-update and the no-update-conflict-misses using an argument similar to the one presented for Theorem 2+. This isolation level is also called *update serializability* (the work in [HP86] defines this condition based on view serializability). Isolation level PL-3U is weaker than PL-3 since read-only transactions are not considered in phenomenon G-update. In PL-3U, two transactions T_i and T_j may observe a serializable database state but unlike PL-3, the serial ordering observed by both transactions could be different; an example that differentiates PL-3U from PL-3 is presented in Section 4.7.2. As we will see in Chapter 7, providing PL-3U to read-only transactions can be less expensive than providing PL-3 in a distributed client-server system.

4.7.1 Differentiating Between Levels PL-2+ and PL-3U

PL-2+ ensures that a transaction observes a consistent state of the database and PL-3U states that the transaction observes a serializable state of the database. It may seem that PL-3U does not add much value over PL-2+ for read-only transactions. Here is an example that shows the difference between the two consistency guarantees and motivates the need for PL-3U.

Consider a stock reporting system. We know that stock prices will not change after the stock market has closed. We assume that there is an object called *Market-status* that indicates whether the market is open or not. Suppose there are companies *X* and *Y* whose stock prices are always the same and suppose transactions T_1, T_2, T_3, and T_q are executed in the following way:

H_{non-3U}: $r_1(M_0, Open)$ $w_1(X_1, 50)$ $w_1(Y_1, 50)$ c_1 $r_2(M_0, Open)$ $w_2(X_2, 55)$ $w_2(Y_2, 55)$ c_2
$w_3(M_3, Close)$ c_3 $r_q(M_3, Close)$ $r_q(X_1, 50)$ $r_q(Y_1, 50)$ c_q
$[M_0 \ll M_3, X_1 \ll X_2, Y_1 \ll Y_2]$

Transactions T_1 and T_2 check that the market is open and update the stock prices to be $50 and $55 respectively and transaction T_3 closes the market.

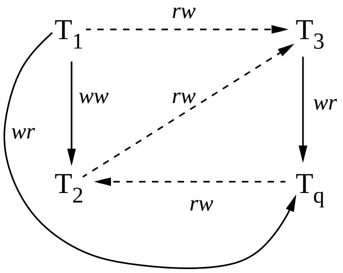


Figure 4-13: DSG of a history that executes at PL-2+ but not at PL-3U (T_0 is not shown).

Suppose that a read-only transaction T_q reads the stock market status and the stock prices of *X* and *Y*. PL-2+ guarantees that T_q will see the same prices for *X* and *Y*, e.g., the two observed values will be $50 or both will be $55. However, with PL-2+, it is possible that T_q observes the stock market to be closed and reads the stock prices to be $50, i.e., it observes T_3's updates but misses the effects of transaction T_2 that T_3 anti-depends on; this is what happens in history H_{non-3U}. In general, if the observed order of conflicting transactions is important, PL-2+ may not be sufficient and PL-3U may be needed. With PL-3U, if T_q finds out the market is closed, it is guaranteed to observe the stock price to be $55. Figure 4-13 gives the DSG of H_{non-3U}. Level PL-2+ allows this history since the only cycle in the graph (among T_2, T_3 and T_q) involves at least two anti-dependency edges.

4.7.2 Differentiating Between Levels PL-3U and PL-3

The reader might wonder how the introduction of another read-only transaction in the direct serialization graph results in a cycle. Here is a variation on the above stock-market example that shows a history that executes at PL-3U but not at PL-3. We assume that transaction T_1 has executed; transactions T_2 and T_3 are not included in this example. Suppose that the following transactions are executed (we are no longer maintaining the invariant that the stock prices of *X* and *Y* are the same):

$$\begin{array}{ll} H_{3U}\text{:} & r_4(M_0, \text{Open})\ \ w_4(X_4, 70)\ \ c_4\ \ r_5(M_0, \text{Open})\ \ w_5(Y_5, 75)\ \ c_5 \\ & r_a(X_4, 70)\ \ r_a(Y_1, 50)\ \ c_a\ \ r_b(X_1, 50)\ \ r_b(Y_5, 75)\ \ c_b \qquad\qquad [X_1 \ll X_4, Y_1 \ll Y_5] \end{array}$$

Transactions T_4 and T_5 check that the market is open and update the stock prices of X and Y to be \$70 and \$75 respectively (they do not conflict with each other). Transaction T_a observes the updates of T_1 and T_4 but misses T_5's effects. Transaction T_b reads the updates of T_1 and T_5 but misses T_4's effects. Thus, each read-only transaction "forces" transactions T_4 and T_5 to be serialized in the opposite order to what the other transaction requires: transaction T_a forces the serialization order $\langle$ T_1, T_4, T_a, T_5 $\rangle$ whereas T_b forces an order where T_5 must be serialized before T_4, e.g., $\langle$ T_1, T_5, T_b, T_4 $\rangle$. The DSG of history H_{3U} is shown in Figure 4-14 (for simplicity, we do not show T_1 in the graph or the history).

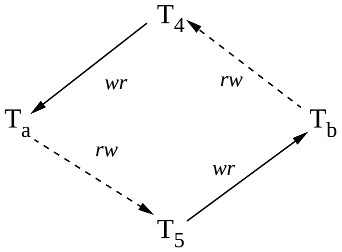


Figure 4-14: DSG of history H_{3U} that executes at PL-3U but not at PL-3.

The DSG contains a cycle but if we remove either T_a or T_b from the graph, the cycle no longer exists. Thus, each read-only transaction is at level PL-3U but the whole system is not serializable. If clients/entities that execute T_a and T_b communicate with each other, they may be confused about the relative order in which the stock prices went up. Thus, this departure from serializability matters only when clients of read-only transactions communicate with each other directly [GW82]; otherwise, PL-3U is as good as serializability.

4.8 Intermediate Degrees for Running Transactions

All isolation levels presented in this chapter can be extended to specify consistency guarantees for running transactions. For each isolation level, we modify the relevant graph to include the executing transaction, T_i, under consideration. Then we define phenomena on the new graph that are analogous to the ones disallowed at the corresponding level for committed transactions.

For EPL-2+, we use the Direct Transaction Graph or DTG presented in the last chapter to define a phenomenon that is analogous to G-single. Level **EPL-2+** disallows P1 and E-single:

> **E-single: Single Anti-dependency Cycles at Runtime.** A history H and an executing transaction T_i exhibit phenomenon E-single if DTG(H, T_i) contains a directed cycle involving T_i with exactly one anti-dependency edge.

To define EPL-2L, we use a modified form of the USG called the *Unfolded Transaction Graph* or *UTG* for a history H and an executing transaction T_i. The graph UTG is the same as USG except that the UTG only contains read nodes (and no write nodes) due to T_i; since T_i's predicate-based writes are considered as predicate-based reads, nodes and edges corresponding to such "reads" are also added. A phenomenon, E-monotonic, that is analogous to G-monotonic is defined and transaction T_i is provided level **EPL-2L** guarantees if phenomena P1 and E-monotonic do not occur:

> **E-monotonic: Monotonic Reads at Runtime.** A history H and an executing transaction T_i exhibit phenomenon E-monotonic if there is a cycle in USG(H, T_i) containing exactly one anti-dependency edge from a read node $r_i(x_j)$ (or r_i(P: x_j, ...)) to some transaction node T_k (*and* any number of order or dependency edges).

In a similar manner, other levels such EPL-SI, EPL-FCV, EPL-CS, EPL-MSR and EPL-3U can be defined by extending the relevant graph and phenomena conditions. The definitions for these levels are given in the Appendix A.

4.9 Summary

In this chapter, we presented specifications for a variety of isolation levels that lie between PL-2 and PL-3; we presented our specifications for committed transactions and then extended them for running transactions.

We used the framework presented in the previous chapter to specify two new isolation levels, PL-2+ and PL-2L. Isolation level PL-2+ ensures that a transaction commits successfully only if it has observed a consistent state of the database. We showed that it is the weakest level that ensures consistent reads. Level PL-2+ is especially useful for read-only transactions in certain applications where observing a consistent database state is sufficient, e.g., if a transaction reads the savings and checking account values of a bank customer and commits, PL-2+ will ensure that the transaction observes consistent balances in the two accounts. PL-2+ disallows all phenomena presented in [BBG$^+$95] that Snapshot Isolation was designed to prevent, i.e., inconsistent reads, read phantoms, and lost updates. Since PL-2+ is weaker than Snapshot Isolation, it has the potential of being implemented more efficiently, especially in a distributed system with multiple servers. Thus, we believe that PL-2+ is an isolation level that database vendors may be interested in providing.

Our second new level, PL-2L, was inspired by a lock-based implementation of degree 2 and allows a transaction to commit only if it has observed a monotonically increasing prefix of the database as it executed. PL-2L has been designed to support legacy applications when the underlying

Level	Name	Phenomena disallowed
PL-CS	Cursor Stability	G1, G-cursor
PL-2L	Monotonic View	G1, G-monotonic
PL-MSR	Monotonic Snapshot Reads	G1, G-MSR
PL-2+	Consistent View	G1, G-single
PL-FCV	Forward Consistent View	G1, G-SIb
PL-SI	Snapshot Isolation	G1, G-SI
PL-2.99	Repeatable Reads	G1, G2-item
PL-3U	Update Serializability	G1, G-update
PL-3	Full Serializability	G1, G2

Figure 4-15: Isolation levels stronger than PL-2

concurrency control mechanism is changed to (say) optimism. However, it is a useful level in its own right, e.g., many browsing applications may find its guarantees more desirable than just PL-2. In fact, Oracle provides a level called Read Consistency that is similar to PL-2L.

This chapter has also shown that a graph-based technique can be used for defining a variety of isolation levels. We have redefined existing commercial levels such Snapshot Isolation, Read Consistency, and Cursor Stability in an implementation-independent using variations of direct serialization graphs. For example, to specify Snapshot Isolation, we add new edges in the graph that capture the notion of logical time, for Cursor Stability, we labeled the edges with objects names, and to capture the essence of Read Consistency, we added nodes corresponding to individual events in a transaction.

Various levels can be ranked according to their "strength": one level is stronger than another if it allows fewer histories. Figure 4-16 shows how various levels are related to each other: if level *Y* is stronger than level *X*, there is a directed path from *X* to *Y*; if there is no path between two levels, they are unrelated to each other. This figure is the same as Figure 4-1; we have simply repeated it here for convenience.

Read Consistency and Snapshot Isolation are unrelated to some of the levels on the right side of the figure because of their snapshot read requirement for queries and transactions respectively; this is also the reason why PL-2+ and PL-2.99 are not strictly stronger than Read Consistency. Snapshot Isolation is weaker than strict serializability. Strict serializability [Pap79] ensures that all transactions can be serialized in an order that also respects the real-time ordering of non-overlapping transactions. A lock-based system (i.e., long read-locks and write-locks) provides strict serializability guarantees to transactions but a multi-version timestamp scheme may not.

We can easily define strict serializability in our framework. Consider a graph called the *Real-time Serialization Graph* or *RSG* which is the same as the DSG except that there are some extra edges to capture the order of non-overlapping transactions: If a transaction T_i commits before T_j

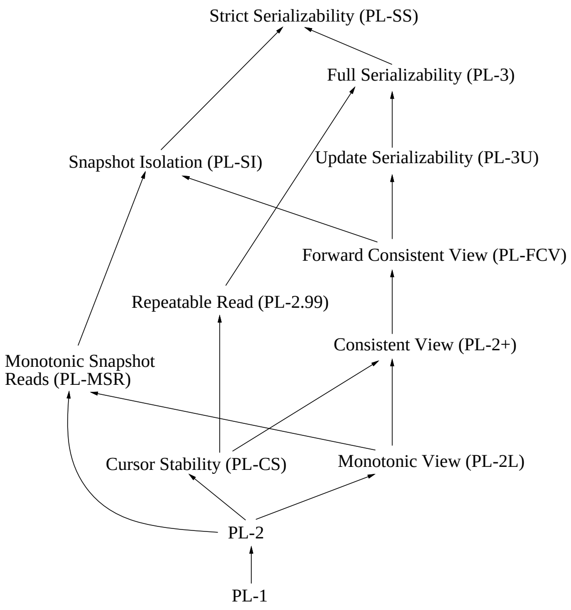

Strict Serializability (PL-SS)
Full Serializability (PL-3)
Snapshot Isolation (PL-SI)
Update Serializability (PL-3U)
Forward Consistent View (PL-FCV)
Repeatable Read (PL-2.99)
Consistent View (PL-2+)
Monotonic Snapshot
Reads (PL-MSR)
Cursor Stability (PL-CS)
Monotonic View (PL-2L)
PL-2
PL-1

Figure 4-16: A partial order to relate various isolation levels for committed transactions.

executes its first event in real-time, we add a *real-order* edge from T$_i$ to T$_j$; these extra edges force non-overlapping transactions to be serialized in the order in which they executed in real-time. Strict serializability or PL-SS is defined as the level that disallows G1 and G2 when these phenomena are defined on the RSG instead of the DSG.

Chapter 5

Optimistic Implementations for Client-Server Systems

This chapter presents optimistic mechanisms that provide different degrees of isolation to running and committed (update and read-only) transactions in a distributed client-server system. Since the consistency guarantees provided at a particular level for running and committed transactions are similar, the implementation for both cases is also similar. Our schemes have been designed for a system where database objects may be distributed over multiple servers and clients may cache some of the objects on their machines for better performance. In our implementations, a transaction can either read its own modifications or the updates of other committed transactions (dirty reads are not allowed).

To provide serializability for committed transactions in these systems, we use an optimistic scheme called CLOCC [Ady94, AGLM95, Gru97]. This scheme has been shown to outperform the best-known locking implementation, ACBL [CFZ94, ZCF97], in a client-server system [Gru97] on many workloads. Our optimistic mechanism for providing PL-2 is based on CLOCC and shares many of its advantages; we call this scheme *Weak*-CLOCC.

For levels such as PL-2+, PL-2L, PL-3U, and their corresponding execution-time levels, we have designed a mechanism that captures conflict relationships efficiently using multipart timestamps or *multistamps*. Multistamps contain timestamp entries for every server or client-server pair in the system. However, in earlier systems [BSS91, PST^{+}97], multistamps have not scaled well with a large number of clients and servers. We have designed a simple and novel technique called *multistamp truncation* that keeps multistamps small. The truncation technique takes advantage of the fact that our multistamps contain real time values and makes time-based judgements to approximate old timestamp entries.

Our mechanisms for lower isolation levels offer a number of performance advantages over CLOCC. These schemes (except PL-3U) do not require clients to send information about the objects read by their transactions to the servers; this results in lower network bandwidth requirements. They also provide additional benefits for read-only transactions compared to CLOCC. First, read-only transactions do not interfere with update transactions, i.e., they do not cause update transactions to abort. Second, transaction latency is decreased: when a read-only transaction finishes, it can be committed without communicating with the servers (CLOCC requires such communication); although a small message may be sometimes required for PL-2+ and PL-3U, our results in Chapter 7 show that such messages are rare. Finally, the system is more scalable since these mechanisms reduce the utilization of resources such as the network (fewer and smaller messages) and the server CPU (most of the work for committing read-only transactions is offloaded to clients and the server validates only update transactions with other update transactions). As in CLOCC, we do not require a transaction to declare whether it is read-only or not when it starts; we detect this property when a transaction ends.

The rest of this chapter is organized as follows. Section 5.1 describes CLOCC and gives a brief overview of the Thor distributed database system since our work has been done in the context of Thor. Section 5.2 presents our modifications to CLOCC for providing PL-1 and PL-2. Section 5.3 presents our multistamp-based technique for guaranteeing PL-2+ to committed transactions; we also show how this scheme can be modified to provide EPL-2+ for running transactions. In Section 5.4, we show how the multistamp-based scheme can be extended for committing read-only transactions efficiently at level PL-3U; our technique ensures that the read-only participant optimization [GR93] for two-phase commit is not sacrificed. Finally, Section 5.5 discusses related work. Implementations for PL-2L and EPL-2L along with a technique for efficiently committing read-only transactions at PL-3 are discussed in Appendix B; this appendix also presents techniques for providing different levels of causality to clients.

5.1 Database Environment and CLOCC

Our work has been done in the context of the Thor database system; detailed information about Thor can be found in [Ghe95, LAC$^+$96, CALM97]. We assume that application computations occur within transactions so that persistent objects can be maintained consistently despite concurrent accesses from multiple applications and possible failures. Each application session runs a single transaction at a time. It specifies when to commit the current transaction: the transaction includes all objects accessed by the application since the last commit point.

Persistent database objects are stored at servers on server disks; each object resides in a single page and objects are typically much smaller than pages. There may be many servers; the server where an object resides is called its *owner*. The owner can be ascertained from the object identifier (oid) of an object. Servers are replicated for high availability in Thor.

To improve performance, the application code is executed on the client machine using locally cached copies of objects. Each client has an object cache that is maintained using techniques presented in [CALM97]. There could be multiple clients on a single machine; each client would have its own cache of objects. Transactions run entirely at clients; clients communicate with servers only when there is a miss in the cache, and to commit transactions. Clients and servers use an ordered delivery protocol such as TCP for communication.

The server maintains information about data being cached by each client. For each client C, it maintains a *cached set* that keeps track of objects being cached at C (as discussed later, these sets are maintained at a coarse-granularity to reduce space overheads). Cached sets are used for performing validation checks in CLOCC. For recovery purposes, a server keeps track of the clients that are connected to it on stable storage; if the server crashes and recovers, it contacts the clients to determine the pages cached by those clients.

The code that manages the cache on the client machine is a part of Thor and will be referred to as the *client*. Each server has a cache of objects in main memory, which it uses to satisfy fetch requests from clients. The organization of the system is shown in Figure 5-1.

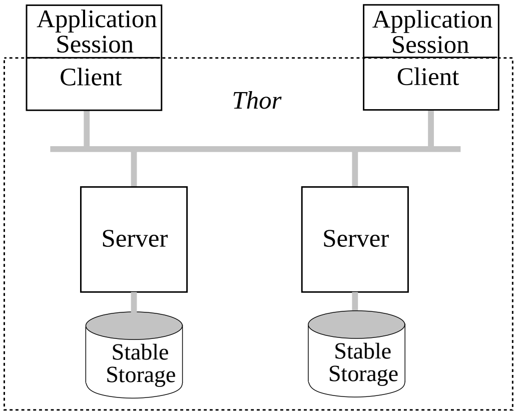


Figure 5-1: The client-server model of Thor.

5.1.1 Serializability for Committed Transactions: CLOCC

We now present CLOCC or *Clock-based Lazy Optimistic Concurrency Control* scheme; for details on CLOCC, see [Ady94, AGLM95, Gru97] (CLOCC was earlier referred to as AOCC). CLOCC has been designed to perform well for cacheable transactions, i.e., for workloads in which a client can cache all objects read/written by a transaction during its execution. Gruber [Gru97] has shown that CLOCC outperforms the best known locking scheme, ACBL [CFZ94, ZCF97], for client-server systems, across a wide range of workloads. Low communication requirements, fewer delays, and low cost of aborts are some of the reasons why CLOCC has a higher throughput and scales better than ACBL. CLOCC has been designed to work well for environments where all operations are executed by clients. In environments where servers may perform part of the work, another scheme called AACC (Asynchronous Avoidance-based Cache Consistency) has been shown to outperform CLOCC [OVU98]. Both CLOCC and ACBL are not expected to perform well in workloads where there are hotspots, i.e., high contention on a single or very few objects. For such cases, mechanisms such as field calls [Reu82] and escrow reads [O'N86] are known to offer superior performance.

In a distributed system, transactions that have accessed objects at multiple servers must be serialized in the same order at all servers. In CLOCC, transactions are serialized in timestamp order where timestamps are taken from real clocks. When a client wants to commit transaction T_i, it assigns a timestamp T_i.ts that contains the client's local clock value augmented with the client's identity to make it globally unique. We assume that clocks are loosely synchronized, i.e., clocks at different nodes in the network may differ by at most a small skew (say, a few tens of milliseconds). The presence of such clocks is a reasonable assumption for current systems; protocols such as the Network Time Protocol [Mil92, Mil96] provide such a facility. Mills [Mil96] has shown that NTP provides synchronization within a few milliseconds even across wide area networks. We assume that server clocks never run backwards, and advance rapidly enough that each transaction can be assigned a distinct timestamp; these assumptions are easy to guarantee (see [Lis93]). These clocks simplify our algorithm and, since their values are close to real time, they allow us to make time-dependent design decisions, and to reason about the performance of our scheme. In CLOCC, loose synchronization is needed only for performance reasons and not for correctness.

Commit and Coherence Protocol

To enable serializability checks at the end of a transaction, the client keeps track of objects read and written by its current transaction T_i. At commit time, along with T_i's timestamp (T_i.ts), the client sends the identity of all objects read and written by T_i (i.e., T_i.ReadSet and T_i.WriteSet) to the servers that own these objects; it also sends copies of objects modified by T_i.

If objects from only one server are used by the committing transaction, this server can commit the transaction unilaterally if the serializability checks succeed. Otherwise, the client chooses one of the servers as the *coordinator* of a commit protocol with the other owners, called the *participants*. We use a standard 2-phase protocol [GR93]. We describe the protocol briefly here to provide the context for our scheme.

In phase 1, the client sends information about the objects accessed by its transaction to all the participants. Each participant (the coordinator is also a participant) *validates* the committing transaction T_i; we describe how validation works later in this section. If validation succeeds, the participant logs the installation information on stable storage and sends a positive response to the coordinator; otherwise, it rejects the transaction. If all participants respond positively, the coordinator commits transaction T_i by logging a commit record on stable storage; otherwise, the transaction is aborted. In any case, the coordinator notifies the client of its decision. Phase 1 includes two log updates to stable storage, but the optimizations suggested by Stamos [Sta89] can reduce this to a single log update.

In phase 2, the coordinator sends *commit* messages to the participants. On receiving a commit message, a participant *installs* new versions of the objects that were modified by T_i (so that future fetches see the updates), logs a commit record on stable storage, and sends an acknowledgement to the coordinator. When the coordinator receives acknowledgements from all participants, the protocol is complete. This phase is not executed for transactions involving a single server; objects are installed by this server after the transaction succeeds validation. Furthermore, the delay observed by the client before it can start the next transaction is due to phase 1 only; phase 2 happens in the background.

If T_i has not modified any object at some participant, that server is called a *read-only participant* (other participants are called *read-write participants*); no installation information needs to be logged at this participant. Furthermore, a read-only participant does not need a commit message from the coordinator during phase 2; this optimization is also referred to as the *read-only participant optimization* for two-phase commit [GR93].

Some concurrency control schemes (e.g., callback locking [ZCF97] and the multi-version scheme in [ABGS87]) guarantee that all read-only transactions are serializable; thus, committing such transactions does not require any communication with the servers. However, our scheme requires validation to be performed for read-only transactions; therefore, phase 1 messages must be sent to all participants for such transactions also. But, in our case, the coordinator of a read-only transaction does not need to be reliable, and does not need to use stable storage. Therefore, the client can act as the coordinator: it sends the prepare messages to the participants and collects their responses; for details on this mechanism, see [AGLM95]. This saves a message roundtrip and reduces the latency for committing read-only transactions to a single message roundtrip; furthermore, there are no updates to stable storage. (In Appendix B, we discuss a technique that avoids this roundtrip as well.)

After a client C's modification transaction T_i has committed at a server, the server examines the cached set to determine clients other than C that may be caching objects modified by T_i. The server maintains an *invalid set* for each client D and adds the list of obsolete objects (because of T_i's modifications) to D's invalid set. It informs clients about these old objects by sending *invalidation messages* to them. Note that client C is not waiting while invalidation messages are being sent since a server sends them after phase 2 has started. When a client D receives an invalidation message that contains object *x*, it removes *x* from its cache. If D's current transaction has accessed object *x*, the transaction is aborted. When a transaction aborts at the client, any objects updated by the transaction are also removed from the cache. To avoid refetching these modified objects after an abort, the client maintains an *undo log*; before any object is updated, the client makes a copy of the object. Thus, when the transaction aborts, the updated objects are simply reverted to their original state unless they have been invalidated; this simple optimization is very important in enhancing CLOCC's performance.

The two-phase commit protocol for Thor is shown in Figure 5-2. Numbers indicate the order of messages, i.e., message *i* precedes message *i+1*; messages with the same numbers can be sent in parallel. A log force to stable storage that is shown as *i/j* indicates that it is done after receiving message *i* but before sending message *j*.

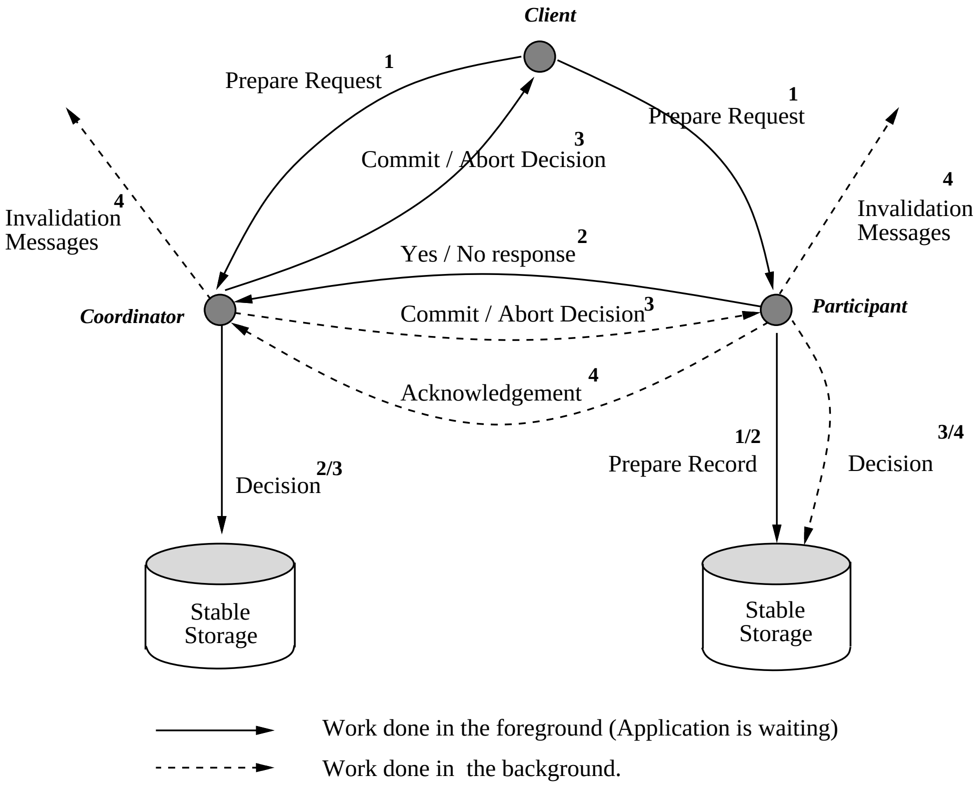


Figure 5-2: The two-phase commit protocol in Thor.

Clients send acknowledgements after processing invalidation messages; when the server receives the acknowledgement, it removes the information about the invalidated objects from that client's invalid set. Invalid sets, and hence invalidation messages, are small (just a few entries) because of these acknowledgements (see [AGLM95, Gru97] for experimental results).

Both invalidation messages and acknowledgements are *piggy-backed* on other messages being exchanged between the client and server; the term "lazy" in CLOCC comes from the fact that this scheme avoids sending synchronous messages as much as possible. There is always a certain amount of such traffic: in addition to fetch and commit requests and replies, clients and servers exchange "I'm alive" messages for failure detection purposes; these messages are sent if there has been no communication between a server and a client for a certain *timeout* period, e.g., half a second in a LAN environment. Therefore, our scheme does not cause extra message traffic, although messages may be bigger; however, as shown in [AGLM95], the increase in messages size due to invalidations is small.

Invalidation messages are not required for correctness but they have the following desirable effects:

1. If a client receives an invalidation message for an object *x* that has *not* been read or written by the current transaction, it simply evicts *x*, thus avoiding a potential abort that would otherwise result from reading *x* later.
2. If the current transaction has already read or written *x*, the client aborts the transaction immediately, thus limiting wasted work (since the transaction would abort later anyway).
3. Invalidation messages also help in ensuring that the effective client cache is not lowered because of obsolete objects. If these messages were not sent, a large number of objects in the client cache can become stale, resulting in low utilization of the cache since old objects are effectively useless (and even harmful because a transaction that reads obsolete objects is forced to abort later).
4. Some of the checks that would have been performed at the server are now performed at the client, i.e., work is offloaded from servers to clients making the scheme more scalable. A client checks its read set against the list of old objects received in the invalidation messages; without these messages, the server would performed these checks at commit time.

Although cached sets are maintained for each client, their space overhead is not large because the server stores them at the page level: a cached set for client C contains the identifiers of the pages containing objects cached at C rather than individual object identifiers. The client cache may not contain all objects in a page and therefore some unnecessary invalidation messages may be sent (piggy-backed on other messages). However, these messages do not result in spurious aborts at the client since an invalidation message causes an abort only if the corresponding object is being used at the client. At the cost of larger invalidation messages, the size of these cached sets can be substantially reduced by further approximating this information using Bloom filters [FCAB98].

Validation

The purpose of validation is to prevent the commit of any transaction that violates PL-3 requirements. Our scheme uses *backward validation* [Hae84] to provide serializability: a validating transaction T_i is checked against all transactions that have already validated successfully. Our validation algorithm is embedded in the commit protocol described above. The presentation assumes that servers do not fail; for a discussion on how we deal with failures, see [AGLM95]. We define the following relationship for conflicts: two transactions *conflict* if one has modified an object that the other has read or modified.

When a participant receives the validation information of a committing transaction T_i (T_i.ts, T_i.ReadSet and T_i.WriteSet), it checks T_i's read set and write set against the information of earlier committed and prepared transactions. Validation information of all successfully validated transactions is maintained in a data structure called the *validation queue*, or VQ; thus, the VQ holds the read and write sets of transactions that have prepared or committed previously at the server. If T_i cannot be serialized in timestamp order due to some transaction T_j in the VQ, the participant aborts T_i by sending a negative acknowledgement to the coordinator. (Even if T_j is only prepared, the server cannot abort T_j since the 2-phase commit protocol disallows a participant from unilaterally aborting a prepared transaction.)

To simplify our algorithm, we arrange the read set to always contain the write set (no blind writes), i.e., if a transaction modifies an object but does not read it, the client enters the object in the read set anyway. This implies that some transactions might exhibit spurious read-write conflicts, but aborts due to such spurious conflicts are rare because it is unlikely that a transaction writes an object without reading it.

The rules for validation differ according to the timestamp order between an already validated transaction T_j and the incoming transaction T_i. We now discuss these checks.

Suppose T_j has a timestamp later than T_i. Since T_j is prepared/committed, it could not have observed T_i's updates (there are no dirty reads in CLOCC). Since T_j is later in the schedule than T_i, T_i should not modify any object that T_j has read. If T_i has read an object that T_j has modified, T_i can still be serialized. However, as discussed in [AGLM95], we abort T_i in this case also to provide external consistency (so that transaction commit order as observed by clients is the same as the real time order). Thus, the validation check against a later transaction T_j is the following: if T_i conflicts with T_j, T_i is aborted.

Now we consider the checks performed by each participant to validate T_i against a prepared/committed transaction T_j that has an earlier timestamp than T_i. If T_j has read an object x and T_i has modified x, T_i can be serialized after T_j. However, if T_i has read x and T_j is in prepared state and has modified x, T_i must be aborted because T_i could not have read that version (dirty reads are not allowed). If T_j is a committed transaction, we must ensure that T_i has read the latest version of x; this is called the *version check*.

The version check can be performed by associating a version number (such as the timestamp of the installing transaction) with each object. The version number of an object read by T_i can then be checked against that installed by T_j. However, storing a version number per object consumes disk storage as well as space in the server cache. More importantly, if the objects are not in the server memory at validation time, the server may have to perform read the pages from disk to obtain the version numbers.

Instead, we perform the version check without using version numbers. Recall that the server maintains an invalid set for each client; this set keeps track of those objects in the cached set that have been invalidated but whose invalidations have not been acknowledged by the client. To check whether a validating transaction T_i has read the latest version of an object, the participant checks the invalid set of T_i's client and rejects T_i if it has used an object in that invalid set.

The memory requirements for validation are low enough that the server can keep the needed information in primary memory. The data structures used for validation are cached sets, invalid sets, and the VQ. We have already argued that the cached sets and invalid sets are small. The VQ is also kept small by removing old validation records in a timely manner. We take advantage of loosely synchronized clocks to determine which entries are old and remove them; the removed entries are summarized using a *watermark* timestamp, which is greater than or equal to all the timestamps of the removed entries. Since a server no longer has fine-grained information about transactions below the watermark, it must be conservative and abort an incoming transaction T_i if T_i's timestamp is earlier than the watermark (or ask T_i's coordinator to retry after assigning a higher value to T_i's timestamp); this check is called the *watermark check*. Our notion of approximating a transaction's information is similar to the Commit_LSN technique [Moh90] that uses a watermark called Commit_LSN to determine whether a piece of data is committed or not. Their technique approximates information about objects on a page using the Commit_LSN whereas we use a watermark to remove validation information and summarize the removed entries.

Setting the watermark to a high value (i.e., later timestamp) has the advantage that the VQ contains few entries but has the disadvantage that transactions may be rejected unnecessarily; setting it far back in time reduces the probability of unnecessary aborts but increases the size of the VQ.

We set the watermark by taking network delays into account. When a transaction T_i commits, the client uses its local clock and assigns a timestamp T_i.ts to the transaction. The client's prepare request reaches each participant when the latter's clock time is $T_i.ts + \delta + \epsilon$ where δ is the network delay and ϵ is the approximate clock skew. Thus, to ensure that T_i is not rejected due to the watermark check, a participant must not have not removed any VQ entry with a timestamp less than T_i.ts, i.e., a server keeps the watermark at least $\delta + \epsilon$ below its current time. Details about the watermark mechanism are discussed in [AGLM95].

We have now described all of the validation checks performed at each participant to validate a given transaction T_i; Figure 5-3 summarizes these checks.

5.2 Mechanisms for Isolation Levels PL-1 and PL-2

We now modify CLOCC to provide PL-1 guarantees for committed transactions; in this scheme, servers perform checks to ensure that G0 is disallowed. Since dirty reads are not permitted in our environment, this scheme also ensures that G1 cannot occur. Hence, our PL-2 scheme is identical to our PL-1 implementation.

The checks for the PL-2 scheme are similar to the ones in CLOCC except that we do not have to validate the reads of a committing transaction; we compare the write set of a PL-2 transaction with the write sets of other transactions (the checks are changed to only consider write sets). Invalidation messages are handled as in CLOCC except that a client aborts its current transaction T_i only if T_i has

```
Watermark Check
If $T_i$.ts < Watermark then send abort reply to coordinator

Checks Against Earlier Transactions
For each uncommitted transaction $T_j$ in VQ such that $T_j$.ts < $T_i$.ts
% Committed transactions are handled by the version check
    If ($T_j$.WriteSet $\cap$ $T_i$.ReadSet $\neq \phi$) then send abort reply to  coordinator

Version Check
% $T_i$ ran at client C
For each object x in $T_i$.ReadSet
    If x $\in$ C's invalid set then send abort reply to coordinator

Checks Against Later Transactions
For each transaction $T_j$ in VQ such that $T_i$.ts < $T_j$.ts
    If  ($T_i$.ReadSet $\cap$ $T_j$.WriteSet $\neq \phi$) or ($T_i$.WriteSet $\cap$ $T_j$.ReadSet $\neq \phi$) then
        Send abort reply to coordinator
```

Figure 5-3: Validation checks in CLOCC for transaction T_i

modified an object *x* specified in the invalidation message; if T_i has just read *x*, the client discards *x* from its cache (T_i is not aborted since simply reading from committed transactions ensures G1). We do not store the read set of a PL-2 transaction in the VQ since its reads need not be validated against writes of transactions that commit later.

In a mixed system, some of the transactions may commit at PL-3. To ensure that all histories generated in our system are mixing-correct, we need to ensure that PL-2 transactions handle obligatory conflicts correctly. As discussed in Section 3.3, obligatory conflicts for PL-2 transactions correspond to anti-dependency edges originating from PL-3 nodes, i.e., overwriting of reads by PL-3 transactions needs to considered. Thus, we ensure that if a committing PL-2 transaction T_i overwrites an object that was read by a PL-3 transaction T_j, T_i's timestamp is greater than T_j's timestamp. We call this check as *obligatory check* and our PL-2 scheme is called *Weak*-CLOCC.

The validation checks in Weak-CLOCC are actually identical to the ones given in Figure 5-3 if we simply consider a PL-2 transaction's read set to be null for these checks. A similar change is needed to use CLOCC in a mixed system where both PL-2 and PL-3 transactions can commit: when a PL-3 transaction is validated against a PL-2 transaction T_j, the server assumes that T_j.ReadSet is null; this is acceptable in a mixed system since anti-dependencies for PL-2 transactions are not considered.

Weak-CLOCC and CLOCC ensure that all histories generated in the system are mixing-correct (see Section 3.3.1 for definition of mixing-correct). Since we do not allow dirty reads, phenomena G1a and G1b cannot occur for any transaction/ We can prove that the MSG is acyclic by using the following property: if there is an edge from T_i to T_j in the MSG, T_i's timestamp must be less then T_j's timestamp. Both CLOCC and Weak-CLOCC ensure that this property is valid for transactions that commit at PL-3 and PL-2 levels respectively. Furthermore, the obligatory checks in Weak-CLOCC ensure that this property is valid for anti-dependency edges originating from PL-3 transactions.

Weak-CLOCC has a number of advantages over CLOCC. First, it has lower network bandwidth requirements since clients do not send information about their read sets to servers. Second, servers perform much less validation work since they only check write sets of transactions and not the read sets; write sets are expected to be much smaller than the read sets. Third, when a read-only transaction finishes, it can commit immediately without communicating with the servers. Thus, read-only transactions do not interfere with update transactions and the server does not need to validate them. Finally, read-only transactions are *never* aborted by Weak-CLOCC since an invalidation message can result in an abort only if the client's current transaction has modified that object; also, update transactions abort less frequently because they never conflict with read-only transactions. Of course, these performance benefits over CLOCC come at a cost: Weak-CLOCC offers much weaker guarantees than serializability.

Weak consistency schemes based on locking for providing PL-1 and PL-2 in a client-server system have been presented in [BK96]. However, like locking schemes for serializability, these mechanisms require extra message roundtrips for acquiring write locks; our optimistic schemes have lower communication requirements.

5.3 Multistamp-Based Mechanism for PL-2+ and EPL-2+

We now present our multistamp-based consistency mechanisms provide PL-2+ to committed transactions and EPL-2+ to running transactions. These techniques form the basis for other schemes discussed in later sections. Multistamps lie at the heart of these schemes and are used to place constraints on the reads performed by transactions.

Section 5.3.1 gives an overview of our multistamp-based schemes for PL-2+ and EPL-2+. Sections 5.3.2 and 5.3.3 describe the processing at the server and the client, ignoring size issues: multistamps are allowed to grow without bound and so are local tables at the server. Section 5.3.4 describes how validation is performed for PL-2+ and also gives an informal argument to show that the scheme is correct. Section 5.3.5 shows how the mechanism can be modified for providing EPL-2+ to running transactions. Section 5.3.6 describes how we solve the size problems for multistamps and other data structures maintained at the server; it also discusses how the truncation technique can be applied in other systems. Section 5.3.7 presents an optimization to offload work from servers to clients.

5.3.1 Overview of the PL-2+ and EPL-2+ Implementations

Before describing the details, we give a brief overview of our PL-2+ and EPL-2+ implementations. We first present a scenario in which a transaction observes an inconsistent state of the database; this scenario is allowed by CLOCC for running transactions and by Weak-CLOCC for committed transactions. We then describe how our mechanism prevents this situation.

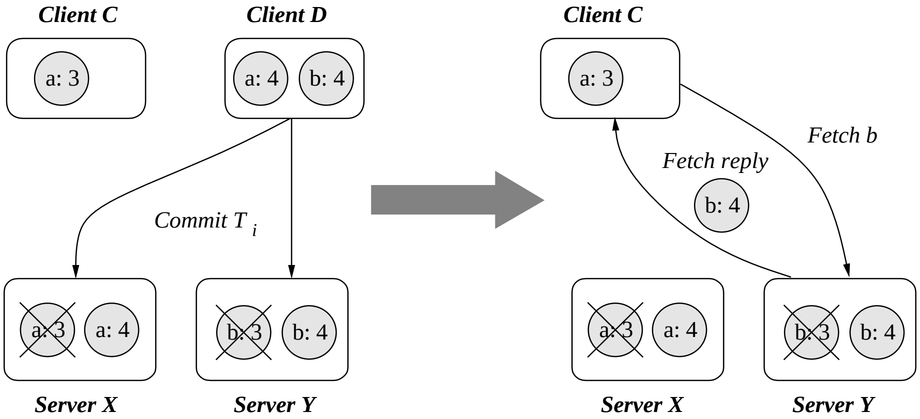


Figure 5-4: Inconsistent database state viewed by client C due to observation of T_i's partial effects.

Suppose that two servers X and Y store objects *a* and *b* respectively and an application maintains the invariant that *a* is equal to *b*. A transaction T_i from client D updates the values of the objects from 3 to 4 and commits (see Figure 5-4). Another client C's transaction T_j reads the old value of *a* and tries to read *b*. Since *b* is missing from its cache, it fetches *b* from server Y and observes an inconsistent state of the database. The Weak-CLOCC scheme allows T_j to commit because PL-2 does not guarantee consistent reads. On the other hand, any implementation of level PL-2+ would abort T_j. Note that CLOCC allows T_j to observe inconsistent values of *a* and *b* but does not permit T_j to successfully commit. However, observing an inconsistent state of the database may cause T_j to behave in an unexpected manner, e.g., the application could crash. To prevent T_j from observing an inconsistent view, it must be provided level EPL-2+ during its execution.

Let us see how EPL-2+ can be provided to client C as it executes. The protocol is shown in Figure 5-5; the numbers indicate the order of messages sent in the system. When transaction T_i commits at servers X and Y, the servers store extra consistency information with objects *a* and *b*. The extra information stored at server Y is a requirement on transactions that read object *b*: if transaction T_j reads the new value of *b*, it must have communicated with object *a*'s server X in the recent past so that T_j does not see an old value of *a*. A similar requirement is placed at server X along with object *a*. When client C fetches object *b*, server Y piggybacks these requirements on the fetch reply. In this scenario, since client C is not sufficiently recent with respect to server X, it sends a message to X requesting for recent consistency information. When it receives a reply, it is also informed that it has read an old value of object *a*. As a result, transaction T_j is aborted before it views an inconsistent state, i.e., T_j executes at level EPL-2+. To provide PL-2+ instead of EPL-2+, client C checks the requirements sent by server Y only at commit time and not immediately after a fetch reply.

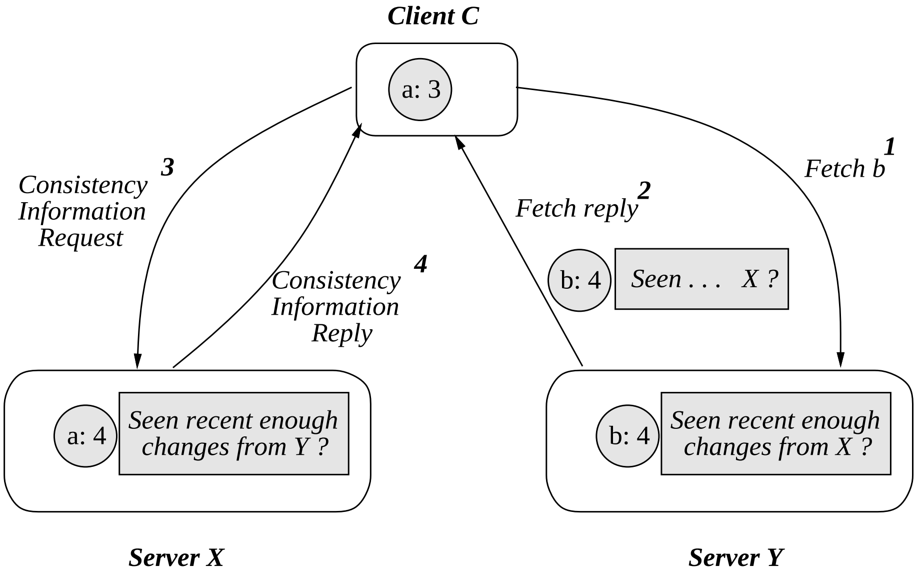

Client C
a: 3
Fetch b
1
Fetch reply
2
b: 4
Seen . . . X ?
Consistency Information Request
3
Consistency Information Reply
4
a: 4
Seen recent enough changes from Y ?
b: 4
Seen recent enough changes from X ?
Server X
Server Y

Figure 5-5: With EPL-2+, client C is prevented from viewing an inconsistent state due to extra consistency information stored with *b*.

The above example describes the essential ingredients of our algorithm: the consistency requirements are generated by servers during the two-phase commit protocol and piggybacked on fetch replies to clients; clients ensure that they are sufficiently recent with respect to the relevant servers.

The requirements on clients can be captured by maintaining lists of transaction ids and propagating them among clients and servers. However, due to the transitive nature of the dependency relationship, these lists can become very large resulting in significant memory, processing and network overheads. Instead, consistency requirements contain information about invalidations in the form of multistamps.

Recall from Chapter 4 that a client C's transaction T_q is provided PL-2+ or EPL-2+ if it does not miss the effects of any transaction that T_q depends on. One of the key insights regarding invalidations in our system is that they can easily used to capture the notion of "not missing a transaction's effects". Suppose a client C receives the invalidations generated by a transaction T_i, *After* this point, if C tries to access an object *x* that was modified by T_i, it will be forced to go to the server and receive the latest version of *x*, i.e., it cannot miss T_i's effects. Thus, to provide PL-2+, we need to ensure that a client C must receive the invalidations of T_i and all transactions that T_i depends on before C observes T_i's updates.

Information about invalidations is maintained in the form of multistamps. Each committed transaction T_i is associated with a multistamp, T_i*.mstamp*, that indicates its invalidations and those of all transactions it depends on. A multistamp is a set of tuples ⟨*client, server, timestamp*⟩; each tuple ⟨*C, X, ts*⟩ says that an invalidation was generated for client C by the prepare of some transaction at server X when the value of X's clock was *ts*. When client C receives a multistamp with a tuple ⟨*C, X, ts*⟩ in its fetch reply, it can continue execution (for EPL-2+) or commit (for PL-2+) only after it has received all invalidations that were generated by server X before X's clock reached time *ts*.

We assume the obvious merge operation on multistamps: if the two input multistamps contain a tuple for the same client/server pair, the merge retains the larger timestamp value for that pair.

We now describe the details of how multistamps are generated and used in our implementation.

5.3.2 Processing at the Server

Servers have two responsibilities: they must compute multistamps and they must send them in fetch responses so that clients can act accordingly. When a server replies to a fetch, it sends the requested page P along with P's multistamp; the multistamp of a page P is the merge of the multistamps of all transactions that have ever modified P.

The server maintains the following data structures. The PSTAMP table maps pages to their multistamps. The invalid set or ILIST maps clients to invalidation information. Each element, ILIST[C], contains a timestamp *ts* and a list of object ids, indicating that these objects were invalidated for client C at time *ts*. The VQ stores multistamps of committed transactions along with other information needed for validation.

Commit Processing. In the prepare phase, the server validates a preparing transaction T_i to ensure that it has not missed any updates, and that its updates do not conflict with updates of other transactions. The latter check is done using the Weak-CLOCC algorithm discussed in Section 5.2 (i.e., assuming that T_i's read set is empty). The former check requires tracking of transaction dependencies and is done using multistamps; we describe how this test is performed in Section 5.3.4.

If the validation of transaction T_i succeeds, participant X computes multistamp T_i*.mstamp* as follows:

1. X initializes T_i*.mstamp* to be empty.

2. If T_i's commit would cause invalidations to be generated for any client, X sets *ts* to be the current time of its clock. For each potentially invalidated client C:

    (a) X adds tuple <*C, X, ts*> to T_i.mstamp.

    (b) X adds <*ts, olist*> to the ILIST for C, where *olist* contains ids of all objects modified by T_i that are located on pages listed in X's directory for C.

3. For each transaction T_j that T_i depends on, X merges VQ[T_j].mstamp with T_i*.mstamp*. The dependencies are determined using X's VQ (we do not merge the multistamps of transactions that T_i anti-depends on since PL-2+ does not capture anti-dependencies).

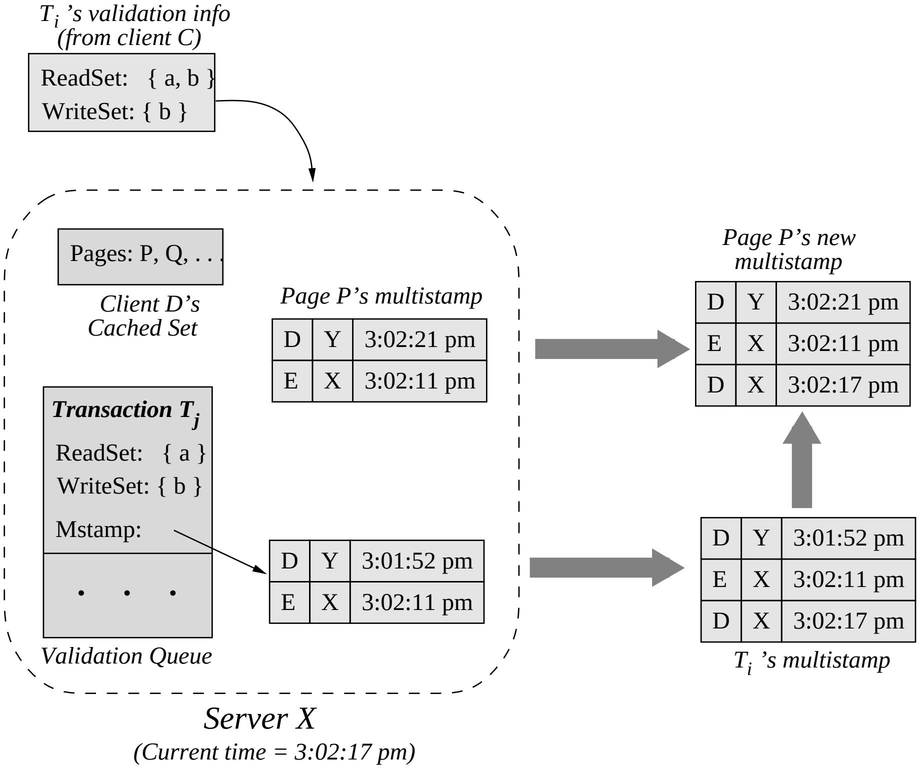


Figure 5-6: Generation of multistamp for a committing transaction T_i and page P.

Then X sends T_i*.mstamp* in the vote message to the coordinator. If the coordinator decides to commit T_i, it merges multistamps received from participants to obtain T_i's final multistamp. This multistamp is sent to participants in the commit messages. The participants store T_i's final multistamp in VQ[T_i].mstamp. Furthermore, for each page P modified by T_i, the participant merges this multistamp into PSTAMP[P]. Overheads of this scheme are determined by the processing overheads associated with multistamps. In Section 5.3.6, we present our truncation technique that keeps the size of multistamps small.

If the transaction is aborted, the participant receives the abort decision and removes information about T_i from the ILIST.

Figure 5-6 shows the multistamp of a transaction T_i and a page P being computed at server X when T_i commits; T_i is a single-server transaction that has modified object b (on page P) and read object a. Since page P is being cached by client D, an invalidation entry is generated for it and added to T_i's multistamp; T_j.mstamp is also merged into T_i.mstamp since T_i depends on T_j. We obtain P's multistamp by merging T_i's multistamp into its current value.

Unlike CLOCC, the VQ simply contains a transaction T_i's write set. The read set is needed just to compute T_i's multistamp at prepare time; it is not required to calculate the multistamps of later transactions or for validation purposes. In Section 5.3.7, we present a technique that allows clients not to send read sets to servers.

Fetch Processing. When a server receives a fetch message for object *x* on page P and a prepared transaction T_j has modified *x*, it waits for T_j to complete. Then it sends the fetch reply, which contains P and also PSTAMP[P]. (In CLOCC and Weak-CLOCC, the system never delays a fetch reply. However, the likelihood of a delay is low since the prepare window is very small.)

Invalidations.

To produce an invalidation message, the server goes through the ILIST in timestamp order from smallest to largest, stopping when it has processed the entire list, or it reaches an entry for a prepared (but not yet committed) transaction. The ids of all objects in the processed entries are sent in the message along with the *largest timestamp* contained in the processed entries. When a server sends an invalidation message to a client C and C's ILIST is empty, it sends its current clock value as the timestamp.

As mentioned earlier, invalidation messages are piggybacked on every message sent to a client and clients acknowledge invalidations. The acknowledgement contains the timestamp *ts* of the associated invalidation message. The server then removes all entries whose timestamps are less than or equal to *ts* from the ILIST.

A client may also send an *invalidation-request* message to the server containing a timestamp *ts*; this request indicates that the client wants all invalidations that were generated for it before time *ts*. The server responds by sending back an invalidation message as above except that the timestamp in the message must be greater than or equal to *ts*. It is possible that some entry in the table with a timestamp less than or equal to *ts* exists for a transaction that has not yet committed (it is still prepared); in this case, the server simply sends the invalidations due to that transaction optimistically assuming that it will commit. Furthermore, if *ts* is greater than the timestamp of all entries in the ILIST, the server responds with its current time; it waits for its clock to advance past *ts* if necessary, but a delay is extremely unlikely since clocks are loosely synchronized.

5.3.3 Processing at the Client

A client C is responsible for maintaining sufficient information about dependencies so that it can validate the transaction's reads at commit time. Client C maintains two tables that store information about servers it is connected to. LATEST[X] stores the timestamp of the latest invalidation message it has received from server X; REQ[X] is the largest timestamp for X that C is required to hear about if C accesses X and tries to commits its transaction; if REQ[X] $>$ LATEST[X], this means server X has invalidations for C that C has not yet heard about.

The client also maintains a set CURR that identifies all servers used by its currently running transaction. When the client receives an invalidation message from server X, it stores the timestamp in the message in LATEST[X].

Client C does the following when a transaction first uses object *x*:

1. Adds *x*'s server X to CURR.

2. Fetches *x* if it is not already present in the client cache. When the fetch reply arrives it updates the information in REQ to reflect the multistamp in the fetch response: for each multistamp entry $< C,\ Y,\ ts >$ such that *ts* is larger than REQ[Y], it stores *ts* in REQ[Y].

Invalidations are handled as in CLOCC, i.e., if the client's current transaction has accessed an object specified in the invalidation message, the transaction is aborted.

5.3.4 Validation

A transaction T$_i$'s reads are validated by its client and its writes are checked by the servers using Weak-CLOCC. To validate T$_i$'s reads, its client C ensures that C has recent-enough information about invalidations for all servers used by T$_i$, i.e., for each server X in CURR, the client checks that LATEST[X] $\geq$ REQ[X]. This check is called the *read-dependency check*. If this condition does not hold for some servers, a *consistency stall* occurs and the client sends an *invalidation-request* messages to each server X where the information was not sufficiently recent; in this message, the client requests server X to send all invalidations that were generated at X before time REQ[X]. When the client receives the replies, it processes them as in CLOCC, i.e., it aborts T$_i$ if the reply indicates T$_i$ has read an obsolete object; otherwise, T$_i$ passes the read-dependency check.

If transaction T$_i$ is read-only, there is no more work to be done and the transaction commits locally at the client. Otherwise, the client sends the read and write sets to the servers so that the Weak-CLOCC checks can be performed, and so that T$_i$'s multistamp can be computed. In Section 5.3.7, we will show an optimization in which the client computes the read-dependency part of the multistamp and sends it to the servers; this allows it to avoid sending the read sets in the commit request.

Figure 5-7 shows an example where the REQ and LATEST arrays change after a message. In client C's current transaction, servers X and Y are being accessed. When C fetches page P from X, it updates LATEST[X] with the time sent in X's fetch reply. It also updates REQ[Y] using P's multistamp. With this update, LATEST[Y] is now less than REQ[Y]. Thus, if C tries to commit its transaction now, there will be a consistency stall due to Y. No invalidation-request message will be sent to server Z since it is not being accessed in the current transaction (even though the read-dependency condition is violated for server Z).

In CLOCC, the validation information in the VQ is logged on stable storage as part of the prepare record by the participants. In the PL-2+ scheme, the final multistamp of a transaction T$_i$ is logged

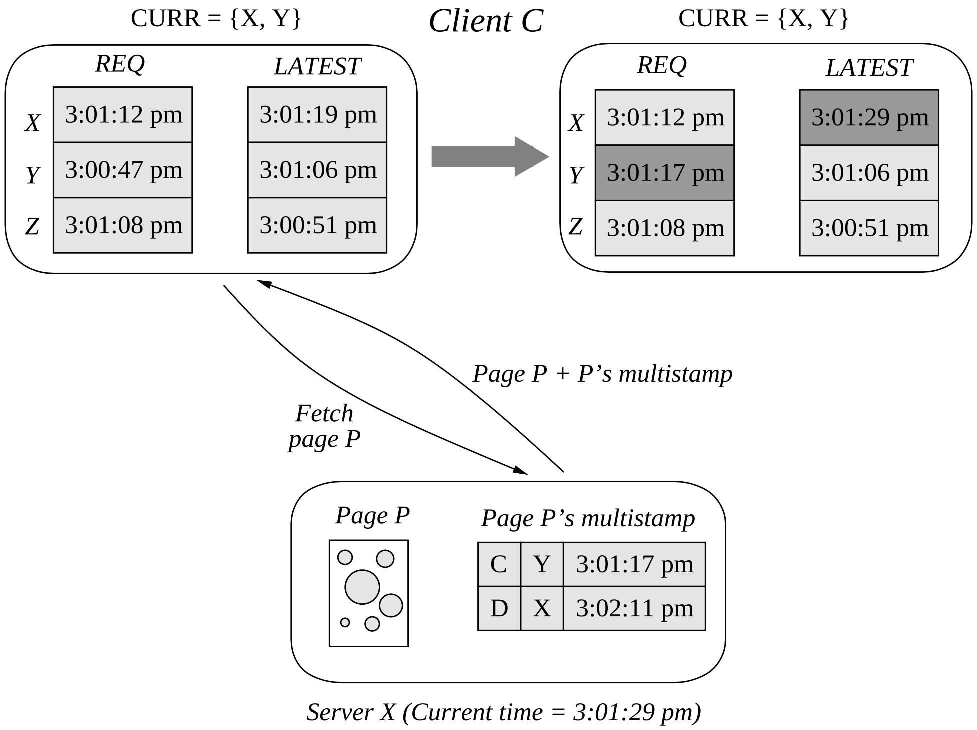


Figure 5-7: Handling of a fetch reply by client C (changed entries are shown in a darker background)

along with the coordinator's commit decision and can be recovered after a crash; similarly, the PSTAMP table can also be reconstructed from the prepare and commit records.

An advantage of the PL-2+ scheme over CLOCC is for read-only transactions. In CLOCC, all read-only transactions must communicate with the servers for validation purposes. In our PL-2+ scheme, a multi-server transaction T_i can be committed locally if the read-dependency check is satisfied for T_i at commit time. Our results in Chapter 7 show that most multi-server read-only transactions do not require communication with the servers at commit point. Furthermore, single-server read-only transactions can be committed locally without any communication. The reason is that when client C receives invalidations from server X due to a transaction T_j, it also receives invalidations of all transactions that have committed earlier at server X; this includes all transactions that T_j depends on at server X. Thus, when a read-only transaction T_i reads T_j's updates from server X, the no-depends-misses condition property must be satisfied with respect to server X for transaction T_i.

Correctness

The checks in Weak-CLOCC ensure that phenomenon G0 does not occur and disallowing dirty reads guarantees that G1 is disallowed. We now argue why the no-depend-misses condition (Section 4.1) is satisfied, i.e., G-single is disallowed.

The no-depend-misses condition requires that if T_j depends on T_i, it must not miss the effects of T_i. The notion of not missing the effects of a transaction is captured using invalidations and, ultimately, multistamps. Suppose that a transaction T_i modifies objects *x* and *y*. When T_j observes the *x*'s new value, the server can inform T_j's client that there *might* be an object modified by T_i which T_j must not miss (from other servers). This is essentially the information that is stored in invalidation entries of multistamps. Thus, for the purposes of PL-2+, T_i's update information is completely captured using invalidations/multistamps as a requirement on later transactions.

Given that multistamps represent dependency information, the correctness of our scheme relies on two properties: (i) multistamps reflect dependencies properly; (ii) multistamp information flows and is acted upon correctly.

Step 2(a) of the commit processing at the server ensures that the multistamp for a transaction T_i at server X contains tuples for all invalidations caused by it at X. Furthermore, step (3) ensures that T_i's multistamp contains all tuples in multistamps of transactions that T_i depends on at X. In the PL-2+ scheme, T_i's final multistamp is obtained by merging its initial multistamp generated during the prepare phase. By induction on the dependency relation, we can show that T_i's final multistamp contains tuples reflecting all invalidations of T_i and any transaction that T_i depends upon across *all* servers. Note that a transaction T_i's final multistamp is sent by the coordinator only to the read-write participants. This is acceptable since later transactions can become dependent on T_i only by reading objects from these servers.

Multistamp information is propagated and used appropriately in our scheme. The only way a transaction T_j becomes dependent on another transaction T_i is by reading a modification of T_i. This can occur only if T_j's client C fetches a page modified by T_i. But when such a fetch occurs, T_i's multistamp is sent to C in the multistamp of the returned page; the fact that we delay client C that fetches an object modified by T_i while T_i is prepared ensures that C is given T_i's final multistamp and not an intermediate value. Since client C merges the multistamp information in REQ, this data structure correctly captures the dependency constraints on T_i's reads. Thus, when the read-dependency check is performed at commit time, the client ensures that T_i has not missed the effects of any transactions it depended on, i.e., the no-depend-misses condition is satisfied.

5.3.5 EPL-2+ for Running Transactions

It is easy to modify the above scheme to provide EPL-2+; we simply perform read-dependency checks while the transaction is executing.

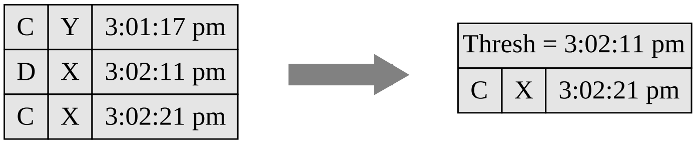
A client C that wants to provide EPL-2+ guarantees to its running transaction T_i ensures that the cache is kept EPL-2+-consistent with respect to the currently accessed servers (CURR). The multistamp-based scheme is modified so that the read-dependency condition (LATEST[X] $\geq$ REQ[X]) is true for each server in CURR *during* T_i's execution, i.e., client C guarantees that it has received all invalidations specified in a page P's multistamp before T_i reads P. The read-dependency condition can be violated in two ways: when C fetches a page from a server and, when a server is accessed for the first time in a transaction and added to CURR (since the cache may not be consistent according to EPL-2+ specifications with respect to a server not present in CURR). As in the PL-2+ case, if the read-dependency condition fails for a server X, a consistency stall occurs and the client sends an invalidation-request message to X asking for all invalidations that were generated before time REQ[X] and waits for a reply. Thus, the cost of providing EPL-2+ shows up in the form of consistency stalls as the transaction executes. In [AL97], we have described our multistamp-based scheme for providing EPL-2+ to running transactions when CLOCC is used for committed transactions.

5.3.6 Truncation

Due to the transitive nature of the dependency relationship, the size of the multistamp can become prohibitively high in a distributed system with a large number of servers and clients. Large multistamps can lead to larger messages, high memory storage costs at servers and increased server CPU overheads for manipulating them. We now present a mechanism called *multistamp truncation* that keeps multistamps small and addresses all these problems. We also show how to keep the VQ and PSTAMP tables small.

How Truncation is Performed

All our optimizations for truncating information rely on the fact that our multistamps contain real time values. We use these clock values to determine which tuples are old and then remove them from multistamps. To account for the removed tuples, each multistamp *m* also contains a timestamp *m.threshold*; *m.threshold* is greater than or equal to timestamps of all tuples that have been removed from *m*. The threshold allows us to compute an *effective multistamp* EFF(*m*) containing a tuple $\langle\ C,\ X,\ ts\ \rangle$ for every client/server pair, where *ts* is the timestamp in the tuple for client C and server X in *m* if one exists and otherwise, *ts* is *m.threshold*. Suppose that *m* is truncated to obtain a new multistamp *m'*. We know that this truncation preserves dependencies because, the timestamp for any client/server pair in EFF(*m'*) is greater than or equal to the timestamp for the pair in EFF(*m*), i.e., the truncated multistamp is a more conservative/restrictive version of the original multistamp. Figure 5-8 shows an example of truncation where the entries below 3:02:11 pm are removed and replaced by a threshold timestamp.



C	Y	3:01:17 pm
D	X	3:02:11 pm
C	X	3:02:21 pm

Thresh = 3:02:11 pm		
C	X	3:02:21 pm

Figure 5-8: Multistamp truncation.

When a client receives a truncated multistamp *m* from a server X, it computes EFF(*m*) and updates REQ[X] using this value; it then proceeds as described in Section 5.3.3. Thus, the correctness of our scheme not sacrificed since a truncated multistamp simply increases the client's requirements; truncation does not cause a client C to miss any invalidations that C was required to receive with the original multistamp.

The result of the truncation process is a loss of precision in multistamps. For example, in truncating a multistamp, the server may have removed an entry <*D, X, ts*> concerning some client D different from C. When that multistamp arrives at C, C may increase REQ[X] unnecessarily. During the read-dependency check at client C, C may request invalidations from X even though X has no invalidations for it. Thus, there is a tradeoff involved with the size of the multistamp and the number of consistency stalls. Chapter 7 examines this tradeoff and shows that small multistamps (less than 100 bytes) are sufficient to ensure that consistency stalls have a minimal impact on performance.

Using Real Time to Remove Old Entries

We now describe *when* servers decide to truncate multistamps. As mentioned in Section 5.1.1, clients receive invalidation information from servers within a *timeout* period of when it was generated. This communication implies that if a server has a tuple containing a timestamp *ts* that is older than the server's current time by more than this period, this tuple is not useful since invalidations specified in it will almost certainly have been propagated to the relevant clients by this time. Therefore, such old entries are *aged*, i.e., automatically removed from multistamps.

Simple aging of tuples may not be enough to keep multistamps small, so we also *prune* the multistamp by removing tuples that are not old. The system bounds the size of multistamps: whenever a multistamp *m* exceeding this size is generated, it is immediately pruned by removing some tuples. Pruning occurs in two steps: First, if there is more than some number of tuples concerning a particular server, these tuples are removed and replaced by a *server stamp*. Then, if the multistamp is still too large, the oldest entries are removed and the threshold is updated. A server stamp is a pair <*server, timestamp*>; EFF(*m*) expands a server stamp into a tuple for each client for that server and timestamp.

The VQ and PSTAMP are also truncated. Retaining information in the VQ about transactions that committed earlier than the timeout period is also not useful. Whenever the multistamp of a transaction contains no tuples (i.e., it consists only of a threshold), it is dropped from the VQ. The VQ has an associated multistamp VQ.*threshold* that is greater than or equal to the (effective) multistamps of all transactions dropped from VQ. When a server generates a transaction T_i's multistamp (T_i.mstamp), it initializes T_i.mstamp to be VQ.*threshold*. Recall that CLOCC also removes entries from the VQ using a similar strategy. Thus, to take both these schemes into account, we remove an entry from the VQ when it is not needed by both algorithms.

Information is dropped from PSTAMP in the same way, with information about multistamps of dropped entries merged into PSTAMP.*threshold*; if a fetch request for page P arrives and PSTAMP contains no entry for P, the fetch response contains PSTAMP.*threshold*.

For both the VQ and PSTAMP, the server can remove entries earlier at any point in time; it just needs to ensure that the associated threshold multistamps are updated accordingly. Thus, a server maintains entries only for recently committed transactions in the VQ and information only about recently modified pages in PSTAMP.

The above decisions for truncating multistamps are based on the fact that clocks are loosely synchronized in the system. The use of loosely synchronized clocks to truncate old information is a well-known technique; CLOCC also uses this idea to truncate the VQ. The reader can see [Lis93] for other practical uses of loosely synchronized clocks.

Using Multipart Timestamp Truncation in Other Systems

Multistamps have been proposed and used in many distributed systems [BSS91, LLSG92, TTP$^+$95, ACD$^+$96] for capturing causality constraints. Our truncation technique can be used for keeping multistamps small in some (but not all) parts of these systems. We enumerate guidelines that can be used to determine if our truncation technique is applicable to an implementation. These guidelines cover a large class of schemes in the literature but are not supposed to be sufficient or necessary conditions for using our truncation mechanism because multistamps are used in a variety of ways to capture causality constraints:

1. Each multistamp entry contains a logical clock value so that it can be replaced by a real clock value.

2. A multistamp represents a requirement on some entity (e.g., clients in our case) and a stronger requirement must not affect the correctness of the scheme; this also implies that a truncated multistamp must represent a stronger constraint than the original multistamp.

3. Exact values are not required for the correct functioning of the scheme.

Our implementation satisfies all the conditions stated above. We now consider the lazy replication system [LLSG92], and discuss where our truncation technique can be applied and where it cannot be; the arguments are similar for other systems.

In lazy replication, a database is replicated at several server replicas. Each server X maintains a *server multistamp* that contains an entry for every other server Y in the system. Each entry contains a time *ts* indicating that X has received all updates from Y generated before *ts* (Y's clock time). Clients also maintain multistamps; when a client sends a request to a server X, the operation is delayed until X's multistamp dominates the client's multistamp (a multistamp A dominates multistamp B if all entries in A are greater than or equal to the corresponding entries in B).

Server multistamps cannot be truncated since an exact value is required in each entry; thus, the third guideline given above is violated. Approximating some entries by a threshold timestamp will incorrectly indicate that messages have been received from all servers up to the threshold. To reduce the network overheads of server multistamps, techniques such as sending multistamp differences [BSS91] will have to be used.

Our truncation technique is applicable to client multistamps since they act as requirements on servers; servers cannot process client requests until they are sufficiently recent. Truncation simply makes the requirements more conservative: if some entries in a client C's multistamp are replaced by a threshold timestamp, C can access server X only if X has received messages that are marked higher than the threshold timestamp from all servers in the system.

In general, for many schemes, when multistamps are used for represent the state of the system, it may not be possible to truncate them. Thus, the server multistamps cannot be truncated, whereas client multistamps can be truncated since they represent causality constraints/requirements.

We now discuss some general performance considerations for truncating a multistamp. When truncation is performed on a multistamp, we lose some fine-grained information about individual entries. This loss of information usually shows up in the form of an inefficiency in some other part of the system, e.g., as extra messages or blocking of operations. In our PL-2+ scheme, it results in extra invalidation-request messages being sent to the servers. To reduce the impact of this performance penalty, it is desirable that periodic communication occurs among the relevant processes in the system. In our system, "I'm alive" messages exchanged between a client and servers ensure that the client is kept reasonably recent with respect to all its connected servers. Similarly, "gossip" message are sent in lazy replication [LLSG92] to ensure that servers are reasonably up-to-date with respect to other servers; these messages reduce the likelihood that a client operation is delayed at a server.

In lazy replication, the threshold timestamp in a client-request to server X will require that X delay the operation until X has received messages higher than the threshold from *all* servers in the system; this can result in significant delays for the client's operation. Thus, the notion of "all servers" denoted by the threshold needs to be *localized* so that the threshold constraint does not cause a significant performance penalty; this can be done by exploiting the system structure or the semantics of the application. For example, in our PL-2+ scheme, by taking advantage of the system structure, the threshold constraint on a client is reduced to a requirement with respect to servers accessed in the client's current transaction; this localization reduces a large number of unnecessary consistency stalls.

5.3.7 Offloading Multistamp Generation to Clients

We now discuss an optimization that allows clients (instead of servers) to generate multistamps, making the scheme more scalable. When a client C receives a page P's multistamp, it retains the multistamp in a table, CLIENTPMAP; this table maps page ids to their multistamps for all pages in C's cache. The CLIENTPMAP table is maintained in a manner similar to the PSTAMP table, i.e., only multistamps of recently fetched pages are kept; other multistamps are summarized using a threshold multistamp. When a transaction T_i reaches its end, the client computes T_i's initial multistamp by merging the multistamps of pages accessed by T_i and sends it to the coordinator. Each participant simply sends tuples generated due to T_i's invalidations in its vote to the coordinator, who merges these tuples to obtain T_i's final multistamp. Thus, the servers do not perform any intersections to compute T_i's multistamp. We refer to this scheme as the *client-merger scheme*.

Our original approach for generating multistamps used exact write sets from the VQ and T_i's read set. However, in the client-merger scheme, T_i's initial multistamp is computed at a client based on coarse-grained information, i.e., pages accessed by T_i (since the CLIENTMAP table is maintained at a page granularity). The loss of precision can result in larger multistamps, and hence, more truncation, leading to a higher number of consistency stalls. To alleviate this problem of false dependencies, the server maintains an *object bitmap* for each entry in the PSTAMP table; this bitmap keeps track of the objects modified by recently committed transactions on each page. (As mentioned earlier, objects do not span pages in our system.) When a transaction T_j modifies page P and enters P's multistamp into PSTAMP, the server sets the bits in the bitmap corresponding to the objects modified by T_j; the bitmap is removed when P's multistamp is removed from PSTAMP (at this point, P's multistamp is merged into PSTAMP.threshold). When a client fetches page P, this bitmap is also sent in the fetch reply along with PSTAMP.threshold. The client keeps the bitmap in the CLIENTPMAP table and also maintains the last known PSTAMP.threshold for each server. When a client generates T_i's initial multistamp and T_i has read *x* on page P, it merges P's multistamp from CLIENTPMAP only if *x* is marked in P's bitmap; otherwise, it merges PSTAMP.threshold of the server that owns P.

The client-merger scheme has advantages over our original mechanism. First, server CPU overheads are reduced since the server does not have to perform any intersections for generating a transaction's multistamp; with this optimization, servers are responsible only for performing the checks in Weak-CLOCC, generating invalidations, and obtaining the final multistamp of a committing transaction. Second, a transaction's read set does not have to be sent to the servers. Furthermore, if a server X is a read-only participant, it is not sent any message by the client because no invalidations will be generated at X.

5.4 PL-3U Mechanism for Read-only Transactions

We now extend our multistamp-based scheme for providing PL-3U to read-only transactions. We also show how the PL-3U scheme can be modified to provide EPL-3U and EPL-3 for running transactions. For update transactions, we continue to use CLOCC. The advantage of our PL-3U scheme over CLOCC is that the client rarely needs to communicate with the servers for committing read-only transactions. Thus, this scheme is strictly *better* than CLOCC in terms of message requirements and delays (and it provides strong guarantees that are close to serializability).

Our implementation for committing a read-only transaction T_q at PL-3U ensures that the no-update-conflict-misses condition is satisfied, i.e., if the client running T_q observes a modification of transaction T_i, it can commit T_q locally only if it has received all the invalidations of T_i and any update transactions that T_i depends or anti-depends on. Thus, multistamps need to capture dependencies *and* anti-dependencies.

We make the following change to the multistamp-based scheme for PL-2+. When a transaction T_i prepares at a server X, the server computes the multistamp by merging the multistamps of all transactions that T_i depends and anti-depends on. In the client-merger scheme for PL-2+, the PSTAMP table maintains the multistamps of all transactions that have modified a page P at the server. For PL-3U, we also need to maintain anti-dependency information for pages; for this purpose, we use a new table, RPSTAMP, which maps a page P to the (merged) multistamp of all update transactions that have *read* P. For each page P modified by T_i, it merges RSTAMP[P] into T_i's multistamp.

If a read-only transaction T_q requests level PL-3U and accesses objects from only a server X, it can be committed locally regardless of the values of LATEST[X] and REQ[X] (recall that single-server transactions can be committed locally at level PL-2+ as well). In this scenario, if T_q observes the effects of a transaction T_i from X, it will not miss the effects of *any* committed update transaction that T_i depends or anti-depends on. The reason is that after T_i prepares at X, CLOCC prevents any transaction that conflicts with T_i to prepare with a timestamp earlier than T_i. Thus, T_q can be serialized after all update transactions (such as T_i) that T_q depends or anti-depends on.

The PL-3U scheme has higher overheads compared to the PL-2+ scheme. First, in the PL-3U implementation if a preparing transaction T_i anti-depends on a prepared transaction T_j, T_i's prepare is delayed till T_j commits or aborts (we expect these delays to be short since a coordinator sends the commit decisions promptly). This delay is needed to ensure that the server merges T_j's final multistamp into T_i's multistamp; such a delay does not occur in the PL-2+ implementation since anti-dependencies are not captured at PL-2+. Second, since transaction read sets are needed for computing anti-dependencies, the client needs to send them to servers in the PL-3U implementation; this increases the network bandwidth consumption and the server CPU utilization for receiving the read sets. Third, the server CPU is further utilized in the PL-3U scheme since the server has to perform the intersections in the RPSTAMP table; this computation must be performed at the server since page P may have been read by another client after a transaction T_i's client fetches P, i.e., at T_i's commit point, T_i's client may not know about the multistamps of all transactions that have read P. Fourth, the likelihood of a read-only transaction requiring communication with the servers at commit time is higher in PL-3U as compared to PL-2+ since multistamps contain stronger requirements for clients. Finally, as we show below, this scheme requires an additional mechanism to ensure that the read-only participant optimization for two-phase commit is allowed.

5.4.1 Read-only Participant Optimization for PL-3U Implementation

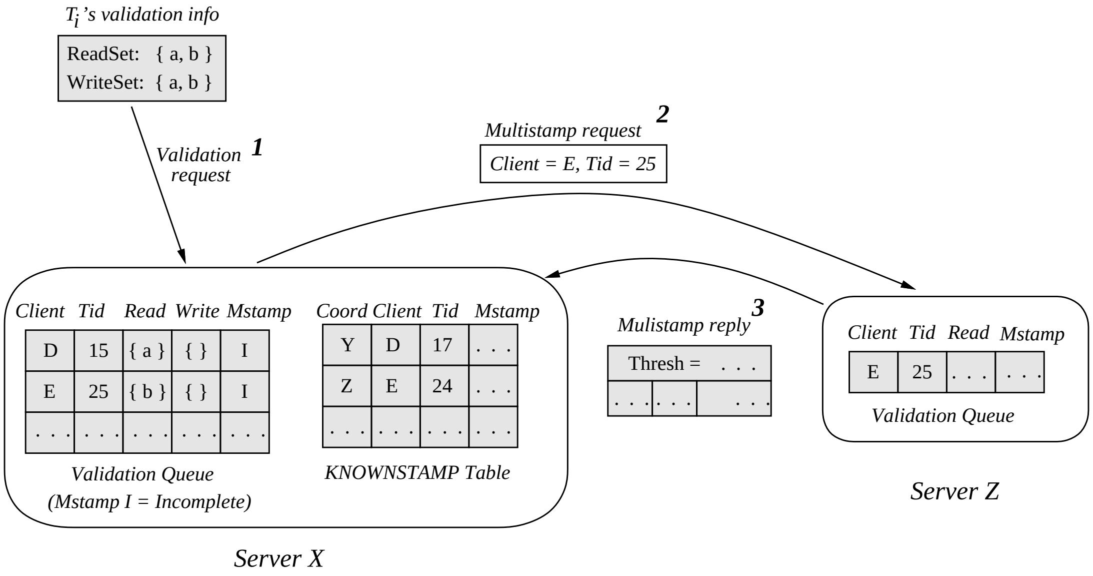
The PL-3U scheme presented above assumes that all participants receive the commit decision since a preparing transaction T_i must merge the final multistamp of all transactions that T_i depends or anti-depends on. If T_i anti-depends on a prepared transaction T_j at server X, X delays T_i's prepare until T_j's commit decision is received. However, with the read-only participant optimization, the commit decision is not sent to read-only participants [GR93]. This implies that if X is a read-only participant for T_j, X will not receive T_j's final multistamp and T_i would be stalled indefinitely.

We could abandon the read-only participant optimization, but this seems unwise since we expect read-only participants to be common. Therefore, we use the following technique: we place the onus of obtaining T_j's final multistamp on the update transaction's (i.e., T_i) commit protocol. In the above scenario, when T_i prepares at X, anti-depends on T_j, and T_j's final multistamp is not known, X sends a *multistamp-request* message to T_j's coordinator; this extra communication occurs during T_i's prepare phase, causing a *prepare block*. T_j's coordinator sends T_j's final multistamp to X; this reply is delayed if T_j is still prepared. Note that X can also send the multistamp-request message to T_j's client but we do not use this approach since clients can crash and cease to exist whereas servers are highly available in Thor.

However, this protocol has a problem. If read-only participants are common (as is expected), this approach can cause a substantial number of prepare blocks for update transactions; a prepare block will occur whenever an update transaction T_i anti-depends on *some* earlier transaction T_j at server X and T_j is read-only at X. We can avoid such blocks by piggybacking some information to servers and taking advantage of the fact that clients communicate with servers at least once every timeout period (see Section 5.1.1). Each client C keeps track of the transaction id or *tid* of its last committed update transaction called CURRTID along with a multistamp called CURRUSTAMP (a transaction id is a tuple that contains the client's id and its current clock value). This multistamp is obtained by merging the multistamps of all update transactions committed by C; it acts as an upper bound on the multistamp of each transaction that was earlier committed by C. In each message to a server, client C sends its CURRTID and its CURRUSTAMP. The server maintains a table called KNOWNSTAMP that maps clients to the most recent values of CURRTID and CURRUSTAMP received from that client.

Thus, in the above scenario, when update transaction T_j prepares at server X, X stores T_j's tid and coordinator in its VQ entry, and marks it as *incomplete*. When T_i prepares at X, X determines all incomplete transactions that T_i depends or anti-depends on. For each such transaction T_j, if T_j's tid is less than or equal to the CURRTID specified in KNOWNSTAMP for T_j's client, X merges that client's CURRUSTAMP into T_i's multistamp; otherwise, it sends a multistamp-request message to T_i's coordinator. Thus, multistamp-request messages are sent to other servers only when T_i commits at server X, anti-depends on T_j, X is a read-only participant for T_j, *and* X has not received a message from T_j's client after T_j's commit. The probability of this situation is low since T_j's client sends a message to X at least once every timeout period and anti-dependencies on recently committed transactions are unlikely in common workloads.

Since CURRUSTAMP is an upper bound and not an exact value, some false dependencies and anti-dependencies may be captured resulting in extra transaction stalls. However, as shown in Chapter 7, the performance impact of stalls (which includes such stalls) is low even for stressful workloads.



Client	Tid	Read	Write	Mstamp
D	15	{ a }	{ }	I
E	25	{ b }	{ }	I
. . .	. . .	. . .	. . .	. . .

Coord	Client	Tid	Mstamp
Y	D	17	. . .
Z	E	24	. . .
. . .	. . .	. . .	. . .

Client	Tid	Read	Mstamp
E	25	. . .	. . .

Figure 5-9: Transaction T_i's prepare is blocked since it anti-depends on a transaction whose multistamp is not known at server X.

Figure 5-9 shows an example of a transaction T_i that anti-depends on two transactions that were read-only at server X. Their multistamp entries are marked as incomplete. Server X checks the KNOWNSTAMP table for the multistamp values of these transactions and determines that an upper bound is available for the transaction with tid 15 but information for the transaction with tid 25 (committed by client E) is not present. Thus, server X sends a multistamp-request message to the transaction's coordinator, server Z, and waits for the reply.

Certain optimizations can improve the performance of the above scheme. First, the probability of sending a multistamp-request message can be reduced further if it is sent only when the update transaction (i.e., T_i in the above case) is a multi-server transaction; otherwise, we simply add T_j's information to T_i's VQ entry and mark T_i as incomplete. Now X needs to send a multistamp-request message to T_j's coordinator only when a multi-server transaction arrives at X and depends or anti-depends on T_i or T_j; we expect the scenario of a multi-server transaction depending on another recently committed multi-server transaction to be rare. We call this the *single-server delay* optimization.

Second, we could be lazier: rather than having T_i find out T_j's final multistamp, we could let this work be done by a transaction that occurs later than T_i in the chain of dependencies and anti-dependencies. For example, if transaction T_k reads T_i's updates and we place this onus on T_k, it is more likely that X knows T_j's final multistamp at T_k's prepare than at T_i's prepare. This scheme might reduce prepare blocks further but it is considerably more complicated and our performance results in Chapter 7 show that prepare blocks are rare. Thus, we believe that this optimization is not needed.

Like the PL-2+ scheme, the server can log the PSTAMP and RPSTAMP information along with the commit decision. For read-only participants, we can avoid logging the RPSTAMP to stable storage by using a technique similar to one presented in [AGLM95]. The basic idea is that a server maintains a stable threshold timestamp that is an upper bound on all the timestamp entries in RPSTAMP. When a server crashes and recovers, the stable threshold timestamp is used to initialize the threshold multistamp of RPSTAMP. To avoid frequent updates of the stable threshold timestamp, we update this value in jumps (e.g., a few seconds). To avoid delaying a client at a read-only participant, this threshold can be increased in the background, e.g., by piggybacking its modifications on log updates by other transactions; more details about this mechanism can be found in [AGLM95].

The CURRTID and the CURRUSTAMP values of different clients can be flushed lazily to stable storage. If a crash occurs, the server can lose this information; when the servers recovers, it will initialize this data about clients with older values. This loss of latest information can result in some extra messages but does not affect correctness; this is not a problem since crashes are expected to be rare.

5.4.2 Providing EPL-3U and EPL-3 to Running Transactions

Running transactions can be provided EPL-3U guarantees by combining the technique presented in this section with the EPL-2+ scheme presented earlier. That is, a client computes the multistamps as presented above and checks for stalls when it fetches a page from as server and when it uses a server for the first time in a transaction. The likelihood of stalls is higher in EPL-3U scheme since multistamps contain stronger constraints than in the EPL-2+ case.

To provide EPL-3, we require that all transactions (including read-only transactions) are committed at the servers using CLOCC. To generate a transaction T_i's multistamp, the server also merges the multistamps of read-only transactions that T_i anti-depends on; recall that only multistamps of update transactions are merged in for PL-3U and EPL-3U. In the client-merger scheme, the RPSTAMP table also needs to maintain multistamps of read-only transactions as well. The rest of the EPL-3 implementation is the same as the EPL-3U implementation.

5.4.3 Requirements on Concurrency Control Implementations

The PL-2+ and PL-3U schemes can be used for efficiently committing read-only transactions while CLOCC can be used for committing update transactions. Even though we presented these mechanisms in conjunction with CLOCC, they can be used in hybrid systems [CG85] where update transactions are executed using a concurrency control scheme such as locking (e.g., ACBL [ZCF97]) and read-only transactions are executed optimistically (using these implementations). Our mechanisms will be beneficial in such hybrid systems because read-only transactions need not hold locks. As a result, there may be fewer delays in the system. For example, with ACBL, fewer callbacks may be sent by a server; also, the callback replies need not be delayed at a client if a read-only transaction is accessing the object.

To commit read-only transactions using our PL-2+ scheme, the concurrency control mechanism must provide serializability for update transactions and use a protocol like two-phase commit for atomicity; this result has been shown in [CG85]. To commit read-only transactions at PL-3U using our scheme, an additional property must be satisfied:

> **No transaction slipping in the past:** After an update transaction T_i prepares at a server X, X will not serialize another transaction T_j *before* T_i (in the equivalent serial order) if T_j conflicts with T_i.

This property ensures that T_i's final multistamp does not miss the information of any transaction that T_i depends or anti-depends on. A locking scheme like ACBL guarantees this property since a transaction T_j that conflicts with T_i and arrives after T_i, will be serialized after T_i (T_j acquires locks after T_i releases them). CLOCC also satisfies this property; as disussed in Section 5.1.1, its aborts a preparing transaction T_i if it tries to serialize before T_j and conflicts with T_j.

5.5 Related work

In this section, we discuss earlier work in the literature related to our optimistic schemes. Since the literature for concurrency control schemes is vast, we only discuss work that is closely related to our mechanisms.

5.5.1 Optimistic Schemes

Eswaran *et al.* [EGLT76], and later Kung and Robinson [KR81], suggested the idea of using optimism for concurrency control. Since then a number of optimistic schemes have been discussed in the literature.

Other distributed schemes achieve a global serialization order using atomic multicast [RT90] or logical clocks [ABGS87, Gru89]. Atomic multicast adds additional per-message overhead, while logical clocks must be explicitly managed as part of the two-phase commit, complicating the algorithm. Our use of loosely synchronized clocks avoids all these problems; more importantly, it also allows us to make time-dependent decisions and optimize our algorithm, e.g., to determine what transactions can be removed from the validation queue.

Apart from CLOCC, other backward validation algorithms have been proposed in the literature [ABGS87, BOS91, CO82, KR81, RT90]; forward validation schemes such as O2PL [FCL97] have also been suggested. In a client-server system, forward validation requires a validating transaction to contact all clients that are caching updated objects, to obtain latches on the cached copies. If a latch cannot be obtained, the transaction aborts; otherwise, it commits and releases the latches (updating or invalidating the client caches). This approach adds an additional communication phase to all commits, even when only a single server is involved; moreover, the delay incurred is observed by the committing client. In contrast, backward validation just uses the standard 1 or 2 phases required for a single-site or distributed commit (as described for our scheme), and therefore is a better choice for a client-server system.

Previous optimistic schemes have largely ignored implementation issues regarding time and space overheads. Some schemes [ABGS87, CO82, KR81, LW84] validate a transaction against all transactions serialized between its start and end times; a large amount of validation information must be stored to validate a long transaction. In our scheme, the invalid set summarizes most information required for validation in a compact way, while our VQ, PSTAMP, and other data structures for the weak consistency schemes are truncated according to a bound on expected message delay; the information we maintain is not proportional to transaction length, but still allows us to correctly validate long transactions.

Most optimistic schemes store some concurrency control information per object (e.g., the scheme in [ABGS87] maintains two timestamp values with each version). Space overhead for version numbers can be significant for small objects; e.g., most objects in the OO7 benchmark [CDN93] are smaller than 100 bytes; an 8-byte version number would add 8% overhead. Another potential problem with version numbers is that they are usually cached and uncached with the rest of their object state; missing version numbers would then require disk reads during validation. In contrast, all of our validation information can be kept in main memory.

Multi-version schemes [ABGS87, BOS91, LW84] keep multiple versions of objects to provide a consistent view of the database to all active transactions, making validation unnecessary for read-only transactions. However, this approach has very high space overheads: multiple versions need to maintained in different parts of the system, including the server cache and the disk. As a result, the effective server cache size and the effective disk bandwidth available to the server may be reduced. Furthermore, there is the additional complexity of maintaining and discarding multiple versions. Maintaining multiple versions also requires storing version numbers in objects — a cost that we wanted to avoid.

5.5.2 PL-2+ Mechanisms and Causality

Chan and Gray [CG85] have proposed PL-2+ as a correctness criterion for read-only transactions and described a mechanism to commit these transactions in a distributed system such that they do not interfere with update transactions. Although their scheme also uses multistamps, their multistamps are different from ours and suffer from a variety of inefficiencies, making their mechanism impractical. In their scheme, multistamps capture the dependency information using transaction ids; hence, it is difficult to truncate this information. Also, multistamps in their scheme capture dependencies in a very conservative manner: a server X assumes that a preparing transaction is dependent on *all* transactions that have already committed at X. In our multistamp-based schemes, this assumption can lead to a large number of unnecessary consistency stalls.

Multipart timestamps have been used widely in distributed systems for capturing causal relationships [BSS91, LLSG92, TTP$^+$95, KCZ92, ANK$^+$95]. However, multipart timestamps can become quite large if there are tens of thousands of processes involved. This can result in large messages, higher server memory usage, and higher server CPU consumption to manipulate these multistamps. To reduce network overheads imposed by multistamps, researchers [BSS91] have suggested a "difference" mechanism in which a process P only sends those multistamp entries to another process Q that P knows Q does not have; this optimization is useful in cases where changes to multistamps are small compared to the size of the multistamps so that the differences result in smaller network messages. However, this solution does not solve all the problems due to large multistamps, especially server storage and server CPU overhead problems: performing a difference between large multistamps can be CPU and memory intensive. Our solution uses loosely synchronized clocks and gets rid of entries in multistamps; as we show in Chapter 7, the cost of approximating the multistamp information is low for our environment.

Snapshot Isolation has been implemented in the Oracle server [Ora95] but we do not have information about distributed client-server implementations of this isolation level. We believe that an implementation for Snapshot Isolation in such environments will have higher communication and space overheads than our PL-2+ implementation because Snapshot Isolation has stronger requirements than PL-2+; extra messages or more versions may need to be maintained just to ensure that a client has a snapshot of the database as of some point in the past. On the other hand, PL-2+ has weaker requirements and our multistamp-based implementation for PL-2+ is lazy: it simply requires that object versions be present in a client's cache according to dependency constraints and sends messages only when there is a potential violation of consistency.

5.5.3 Orphan Detection Mechanisms

Our EPL-2+ scheme for running transactions has been primarily designed to ensure that the application code does not view consistent data and behave in an unpredictable manner. This was also one of the main reasons that motivated the work on *orphan detection* [LSWW87, MH86]. An orphan is a transaction that is doomed to abort. Such a transaction can waste resources and view inconsistent data. Orphan detection and elimination was implemented in the Argus distributed system [LSWW87, Ngu88] to solve these problems. The Argus model consists of multiple processes in the system that communicate with each other via remote procedure call. Orphan detection schemes, like our mechanisms, use the idea of (transitive) propagation of information. However, these orphan detection schemes have much higher space overheads and higher complexity. For example, the consistency information in the Argus schemes contains a list of transaction ids and process ids that is more difficult to compress than our multistamps. Mechanisms to garbage collect old information in these schemes can also result in high overheads. As shown in [Ngu88], some of the orphan detection protocols can impose significant performance penalties.

5.5.4 Read-only Transactions

Researchers in the past have suggested a variety of schemes to optimize read-only transactions [Wei87, GW82, ABGS87]. The main purpose of these mechanisms is to ensure that read-only transactions do not interfere with update transactions. Hence, these schemes use multiple object versions allowing read-only transactions to read old versions while update transactions modify the current version. Our schemes for read-only transactions can also be extended to use multiple versions. However, as discussed earlier, we decided against such schemes because of the costs involved.

The protocols suggested by Weihl [Wei87] for read-only transactions suffer from excessive space overheads or require extra messages in the system compared to our schemes. One of the schemes assigns timestamps to a read-only transaction T$_r$ when it begins; T$_r$ reads object versions according to its assigned time. The problem with this approach is that it requires an *initiation phase* for a read-only transaction, i.e., before T$_r$ starts, it must communicate its timestamp to all sites that it is going to access. This scheme also requires predicting the objects or sites that the transaction is going to access; such predictions are not needed in our schemes. Furthermore, any timestamp-based scheme where a read-only transaction reads object versions according to its timestamp requires access times to be stored with an object, thereby converting every read into a write. This information is needed so that later update transactions can be assigned a higher timestamp. The scheme suggested in [Ree78] also has a similar requirement.

In another scheme suggested by Weihl, each transaction T$_i$ maintains dependency and anti-dependency information and stores it with the versions that T$_i$ creates. A read-only transaction T$_r$ uses this information to determine which object versions to read. However, this approach is impractical since the dependency/anti-dependency information consumes a large amount of space and garbage collecting this information is difficult. Like the previous scheme, this protocol also suffers from the problem that it converts every read into a write; when a read-only transaction T$_r$

accesses an object *x*, it must update *x*'s dependency/anti-dependency information.

5.6 Summary

This chapter presented a variety of optimistic schemes for distributed client-server systems that provide different isolation degrees to running and committed transactions. In all our schemes, we have taken advantage of the system structure and also utilized characteristics of real systems to optimize our mechanisms, e.g., we use loosely synchronized clocks to truncate some of our data structures. We have also taken care to avoid adding a significant number of messages to the system; wherever possible, we try to piggyback the relevant information on existing messages. This chapter described efficient implementations for providing PL-2+ and PL-3U and the corresponding isolation guarantees for running transactions, EPL-2+ and EPL-3U. An important advantage of weaker isolation level implementations over CLOCC is for read-only transactions. In CLOCC, such transactions require communication with the servers at commit time whereas most read-only transactions can commit locally at the client machine using our PL-2+ and PL-3U implementations. Our mechanisms are based on multi-part timestamps or multistamps and we presented a simple technique called multistamp truncation that allows multistamps to be kept small in large distributed client-server systems.

Chapter 6

Experimental Framework

This chapter describes the experimental setup for evaluating the consistency mechanisms that were discussed in the previous chapter. The experiments were performed with the help of an event-driven simulator. We used a simulation technique rather than implementing the schemes in a client-server system such as Thor [LAC$^+$96] because we wanted to evaluate the performance of our techniques in the presence of a large number of servers and clients. Apart from scalability considerations, we also wanted to evaluate our schemes with a range of system parameters such as different network speeds, different CPU speeds, etc.

We constructed the workloads starting from earlier concurrency control studies [Gru97, CFZ94]. These studies were performed for a single-server, multi-client system; we extended them to a distributed database with multiple servers. Some parameters such as the disk and network models, database model at each server, and the access patterns in two of our workloads have been derived from Gruber's work [Gru97]; the simulator scheduling model and some parameters for CPU processing overheads have also been borrowed from his study. Parameters for disks and networks were taken from products that are currently being used in the market and in office environments. Finally, CPU overheads for some operations such as multistamp processing were derived by profiling our simulator: some parts of the simulator code have been derived from the Thor implementation [CALM97, LAC$^+$96]; other parts have been structured such that they can be easily migrated to a real implementation.

In our experiments, we make assumptions similar to the ones that have been made by concurrency control studies in the past [Gru97, ZCF97, CFZ94]. First, we assume that the client cache is large enough to keep all objects touched by a single transaction. This assumption holds for many applications and systems; current memory trends and recent work on efficient cache management [CALM97] suggest that this assumption will continue to hold in the future for many applications. We call these transactions *cacheable transactions*. Second, transactions are restartable, i.e., if a transaction aborts, the code of the aborted transaction can be re-executed; of course, a rerun may access a different set of objects. Thus, the application code must be written to handle aborts; this is a reasonable assumption since applications need to prepared for aborts due to resource problems, deadlocks (in a locking implementation), etc. Third, transactions are medium-sized, i.e., each transaction accesses a few hundred objects and lasts for a relatively short time, i.e., a few seconds. Long-running transactions are usually handled using application-specific concurrency semantics [BK91]. These implementations typically use serializability-based mechanisms as basic primitives for manipulating groups of objects in the database. For example, a check-in/check-out approach for cooperative design work may use atomic operations to retrieve data from the database and release the acquired locks after the retrieval.

Our simulation setup models a system with a large number of servers and clients. We have scaled down some of the system parameters to make the simulation more tractable. However, the scaling has been performed so that we still maintain the important aspects of system components that can affect the relative performance of different isolation mechanisms. For example, even though the total size of the data stored at each server in our simulations is 3.4 MBytes, this does *not* imply that we are modeling a system in which the database size is so small. This size (also called the *server working set size*) actually reflects the amount of data that is accessed by the clients during the simulation. Furthermore, since our aim is to compare different isolation mechanisms, a more important parameter is the amount of contention in the workload rather than the size of the server working set. For this purpose, we model different workloads with varying levels of contention. In fact, our evaluation is somewhat pessimistic since most of the sensitivity analysis is performed using stressful workloads with high contention. In a database system, where clients access a much larger data set size, contention is expected to be lower than our stressful workloads. For realistic low-contention workloads, the results in the next chapter show that our optimistic mechanisms for strong isolation levels such as PL-2+ and serializability have negligible overheads over the PL-2 implementation; they have relatively low costs in the case of high contention workloads as well.

Another property of our simulation setup allows our results to hold for realistic systems: the relative sizes of different components are chosen according to the relative ratios observed in a realistic system. For example, the absolute size of the client cache in the simulation is not very important. The size of the client cache relative to the accessed data is more important because it has a significant impact on the number of fetches performed during a transaction, e.g., choosing a very small client cache relative to the accessed data can bias the results since a high number of capacity misses may dwarf the cost of the concurrency control mechanisms. Our decisions regarding sizes of various system components are further discussed in Section 6.1.

This chapter is organized as follows. Section 6.1 describes how we model various components of a distributed client-server database system and how these components are connected together. Section 6.2 describes the different types of workloads that we use for our simulation study and how accesses for a transaction are generated.

6.1 System Model

To simulate a distributed database system accurately, it is necessary to model all components that can impact performance in a significant manner. In this section, we describe various components of the system — servers, clients, network, disks, database, and the connectivity of clients with the servers. The default parameters and the range of values used for our sensitivity analysis experiments for these system components are shown in Figures 6-1 and 6-2.

Database Parameters		
Parameter	Default Value	Sensitivity analysis range
Object size	40 bytes	—
Page size	4 KB	—
Objects per page	100	—
Server working set	875 pages	875-1750 pages

Connectivity Parameters		
Parameter	Default Value	Sensitivity analysis range
Servers in cluster	2	—
Client in cluster	30	10 – 80
Total clusters in system	8	8 – 30
Client connections with servers	4	4 – 10
Preferred servers for each client	2	—
Non-preferred servers for each client	2	2 – 8

Client/Server Parameters		
Parameter	Default Value	Sensitivity analysis range
Client CPU speed (for DB processing)	600 MIPS	—
Server CPU speed (for DB processing)	1200 MIPS	—
Client cache size	612 pages	150 – 3500 pages
Server cache size	437 pages	—

Network Parameters		
LAN network bandwidth per cluster	120 Mbps	10 – 1024 Mbps
LAN extra delay	0 msec	0 – 20 msec
WAN network bandwidth per cluster	15 Mbps	10 – 60 Mbps
WAN extra delay	75 msec (avg)	40 – 300 msec

Figure 6-1: Summary of System Parameter Settings — I

6.1.1 Database

The database objects are spread over a number of servers. Each server contains 875 pages; we call these pages the *server working set* of the simulator run. The server working set does not represent the complete database stored at each server. Instead, it represents the fraction of the objects at a server that is accessed by the clients during the simulation. As discussed earlier, the actual size of various system parts is less important than the fact that the relative sizes of different components mirror their counterparts in real systems. Similarly, since we are comparing different isolation mechanisms, contention on the server working set has more importance than the absolute size of

Disk Parameters	
Rotational latency	3.0 msec
Average Seek	5.2 msec
Transfer Bandwidth	20 MB/s
Disks per server	4

Multistamp Parameters	
Server name size	4 bytes
Client name size	8 bytes
Threshold timestamp size	8 bytes
Timestamp increment size	4 bytes
Maximum invalidations per server in a multistamp	50% of total entries
LAN timeout period	1 sec
WAN timeout period	30 sec

CPU Processing Overheads	
Client/Server cache lookup	300 instr
Reading an object in client cache (sensitivity analysis: up to 4 million instr)	5000 instr
Writing an object in client cache (sensitivity analysis: up to 8 million instr)	10000 instr
Invalid set lookup for each object	10 instr
Multistamp operation cost	300 instr/entry
Fixed transaction restart cost	5000 instr
Transaction restart cost per modified object	400 instr
Disk setup cost	5000 instr
Fixed network cost	36000 instr
Variable network cost	43000 instr/KB

Figure 6-2: Summary of System Parameter Settings — II

the server working set.

We assume that objects are small and that they do not cross page boundaries. These are reasonable assumptions for object-oriented databases, e.g., the average size of objects accessed by most traversals of the OO7 benchmark [CDN93] in Thor [CALM97] is approximately 30 bytes. The size of each page is 4 KBytes and a page contains 40 objects of 100 bytes each.

6.1.2 Client-Server Connections

We chose to connect clients and servers in a manner that reflects how real systems are configured. Typically, data is distributed across servers to reduce bottlenecks in the system, for load balancing, to enable accessing data at nearby sites, etc. Thus, database users mostly access the servers that are located near them; this results in reducing the latency observed by clients and in localizing the network traffic. To capture this notion of locality, we partition clients and servers into different *clusters*: a server is more likely to be accessed by clients in the same cluster than by those in a different cluster. For example, employees of a department may access a set of database servers in their cluster (e.g., on their LAN) frequently and access other departments' servers occasionally.

In the simulation experiments, each cluster contains 2 servers and 30 clients. Each client is connected to 4 servers in the system of which two are *preferred* and the other two are *non-preferred*. A client's preferred servers are in its cluster whereas its non-preferred servers are chosen randomly from other clusters. Most of our experiments use a system with 8 clusters, i.e., 240 clients and 16 servers. This system size does not represent the complete distributed system; it simply represents a part of the real system that is under observation during the simulator run. As we show in the next chapter, our results also hold for a system with more clusters; we chose a 8-cluster system to keep the simulation tractable. To understand scalability effects, we varied the number of clients per cluster from 10 to 80; we also varied the number of clusters from 8 to 30.

In our default system, each server is connected to 30 clients that prefer it and 30 (on average) non-preferred clients. In the experiments for the default system, each server receives about 1 transaction from a non-preferred client for every 5 transactions received from preferred clients, i.e., non-preferred clients are not very idle from a server's point of view. Our workloads simulate a scenario with reasonably active clients and servers; we use such an environment to stress the performance of our mechanisms.

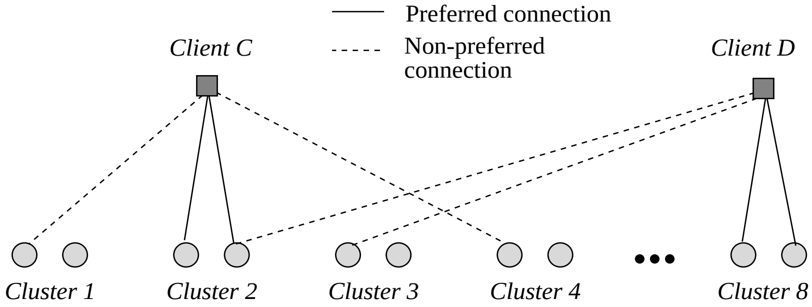


Figure 6-3: Client connections in a simulator run.

Many of our parameters have been chosen to affect the performance of our multistamp-based schemes negatively. We chose 2 preferred servers for a client instead of one (which is what is expected in real systems). As a client switches between the two preferred servers, the likelihood of stalls increases since the client can create inter-transaction dependencies/anti-dependencies across servers. Furthermore, the connectivity in our simulated environment is high; each client has two random connections to non-preferred servers in other clusters. Thus, a client can "spread" the multistamp from its cluster to other clusters via non-preferred servers. Increase in multistamp propagation can lead to aggressive truncation and unnecessary stalls. To study the effect of spreading information through preferred servers as well, we changed the topology such that each client has one preferred server in a cluster other than its own with varying probabilities. Similarly, we also increased the number of non-preferred servers from 2 to 8 to understand the impact on the multistamp-based schemes.

We modeled a default system size in which multistamps can become fairly large if aging or pruning is not used: they can reach a maximum size of 960 tuples (since there are 960 client-server connections). Thus, to ensure that network and server storage costs are low, multistamps must be truncated in our default system.

Figure 6-3 shows a typical setup of the system where client C is connected to two of its preferred servers in its cluster and its non-preferred servers are chosen from two other clusters (in general that is not necessary). In the figure, clients C and D share one server.

6.1.3 Client and Server

We model each client and server as a simple processor where events are scheduled using a nanosecond granularity clock. Various events are charged in terms of CPU instructions and then they are converted into time using a client's and server's speeds specified in MIPS, or million instructions per second. Events on a processor are executed using an event queue after the specified delay; if the server is heavily utilized, an event may be scheduled later than its specified time.

The client speed is chosen to be 600 MIPS and the server speed is 1200 MIPS. These speeds correspond to the amount of processing power that can be devoted by a client or server for database-related activities; in general, we expect client machines to be used for other applications while the transactions are executing.

The server working set in our system is half the size used in [Gru97, CFZ94]. Since the client mostly accesses objects from two preferred servers, the *effective* working set for a client is similar to the values chosen in the earlier studies; this ensures that contention is essentially the same as in earlier studies. In the next chapter, we show results for a system in which we double the server working set size; our basic performance results do not change significantly with a larger working set.

The client cache size is 612 pages; cache management is done using LRU. Although an object cache provides better performance than a page cache [CALM97], we have used a page cache in our simulator for simplicity; an object cache will help in increasing the effectiveness of the client cache but will not change our basic results (our results are essentially the same with bigger client caches).

We chose the client cache size relative to the size of the accessed data so that the concurrency control costs are not dwarfed by a high number of capacity misses; as stated earlier, the relative cache size is more important than the absolute cache size. The relative cache size is approximately 25% of the data accessed by a client in the simulation (this ratio is explained below). This percentage value was also chosen by other concurrency control studies [Gru97, CFZ94]. Furthermore, in the medium database experiments of the OO7 benchmark [CDN93], the client cache size is approximately 25% to 35% of the accessed pages (the percentage depends on how much space is used by a database implementation to store the benchmark objects on disk).

A client can potentially access 3500 pages from the database (875 from each server). So it might seem that our relative client cache size (612 pages) is smaller than what was used by earlier single-server concurrency control studies. However, in our multi-server system, approximately 85-90% of accesses go to objects stored at preferred servers, i.e., to 1750 pages. These pages mostly make up the client's working set and relatively few accesses go to the remaining 1750 pages at the non-preferred servers. Thus, our choice of client cache size between 437 and 875 pages (between 25% of 1750 and 25% of 3500), with a slight bias towards 437, is in line with earlier work.

Each server has a cache whose size was chosen so that the disk costs were not the dominant costs. The server cache size corresponds to 50% of the server's working set, i.e, 437 pages for a 875 page server working set; a similar ratio was chosen in earlier single-server concurrency control studies [Gru97, ZCF97]. Even though the client's cache size is larger than a single server's cache size, it is smaller than the combined cache size of two preferred servers (a client accesses two preferred servers instead of one in our setup). The issue of the server cache size is discussed more in Section 6.2.2.

6.1.4 Disk

The database objects are stored on multiple disks on the server and each database page is statically assigned to a disk independent of the workload. The disk is modeled as a shared FIFO queue on which operations are scheduled in the order they are initiated. Time spent in disk operations are charged to the queue; more contention on the disk can result in delays due to queuing effects. Disk delays are composed of a seek followed by rotational latency. We use an average seek of 5.2 msec and latency of 3.0 msec; the disk bandwidth is 20 MB/s. These performance values correspond to the Cheetah model of Seagate [Sea99] which is one of the faster disks available in the market. We use 4 disks per server to reduce contention on the disk. We also model installation of objects by approximating the MOB architecture presented in [Ghe95].

6.1.5 Network

Instead of modeling a particular type of network like Ethernet, ATM or FDDI, we have modeled a network in a more abstract manner. Each network message has a latency that is divided into four components: processor cost for sending the message, wire time, router delay, and processor cost for receiving the message. Processor costs at each end consist of a fixed number of instructions and a variable number of instructions according to the size of the message. We assume that each cluster of clients and servers is connected by a network that is modeled using a FIFO queue; the network bandwidth for each cluster multiplied by the message size is used to determine the time charged to the network queue, i.e., the network queue is utilized for this duration by a message. When a message is sent to a server, the network of that server's cluster is chosen for transmitting the message. When a message is sent to a client by a server, the latter's cluster network is used.

The network parameters for each cluster network are shown in Figure 6-2; these values have been obtained from a user-level messaging system called the U-Net [vEBBV95] running over a 155

Mbps ATM. We use a network bandwidth of 120 Mbps since this was the bandwidth achieved in the U-Net system for TCP. For our sensitivity analysis, we varied the network bandwidth for LAN environments from 10 Mbps to 1024 Mbps.

We have chosen a network bandwidth of 120 Mbps even though faster networks are available in the market. The reason is that current office environments use networks in the 100 to 200 Mbps range (or even lower bandwidth, e.g., 100 Mbps Ethernet). On the other hand, we have used one of the fastest disks available in the market for our disk parameters; the reason is that users upgrade disks much faster than networks since disks are more autonomous than networks.

We also performed experiments for a WAN environment whose parameters are shown along with the LAN parameters in Figure 6-2. We add a router delay of 75 msecs in the WAN case (a randomly chosen number between 50 and 100 msec); this delay is not added for the LAN case in our default system. The router delay is not charged to the network queue, i.e., it does not increase network utilization. We chose a bandwidth of 15 Mbps for each cluster in WANs. For our sensitivity analysis experiments, we varied the network bandwidth per cluster from 10 to 60 Mbps. We also varied the average router delays from 40 to 300 msecs for WANs and 0 to 20 msecs for LANs.

6.1.6 Multistamps and Other Parameters

Our multistamps contain server names of size 4 bytes and client names of size 8 bytes; these numbers have been derived from the Thor system [LAC$^+$96]. Each threshold timestamp entry in a multistamp uses 8 bytes and a timestamp entry uses 4 bytes; instead of maintaining the full timestamp for each client-server pair, we store it as an increment over the threshold. To ensure that most of the entries in a multistamp do not belong to one server, we allow at most 50% of the entries in a multistamp belong to the same server; if this limit is exceeded, we remove these entries and approximate the deleted entries with a server-stamp (as explained in Section 5.3.6).

We assume that the timeout period for “I’m alive” messages is 1 second in a LAN environment and 30 seconds in the WAN environment. Clock skews are not modeled since they are insignificant compared to the network delays and the timeout period; the Network Time Protocol [Mil92] can maintain clock skews of a few microseconds in LAN environments and a few milliseconds in WAN environments [Mil96].

6.1.7 CPU Processing Overheads

We now describe the CPU processing overheads that are charged at the server and the client for various operations. As stated earlier, our CPU processing overheads have been either derived from Gruber’s work or obtained by profiling our simulator (since our simulator code is similar to the code in Thor).

We charge 300 instructions to perform a cache lookup at the server, to generate an invalidation for a client by checking the cached set, and to set up an object for commit at the client [Gru97]; these costs correspond approximately to a few second level processor cache misses on modern machines. We charge 10 instructions for looking up each object in a client's invalid set. The cost of each multistamp operation is 300 instructions per multistamp entry; we obtained this result by profiling our simulator.

The costs for reading and writing an object at a client are 5000 and 10000 instructions respectively (i.e., 8μsec and 16μsec for our client machines). These times are based on the observation that a transaction operates on an object for a short period when it accesses the object; these times have been obtained from [Gru97]. As in the case of the server, the cost of looking up an object in the client cache is 300 instructions. When a transaction restarts, the client is charged 5000 instructions for resetting some data structures. The client is also charged 400 instructions per modified object (to revert the object to its original version); this corresponds to the memory copying costs on current machines. There is no think time spent at a client between transactions; as a result, transaction latency can be calculated using our throughput results.

The server is charged a fixed cost of 5000 instructions for setting up a disk operation, i.e., the read is initiated after 5000 instructions have executed on the server CPU; this is the same charge used in [Gru97, CFZ94]. For sending and receiving network messages, we charge a fixed message cost of 36000 instructions and a variable cost of 43000 instructions per KB. The time for sending/receiving messages on our client machines with these instruction costs is similar to the time taken in the U-Net system for slower processors; we did not scale these costs for faster machines since a large fraction of the work performed during a message send or receive does not scale with the processor speed [Ous90, ALBL91].

6.2 Workloads

In this section, we present different workloads used in our experimental study. We also describe how accesses for a transaction are generated. Two of our workloads, HOTREG and HICON, have been derived from Gruber's study. These workloads have moderate to high contention since they were designed to compare the performance of different concurrency control mechanisms. These workloads stress our multistamp-based mechanism since contention leads to more dependencies/anti-dependencies, bigger multistamps, more truncation and hence the possibility of lower performance due to more consistency stalls. Furthermore, higher contention enhances the difference between weak and strong isolation mechanisms; we were interested in understanding how high these differences can become.

We also designed a low contention workload called LOWCON to evaluate the performance of our schemes in a realistic environment where most clients access their private regions and occasionally access a shared region. We did not design a workload with access patterns similar to the TPC-C benchmark (used for online transaction processing [TPC92]) since TPC-C models lower contention compared to our moderate and high-contention workloads (and higher contention tend to stress our strong consistency mechanisms).

Common Workload Parameters		
Parameter	Default Value	Sensitivity analysis range
Single server transactions	80%	0% – 90%
Preferred server probability (for less than 3 servers)	90%	—
Read-only transactions	50%	0% – 90%
Read-only participants	50%	—
Minimum object accesses per transaction	180	90 – 450
Maximum object accesses per transaction	220	110 – 550
Object accesses per page	5 to 15	—
Probability of modifying a page	50%	—
Probability of modifying an object on a modifiable page	40%	—
Net object write probability (except for hot-shared in HOTREG)	20%	—
Restart change probability	20%	0% – 100%

Figure 6-4: Summary of Workload Parameter Settings

In our simulation study, we only focus on access patterns of an application and do not consider the fact that inconsistencies may be introduced in the database due to execution of update transactions at weak isolation levels; we assume that a programmer has determined that it is correct to execute the transactions at a particular level.

We first describe the workload model for generating transaction accesses and the parameters related to it. Then we describe the workloads used in our study. Figure 6-4 shows the parameter settings that are common across workloads.

6.2.1 Transaction Generation

Each client runs a single transaction at a time that consists of a sequence of reads and writes. These accesses are generated by a module called the *workload generator*. This generator can generate different shared access patterns with different levels of contention. Depending on the workload, the server working set consists of different regions and each region is accessed with a certain probability, e.g., in LOWCON, each server has a private region for each client and a shared region. Before we discuss particular workloads, we describe our strategy for generating a transaction's accesses; Figure 6-5 gives an overview of our strategy.

Step (1) involves determining the number of servers that will be accessed in the transaction. We first decide whether the transaction involves more than one server and then choose the number of servers. In all workloads, 80% of the transactions use a single server and 20% involve multiple servers. We chose a high percentage of multi-server transactions to stress our multistamp-based schemes (to spread invalidation information more quickly to different servers resulting in larger

Access Generation Steps

(1) Select the number of servers in the transaction.

(2) Determine how many servers are preferred and non-preferred, and select servers according to these constraints.

(3) Determine if transaction is read-only. If not, determine whether each server is a read-only participant or not.

(4) For each server, generate required number of clustered accesses:
   (a) Pick a region according to region access probabilities.
   (b) Pick a page in that region and select a few object accesses on that page.

(5) Repeat step (5) until all accesses have been generated for all servers.

Figure 6-5: Generation of accesses for a transaction

multistamps and more truncation). Benchmarks such as TPC-A and TPC-C [TPC92] have fewer than 10-15% multi-server transactions. In each multi-server transaction, we bias the selection towards a smaller number of servers since transactions involving a large number of servers (accompanied by a high modification rate of 20% that we chose for our experiments) are expected to be uncommon; in most of our experiments, 11.5% of our transactions use two servers and 8.5% use more than two servers. This is in line with what has been reported in the literature. For example, in TPC-C, multi-server transactions involve only two servers; other researchers have also reported two-server transactions to be common for distributed transactions [SBCM95]. In our sensitivity analysis experiments, we varied the percentage of multi-server transactions from 10% to 100%.

In step (2), we choose a preferred server with probability 90% for transactions with one or two servers. For a higher number of servers, we choose both preferred servers and then choose randomly from the remaining non-preferred servers. For experiments with our default system, this choice resulted in 18% of the chosen servers to be non-preferred.

Although real systems are dominated by read-only transactions [GW82, SBCM95], we only allow 50% of our transactions to be read-only; this selection is done in step (3). As before, the purpose of this choice is to stress our schemes. More modifications lead to more conflicts and tend to increase the performance degradation due to higher isolation level implementations. Furthermore, more contention also increases the stall rate in the multistamp-based schemes. For our sensitivity analysis experiments, we varied the percentage of read-only transactions from 0% to 90%. If the transaction is not read-only, each server can be a read-only participant with 50% probability, except that at least one server is chosen to be a read-write participant.

In step (4), we generate the accesses for each server. Each transaction accesses between 180 and 220 objects, i.e., 200 objects on average. For our sensitivity analysis experiments, we varied the number of object accesses from 100 to 500. In a multi-server transaction, the number of accesses is divided according to the probability of accessing that type of server, e.g., if one non-preferred and two preferred servers are chosen, 10% of the accesses go to the non-preferred server. This selection is done so that preferred servers have an overall access rate of 90%.

At each server, a region is selected according to the workload's characteristics and a page that has not been accessed in this transaction is randomly selected in the chosen region. We select between 5 and 15 objects on this page, i.e., on an average, 10 objects are chosen on a page resulting in approximately 20 pages being accessed in a transaction. More than one object is chosen on each page to model the notion of clustering. Clustering enhances database performance since related objects are read from the disk and sent on the network as a single group; researchers have described techniques for reorganizing placement of data periodically such that objects which are likely to accessed together are stored close to each other [CS89, GKM96, MK94, TN92].

Objects are also updated in a clustered manner in the following way. First, when a page is selected, we determine whether the page can be modified; this probability is chosen to be 50%. In the second step, each selected object on that page is updated with a 40% probability. Thus, the net probability of modifying an object is 20%; these probability values were also used in [Gru97]. We chose a high probability of object writes since it affects the performance of the higher isolation schemes negatively due to increased contention; it also results in larger multistamps (due to more invalidations) and hence more truncation.

For each experiment, we measure the performance of the system after 48000 transactions have been executed in the default system; for larger systems, we add 200 transactions per client to this total. To eliminate end-effects, we start our measurements after warming the system.

If a transaction aborts, it is restarted with the same set of accesses. However, when a restarted transaction accesses a newer object version (compared to its previous execution), its *remaining* accesses may change with a probability of 20%; this probability is called the *restart change probability* (in our simulation runs, we observed that approximately 30-35% of aborted transactions access different objects in a rerun after observing a new version of an object). This strategy models the notion that application may take a different path in the code after reading a different value of an object. In our setup, if accesses are changed, all further accesses are chosen independently of the previous run. In reality, new accesses are expected to have a significant overlap with the previous run. This strategy impacts the performance of higher isolation implementations since it increases the cost of aborts: when new accesses are chosen, it is likely that one or more objects in the new run are not cached and fetches need to be performed for them. In the next chapter, we show that the restart change probability is an important factor that increases the performance difference between isolation mechanisms. For our sensitivity analysis experiments, we varied the restart change probability from 0% to 100%.

6.2.2 Workload Descriptions

We now discuss the different workloads that are used in our experimental study. These workloads range from low contention to high contention. The low contention workload models behavior that is expected to be common whereas the moderate to high contention workloads are intended to stress the isolation mechanisms in a variety of ways. Figure 6-6 shows the parameter settings for different workloads.

Parameter	LOWCON	HOTREG	HICON
Private region size at preferred server per client	25 pages (750 total)	25 pages (750 total)	–
Cold-shared region size per server	850 pages	100 pages	750 pages
Server working set (from a client's perspective)	875 pages	875 pages	875 pages
Hot-shared region size per server	–	25 pages	125 pages
Hot-shared region access probability	–	10%	80%
Private region access probability	80%	80%	–
Cold-shared region access probability	20%	–	20%
Rest of database access probability	–	10%	–

Figure 6-6: Characteristics of different workloads

LOWCON

This workload models a realistic system with low contention (like those observed in [KS91, SS96]). Each client has a private region of 25 pages at each preferred server, i.e., each server stores 750 private pages. Each server has a *cold-shared* region of 850 pages. 80% of a client's accesses go to its private region; the rest go to the cold-shared regions at its connected servers. Thus, in this workload, a client can access 875 pages at a preferred server and 850 pages at a non-preferred server. Note that each server has 1600 pages but at most 875 pages are relevant from a client's perspective because private regions of other clients are not accessed by a client. Since most of the accesses of a client go to its private region (80%), there are few conflicts in this workload; most of the conflicts occur at the cold-shared region of the preferred servers (non-preferred servers are accessed with a low probability).

With a server cache size of 437 pages, we observed that the disk costs in LOWCON were the main reason for limiting the system throughput. We observed that various isolation mechanisms had a very similar performance (within 1%) for LOWCON with this choice of the server cache size. To enhance the difference between the isolation schemes, we chose an unrealistic cache size of 800 pages for our LOWCON experiments. Since the private region is not shared by clients, the server cache was able to cache most of the cold shared region resulting in very high server cache hit ratios of more than 90%. Another reason for changing the server cache size for LOWCON was that we wanted to evaluate the performance of our mechanisms in for a workload with few fetches per transaction in an environment where a resource other than the disk is a bottleneck; the next workload, HOTREG, has a similar client cache hit ratio as LOWCON, and the disk is the bottleneck resource for HOTREG.

HOTREG

As in LOWCON, each client has a private region of 25 pages at each preferred server; together these regions consume 750 pages. Each server also has a cold-shared region containing 100 pages and a small region of 25 pages (called the *hot-shared* region). Thus, the working set size at each server is 875 pages. 80% of a client's accesses go to its private region and 10% go to the hot-shared region; the remaining 10% of the accesses go to the rest of the database (or RDB) region consisting of all other pages at its connected servers, including private regions of other clients and the cold-shared region. Private regions exist at the preferred servers only and all reads/writes at non-preferred servers go to the hot-shared and or RDB region.

The hot-shared region might be the top of a naming hierarchy that is accessed often by clients. To capture the fact that such a directory is not modified often, only one in every ten transactions modifies the hot-shared region. But when a transaction does modify this region, half of the objects are updated in the selected pages, i.e., the probability of modifying a hot-shared object is 5%. Note that even at this probability value, the rate of conflicts due to the hot-shared region is high. Since 30 clients share two preferred servers and most accesses are to preferred servers, there are approximately 15 clients accessing objects from a server at any instant of time (there are more than 15 clients since 20% of the transactions are multi-server). At any given time, at least one transaction is modifying objects in the hot-shared region; these updates result in invalidations being generated for a large number of clients since the hot-shared region is almost completely cached at each preferred client. Thus, moderate to high contention on the hot-shared region causes large multistamps to be generated and stresses the performance of our multistamp-based schemes.

The HOTREG workload is used for evaluating the performance of the system when clients have skewed sharing patterns; it models the scenario when a client has higher affinity for its own objects than the rest of the database. The HOTREG workload has higher contention than LOWCON. In HOTREG, contention occurs over the small shared-region, and when an RDB access (20% probability over 825 pages) on a page conflicts with the access of the client that owns that page (80% probability over 25 pages). In LOWCON, a conflict occurs only when both clients access the cold-shared region of 850 pages at a server with 20% probability.

The HOTREG workload has the richest access patterns. It includes both uniform sharing (all clients access the hot-shared region uniformly) and skewed sharing (one client accesses its private region more frequently than other clients). These type of accesses can be expected in an object-oriented database system [Gru97].

HICON

HICON is a very unrealistic workload used in concurrency control studies to model high contention. There are two regions at each server — a hot-shared region with 125 pages that is accessed 80% of the time and a 750-page cold-shared region that is accessed 20% of the time. Both regions are shared uniformly among clients. High contention in this workload is primarily due to the hot-shared region. HICON is a workload that is intended to enhance the difference between weak and strong isolation implementations. It degrades the performance of strong isolation level implementations relatively more since these levels capture more conflicts and hence experience a higher number of aborts. HICON also stresses the multistamp-based schemes: since the preferred clients can cache a significant fraction of the hot-shared region, a large number of invalidations are generated by an update transaction. Thus, the HICON workload can result in large multistamps leading to aggressive truncation and a high consistency stall rate. Along with HOTREG, we also use HICON for our sensitivity analysis experiments.

Chapter 7

Performance Results

In this chapter, we present the results of a simulation study that evaluates the performance of our optimistic mechanisms in a distributed client-server system for cacheable transactions that access a few hundred objects during their execution. We use CLOCC for providing serializability, Weak-CLOCC for PL-2, and the multistamp-based schemes for intermediate isolation levels in our experiments. The reader should note that these schemes are efficient. Recall that CLOCC has been shown to be the best concurrency control mechanism for cacheable transactions in a client-server system [Gru97]; it outperforms the best locking mechanism [CFZ94] for a range of workloads and system parameters. Weak-CLOCC and the multistamp-based schemes share many advantages of CLOCC's basic cache coherency mechanism, e.g., piggybacked invalidation messages.

Our results show that the cost of strong consistency levels such as serializability is negligible for low-contention workloads. Even for moderate to high contention workloads, the performance degradation due to higher isolation levels is low. The low performance difference between serializability and lower isolation levels is primarily due to the fact that CLOCC prevents excessive wastage of work by aborting a transaction early during its execution and CLOCC has low restart costs for aborted transactions. Our results also show that multistamps impose a low performance penalty: multistamps with a size less than 100 bytes are sufficient to ensure that there are few consistency stalls. Thus, strong consistency guarantees such as PL-2+ can be provided to read-only and running transactions at very low costs.

Our results apply only to data-shipping client-server systems in which transactions run at clients. The CLOCC implementation takes advantage of this system architecture and offloads most of the work to clients. Thus, the extra work due to aborts occurs at clients, rather than at the servers, which are the scarce resource. If transactions ran at servers (i.e., pure function-shipping systems), the extra work due to aborts would slow down all clients rather than just the client whose transaction aborted, so that the cost of serializability may be higher in these systems. In fact, in such architectures, optimism may not be the best approach. However, our results may be applicable to hybrid systems in which the client performs most of the transaction's work and short queries are executed at a server, or to three-tier systems in which servers store persistent objects, clients run applications, and transactions are executed at shared "proxies".

Section 7.1 introduces a notation for describing systems that provide different isolation guarantees to update, read-only, and executing transactions. Section 7.2 presents a simple analytical model for comparing the performance of different isolation level implementations. In Section 7.3, we examine the relative cost of providing serializability for committed transactions over PL-2. We evaluate these costs by comparing CLOCC and the multistamp-based for PL-2+ with the Weak-CLOCC scheme. In this section, we also present our sensitivity analysis for a variety of parameters such as longer transactions, different mixes of read-only and update transactions, lower contention, etc. Section 7.4 presents the overheads of providing intermediate isolation guarantees such as PL-2+, PL-3U, EPL-2+ and EPL-3U for read-only and executing transactions using our multistamp-based schemes. Section 7.5 presents a detailed analysis of the cost imposed by our multistamp-based schemes. The cost of providing different levels of causality is presented in Appendix B

7.1 Interaction of Isolation Schemes for Different Types of Transactions

We use the notation *W/R/E* to indicate that the database is updated at level *W* and no inconsistencies below level *W* are introduced in the database state, no read-only transaction is committed if it has observed an inconsistency below level *R*, and no transaction observes an inconsistency below level *E* while it is executing. In a W/R/E system, the update transactions are committed using the optimistic technique presented in Chapter 5 for level W, read-only transactions are committed using the level R scheme, and transactions are executed using the scheme presented for level E. For example, 3/3/2 denotes a system where all committed (read-only and update) transactions are serializable and executing transactions do not observe dirty reads; thus, 3/3/2 denotes CLOCC.

In our simulator runs, all transactions of a particular type (update/read-only/executing) are provided the same isolation guarantees. For example, in the 3/2/2 system, all update transactions commit using the level PL-3 scheme. In a system where all update transactions commit at level *W*, it is guaranteed that inconsistencies disallowed by level *W* are not introduced in the committed database state. For example, in the 3/2/2 system, update transactions are serializable with respect to each other and a DSG constructed using only update transactions contains no cycles, i.e., the committed database state can be achieved by the serial execution of all update transactions.

For our simulation experiments, we implemented levels PL-2, PL-2L, PL-2+, PL-3U, PL-3 and their equivalent levels for executing transactions. However, as discussed in Chapter 3, we ensure that transactions are committed at an isolation level that is at least as strong as the level at which they execute. Similarly, we also ensure that update transactions are committed at a level that is at least as strong as the level chosen for committing read-only transactions; we make this assumption since it is not useful to provide read-only transactions with stronger guarantees when the database is being updated at a lower level. Thus, for a given combination *W/R/E*, level *W* is at least strong as *R*, and level *R* is at least as strong as level *E*. These requirements rule out certain combinations of isolation levels, e.g., PL-2 for update transactions, PL-2+ for read-only transactions, and EPL-2 for executing transactions (the 2/2+/2 system) is disallowed.

Apart from disallowing the combinations that violate the strength order discussed above, we also rule out certain combinations because of implementation considerations. In particular, level PL-3U is not provided along with levels EPL-2L or EPL-2+ since the multistamps for the latter levels only capture dependencies whereas the multistamps for PL-3U capture anti-dependencies also. Providing PL-3U in conjunction with EPL-2+ requires an additional overhead of maintaining and propagating two sets of multistamps in all parts of the system. Thus, systems such as 3/3U/2+ and 3/3U/2L are disallowed in our simulator implementation.

7.2 A Simple Model for Comparing Isolation Implementations

We use a simple analytical model to explain the performance difference between various isolation implementations. In our optimistic implementations, a transaction may abort a few times before it commits successfully. The final execution in which the transaction succeeds is called the *successful run* and all previous executions are called *aborted runs*; the combined runs for a transaction that commits successfully are called the transaction's *complete execution*.

A transaction's complete execution time, R is broken down into three components — time spent for fetching objects, time spent for performing reads and writes on locally cached copies of objects, and time spent in committing the transaction; these components are denoted by F, L, and C, respectively.

Some of the work performed during a transaction's aborted run may be wasted whereas part of it may help the successful run of this transaction or some future transaction executed by the same client. For example, if a transaction T fetches a page P and aborts, the fetch is useful if T's client C uses some object in P during the successful run of T or some future transaction at client C. To model this notion of wasted and useful work, we divide a transaction's complete execution time (R) into two parts — useful running time, R_u, and wasted running time, R_w. We also divide the time taken by for fetching, executing local operations, and committing transactions into their respective useful and wasted components, e.g., F is divided into F_u and F_w. Each useful part contributes to T_u and the wasted part contributes to T_w. We can write simple equations for these parameters:

$$R = F + C + L = R_u + R_w \tag{7.1}$$

$$F = F_u + F_w, \quad C = C_u + C_w, \quad L = L_u + L_w \tag{7.2}$$

$$R_u = F_u + C_u + L_u, \quad R_w = F_w + C_w + L_w \tag{7.3}$$

Thus, there are six basic components of a transaction's complete execution time: F_u, F_w, C_u, C_w, L_u, and L_w. With the help of these parameters, we will explain how different isolation implementations differ from each other. Let us see what each parameter means exactly and how it is computed in a simulation run.

Local Times, L_u and L_w: All local accesses performed during aborted runs are wasted and time spent for this purpose adds up to give L_w. The time, L_u, is computed by adding up all local accesses performed during a transaction's successful run.

Commit Times, C_u and C_w: Time spent to validate and commit a transaction's successful run is classified under C_u. The latency observed by a client for commits that fail validation is added up to give C_w.

Fetch Times, F_u and F_w: Classifying fetches as wasted or useful based on whether they were performed during an aborted or a successful run is not correct due to inter-transaction client caching; some fetches performed during a transaction's aborted run may be useful for the same or some later transaction's successful run. Let us understand how a fetch can be classified as useful or wasted. Fetches whose status (useful or wasted) is not known yet are called *unclassified fetches*.

Consider a fetch F by client D of a page P. We define F's *delta set* to be the set of objects that are retrieved by F but are not present in D's copy of P. We also define F's *delta-union set* to be the union of the delta sets of P's unclassified fetches (including F's delta set) that occurred before F at client D. Fetch F is classified as a useful fetch if some object in its delta-union set is used in the successful run of a transaction T$_i$ and T$_i$ commits:

1. Before P is removed from the cache, and
2. Before all objects in F's delta set are invalidated, and
3. Before another fetch of P occurs.

When a fetch by client D to page P is classified as useful, all other unclassified fetches to P by client D (at that instant of time) are classified as wasted.

Cases (a) and (b) declare fetch F to be wasted if all objects that were brought in by it are discarded before being used in a transaction's successful run. Case (c) says that fetch F is wasted if a later fetch brings in the same page, i.e., all objects that had been retrieved by F are refetched and the latter fetch supersedes the F's work. Thus, if there exists a sequence of unclassified fetches of page P and an object on P is used from one of the delta sets of these fetches in a successful transaction, only the last fetch is classified as useful.

In our simulator, a client obtains a complete page in every fetch reply. We could have used an approach in which a client just fetches objects on a page that are missing from the cache due to invalidations [CALM97, ACL$^+$97]; in that case, some fetches that are being classified as wasted will become useful. However, it would greatly complicate the analysis; our page-based approach simplifies the analysis.

Using the above basic parameters, we will show which components increase or decrease the performance difference among various isolation level implementations. Since a weak isolation level scheme imposes fewer constraints than a stronger isolation level, a transaction that has weaker isolation requirements may abort fewer times and hence perform less wasted work.

To keep our analysis simple, we classify time spent during stalls as wasted fetch time (F_w) for systems where stalls may occur during the execution of a transaction, e.g., in the 3/2+/2+ system. In systems where read-only transactions stall at commit time, we classify the stall time as wasted commit time (C_w). As we will see later, the component due to stalls is small since there are few stalls in the system; hence, this approximate classification does not affect our analysis significantly.

In many of our experiments, we observed that two systems providing different isolation guarantees performed the same amount of useful work (with other simulation parameters kept the same). Thus, the performance difference between the systems was largely determined by the extra wasted work done by the stronger isolation level implementation, e.g., for HOTREG and HICON in the 2/2/2 and 3/2/2 systems. Let us understand why the useful work done was the same for two systems providing different isolation level implementations by considering HICON and HOTREG in the 3/2/2 and 2/2/2 systems. First, the wasted work did not increase the utilization of some resource by a significant amount to affect the useful work done. For the HICON workload, the network is the bottleneck resource (it is saturated) in the 2/2/2 system. Additional fetches per transaction in the 3/2/2 system do not increase the network time since network saturation provides a negative feedback to the system, i.e., clients submit fetches at the same rate in the 3/2/2 system and the 2/2/2 system. A similar phenomenon occurs in the HOTREG workload where the disk is heavily utilized (around 80%); additional fetches did not increase the disk utilization significantly. Similarly, utilizations of other resources did not change for these workloads. Second, our algorithms for different isolation levels do not alter the client caching behavior; as a result, the same number of useful capacity misses and conflict misses occur in both systems. These two factors ensure that F$_u$ is similar for the two systems. Third, both systems commit read-only transactions at the client and all update transactions are validated at the server; furthermore, since the server CPU is not a bottleneck and the server is reasonably fast, the additional validation work in the 3/2/2 system is negligible. Thus, the useful commit work, C_u, is similar for the two systems. Finally, the work done for performing local accesses (L_u) remains the same since the same number of objects is accessed in the successful run in both implementations.

In some cases, a stronger isolation level implementation may perform more useful work to provide its consistency guarantees. For example, CLOCC (the 3/3/2 system) requires a commit message be sent to servers even for read-only transactions whereas the 3/2/2 system does not; thus, CLOCC will have a higher C_u component than the 3/2/2 system.

In our simulator runs, we measured the number of useful and wasted fetches, the number of aborted and successful commits, and the number of useful and wasted accesses. We also measured the average time taken to perform a fetch, a commit, and a local read or write. We used these times and numbers in equations 7.1 and 7.2 to compute the total time taken to execute a transaction. We compared this time with the measured latency of a transaction complete execution time. In all experiments, we observed that the computed sum was within 1% of the measured transaction execution time. This shows that the model and the simulator are self-consistent with each other and increases our confidence about the simulator's fidelity in modeling a real system.

7.3 Cost of Serializability

To understand the performance gains achieved by committing transactions at PL-2 instead of serializability, we use the 2/2/2, 3/2/2 and 3/3/2 systems. Performance is evaluated in terms of transaction throughput, which is defined as the number of successfully committed transactions per second in the database system. We also define a term called the *abort rate*, which denotes the number of aborts that occur in the system for every successful transaction commit; the abort rate is expressed as a percentage. We now give an overview of our results before presenting them in detail.

For a low contention workload such as LOWCON, the relative performance degradation of the 3/2/2 system compared to the 2/2/2 system was very low (less then 3%) across a wide variation of system parameters; the reason is that CLOCC has been designed to impose very low overheads when there are few conflicts in the system. For HOTREG and HICON, workloads with moderate to high contention, we observed that the performance difference varied from 7% to 10%. The performance degradation of the 3/2/2 system is due to the extra wasted fetches performed relative to the 2/2/2 system.

When read-only transactions are also serialized (i.e., we use CLOCC for all transactions), the performance degradation is higher for two reasons. First, since read-only transactions are also validated in the 3/3/2 system, they may abort (they never aborted in the 2/2/2 or the 3/2/2 systems) resulting in more wasted work. Second, an extra roundtrip message delay is added to commit read-only transactions; this factor has a larger impact in WANs since message latencies are high in those environments. For LOWCON, the performance degradation for the 3/3/2 system is 6% in LANs and 14% in WANs relative to the 3/2/2 system. For HOTREG and HICON, the penalty of the 3/3/2 system varies between 9% to 15% relative to the 3/2/2 system.

A sensitivity analysis on different parameters shows that serializability penalizes performance relative to PL-2 when the following two conditions are satisfied:

- Contention is high, *and*

- The cost of restarting a transaction is high

Parameters that increase contention along with restart costs result in increasing the performance difference, e.g., increasing the transaction length, increasing the restart change probability, increasing the amount of work done per object access, decreasing the percentage of read-only transactions, etc. For some of our experiments, we observed that contention was high but the relative performance difference between the PL-3 and PL-2 mechanisms was low. The reason is that CLOCC has been designed such that the restart costs are low for cacheable transactions.

It is important to understand that our workloads (HICON and HOTREG) and our system parameters have been setup to stress the higher consistency mechanisms. These workloads have been derived from concurrency control studies in the past and need not correspond to real applications; in fact, we expect real applications to have less contention resulting in smaller performance differences between the 2/2/2 and 3/2/2 systems.

The performance win for the 2/2/2 system compared to the 3/2/2 system comes with the cost that there is a higher likelihood of corrupting the database system, i.e., a transaction is not aborted when the 3/2/2 system would have aborted it. Hence, it is desirable that update transaction are not committed below PL-3. For the experiments in this chapter beyond (and including) Section 7.4, we provide PL-3 for update transactions using CLOCC.

The rest of the section is organized as follows. In Section 7.3.1, we present the results for our default system. In Section 7.3.2, we compare the performance of the 2/2/2 and 3/2/2 system across a wide range of system parameters. In Section 7.3.3, we compare the performance of the 2+/2/2 system to that of the 2/2/2 and 3/2/2 systems.

7.3.1 Basic Results

Before presenting the results, let us understand some of the important performance characteristics of our three workloads. For the default system parameters, we observed that the disk is the main bottleneck for the HOTREG workload in LAN environments with a utilization of more than 80%; in LOWCON and HICON, the network is saturated (as discussed in the previous chapter, LOWCON would have been disk-bound if the server cache size was smaller). In WAN environments, the extra delay of 50-100 msec results in increasing transaction latency substantially. In LOWCON and HOTREG, each transaction performs 3.7 to 4.4 fetches per transaction whereas there are 19 to 23 fetches per transaction for HICON. In general, if the workload has a high abort rate, we expect the number of fetches to be high due to conflict misses (objects invalidated by other clients); in HICON, more than two-thirds of the fetches are due to conflict misses. For all workloads in these experiments, the time spent in performing page fetches (F) forms a significant fraction of a transaction's execution latency.

Figure 7-1 shows the transaction throughput for different workloads and systems; it also shows

System	LAN environment			WAN environment		
	LOWCON	HOTREG	HICON	LOWCON	HOTREG	HICON
2/2/2	7443	4017	1490	357.2	350.9	79.1
3/2/2	7344 (1%)	3647 (9%)	1355 (9%)	352.9 (1%)	325.1 (7%)	71.4 (10%)
3/3/2	7020 (6%)	3304 (18%)	1227 (18%)	307.6 (14%)	276.7 (21%)	62.9 (20%)

Figure 7-1: Throughput (transactions per sec) variation with and without serializability guarantees

the throughput degradation of each workload/system combination relative to the throughput of the corresponding 2/2/2 system.

LOWCON workload

Figure 7-1 shows that the cost of providing serializability to update transactions in the LOWCON workload is negligible in LAN and WAN environments. The reason is that LOWCON has a low abort rate (see Figure 7-2) and CLOCC has been designed to be inexpensive when there are few conflicts in the system; the cache coherence and validation mechanisms in CLOCC are efficient and have low communication, CPU, and disk costs. With a low abort rate, there is very little wasted work done in LOWCON (see Figure 7-3). The breakdown of a transaction's execution time shown in Figure 7-4 demonstrates that the extra wasted work component for the 3/2/2 system compared to the 2/2/2 system is very small for LOWCON.

When serializability is provided to read-only transactions (the 3/3/2 system), the performance degradation increases to 6% (in LANs) due to the extra roundtrip message for committing these transactions (an increase in the C_u component). Since the network is saturated in LOWCON, these extra messages do have an impact on increasing the transaction latency. However, these commit messages have another interesting effect. If we consider a time interval *t* in both systems, the 3/3/2 system sends more commit messages in time *t* than the 3/2/2 system. Since the 3/3/2 system sends fewer fetch messages in time *t*, it has lower network bandwidth requirements than the 3/2/2 system resulting in lower network queuing delays. As a result, there is a minor decrease in useful fetch time (F_u) in LOWCON for the 3/3/2 system in LANs (see Figure 7-4). If message latency had remained

System	LAN environment			WAN environment		
	LOWCON	HOTREG	HICON	LOWCON	HOTREG	HICON
2/2/2	0.7%	4.6%	25.6%	0.7%	4.2%	26.3%
3/2/2	2.9%	19.7%	105.1%	2.9%	21.1%	111.1%
3/3/2	5.3%	33.1%	182.4%	5.5%	34.4%	191.3%

Figure 7-2: Abort rate with and without serializability guarantees

System	**LAN environment**			**WAN environment**		
	LOWCON	HOTREG	HICON	LOWCON	HOTREG	HICON
2/2/2	1.0%	6.4%	6.9%	1.1%	4.4%	7.0%
3/2/2	2.1%	15.3%	17.5%	2.2%	11.9%	17.6%
3/3/2	3.3%	24.5%	28.6%	3.3%	17.8%	28.6%

Figure 7-3: Ratio of wasted work to useful work (T_w/T_u) for different systems

the same in the 3/3/2 system, the impact of the extra roundtrip message would have been higher. In fact, this effect is seen in WAN environments where message latencies remain the same; the extra message results in a larger performance degradation compared to LANs. As we will see later, a programmer may want to use levels such as PL-2+ or PL-3U for read-only transactions since their implementations avoid the commit roundtrip message and have a lower performance degradation than serializability.

These results show that strong consistency guarantees such as serializability can be provided for low-contention cacheable transactions in a client-server system without any significant costs. This is an important result since many applications exhibit low-contention and such workloads are one of the main scenarios recommended for using weaker isolation levels; this recommendation is based on the assumption that a higher isolation level mechanism imposes unnecessary performance penalties for such an application. However, since CLOCC (which is used for update transactions in the 3/2/2 system) has low overheads, the performance degradation in LOWCON due to serializability is not high in our experiments for the 3/2/2 and 3/3/2 systems.

HOTREG and HICON workloads

For HOTREG and HICON, the performance degradation of the 3/2/2 and 3/3/2 systems is more than in LOWCON because the abort rates are much higher for these workloads (see Figure 7-2). However, a large increase in the abort rate does not result in a corresponding increase in performance penalty. The main reasons are that CLOCC has low restart costs and invalidation messages prevent excessive wastage of work by aborting a transaction early during its execution. In HOTREG, more than 80% of the aborts occur at client machines and, for HICON, client-aborts account for 90% of the total in LAN environments; a majority of aborts occur at client machines in WAN environments as well. In our client-server system, the cost of re-executing a transaction at a client is not very expensive relative to the transaction execution time since the client processor is relatively fast and is not heavily utilized; in centralized systems where a server executes most of the transaction code, these aborts can be much more expensive.

Apart from invalidation messages, other mechanisms in CLOCC also help in reducing the restart costs. For example, maintaining an undo log for modified objects at the client reduces the number of

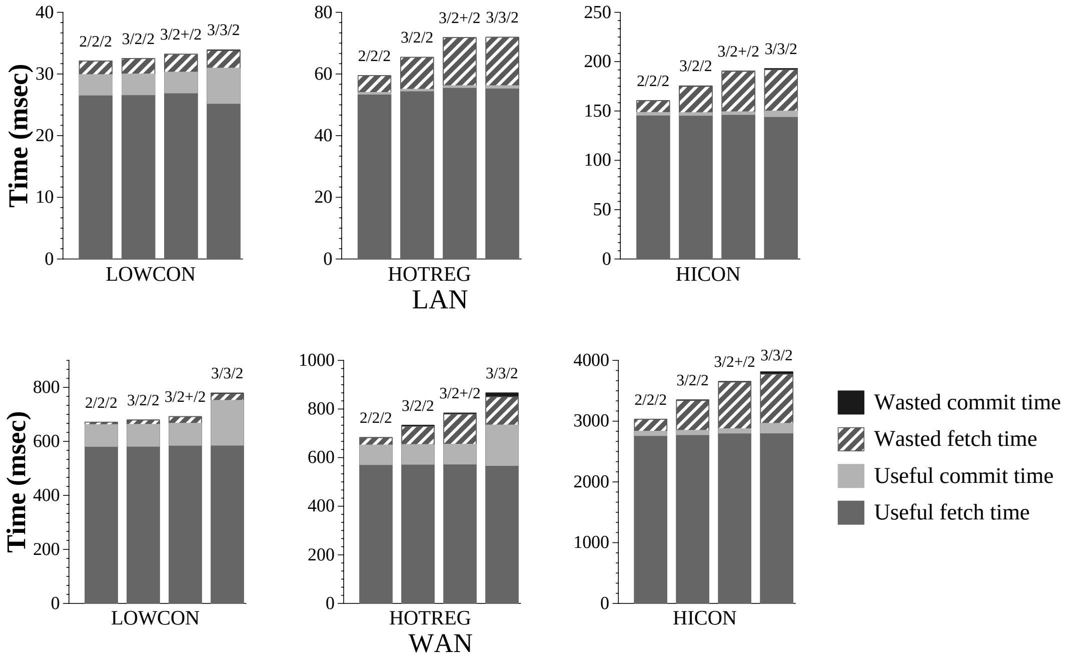


Figure 7-4: Breakdown of a transaction's execution time (L is negligible in these experiments)

fetch messages that need to be performed after a transaction aborts; earlier optimistic mechanisms did not incorporate such techniques and hence incurred excessive abort costs. In HOTREG, if this log were not maintained, there could have been 2 or 3 unnecessary fetches per abort; these extra fetches can have a significant impact on the throughput compared to the 2/2/2 system (recall that there are approximately 4 fetches per transaction for HOTREG).

Another reason why aborts do not hurt performance significantly is that not all work done in an aborted run is wasted, especially due to fetches (see Section 7.2). Figure 7-3 shows that the ratio of wasted to useful work does not increase substantially even for very high abort rates, e.g., for LAN environments, the abort rate for HICON is more than 5 times that for HOTREG in the 3/2/2 system but the fraction of wasted work performed in both workloads is similar and hence there is a similar performance degradation.

The increase in transaction execution time for the 3/2/2 system compared to the 2/2/2 is largely explained by the increase in the time spent in performing wasted fetches, F_w (see Figure 7-4). There are two factors that cause F_w to increase:

- Refetching invalidated objects: In the 2/2/2 system, if a client receives an invalidation message for an object that has been read (not modified) by its current transaction T, T is allowed to continue running; however, in the 3/2/2 system, T is aborted. When the client restarts and accesses the same object, it must refetch the object from the relevant server.

- Fetches due to changed accesses after abort: Due to a 20% restart change probability for each new version read in a rerun, approximately 30-35% of aborted transactions access different objects in their new run; this causes some fetches performed during aborted runs to be useless.

For the 3/3/2 system, the transaction execution time is higher than the 3/2/2 system due to a further increase in F_w caused by extra aborts of read-only transactions and due to an increase in C_u caused by the extra commit roundtrip message sent for read-only transactions (see Figure 7-4). The impact of this roundtrip is more prominent in WANs where message delays are more significant. For example, in HOTREG, the C_u component relative to the total execution time (R_u) is 12% in the 3/2/2 system and 20% for the 3/3/2 system (for HICON, these numbers are 2.5% and 4.5 respectively).

7.3.2 Sensitivity Analysis

We now present a sensitivity analysis to understand how different system parameters affect the throughput difference due to serializability guarantees. We focus on the performance difference between the 3/2/2 and 2/2/2 systems because the 3/2/2 system is the cheapest reasonable system which ensures that updates do not corrupt the database state. We report only the results for HOTREG and HICON workloads since they stress the system more than LOWCON.

Before we present a detailed sensitivity analysis, here is a brief synopsis of the results:

- Increasing the restart change probability causes more fetches when a transaction aborts and hence increases the cost of re-executing a transaction. Thus, the performance penalty of serializability is higher for a high restart change probability; most of the wasted work is due to wasted fetches.

- Increasing the percentage of read-only transactions in the workload reduces contention and hence reduces the throughput difference between the 3/2/2 and 2/2/2 systems.

- If more objects are accessed per transaction, contention is increased along with the cost of re-executing a transaction. Thus, the performance penalty of serializability increases with transaction length.

- If the time spent per object is increased, the performance difference increases because the cost of restarting a transaction is higher (more time has to be spent to access the same number of objects in the rerun). In such cases, most of the wasted work is due to the L_w component, i.e., wasted local time at clients.

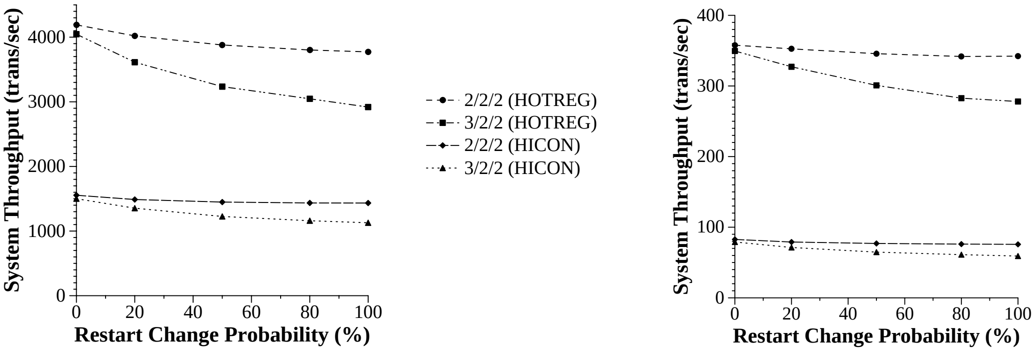
- Varying the cache sizes does not change the performance difference significantly because the reduction/increase in the number of fetches due to different client cache sizes is similar in the 2/2/2 and 3/2/2 systems.

- Increasing the number of clients increases contention for HICON resulting in a higher performance difference. For HOTREG, the contention remains same and the performance difference between the 3/2/2 and the 2/2/2 systems does not change.

- Variations in network delays, network bandwidth, and processor speeds have a low impact on the throughput difference between the 2/2/2 and 3/2/2 systems.

Restart Change Probability

CLOCC has been designed such that transaction re-execution is cheap: when a transaction reruns after an abort, most of the objects accessed in the previous run are cached at the client (except the objects that were invalidated). However, since we have chosen a restart change probability of 20%, it is possible that an aborted transaction accesses a different set of objects than its previous run after reading a newer version of an object. Since it is not necessary that the objects selected in the new run are cached, the client may perform extra fetches that would not have occurred if the transaction had not aborted.



LAN

WAN

Figure 7-5: Variation of the restart change probability

To understand the impact of the restart change probability, we varied it from 0% to 100%. The results in Figure 7-5 show that the performance difference between the 2/2/2 and 3/2/2 systems increases as this probability is increased. With a high probability value, the number of wasted fetches increases because a transaction is more likely to change its accesses when it is restarted; in Figure 7-6, we can see that wasted fetches account for most of the increase in transaction latency. The useful fetch time for HOTREG in the 3/2/2 system with a 100% restart change probability (LANs) increases because there are more disk reads than the 2/2/2 system. Since the disk is a bottleneck resource for this workload and environment, increasing its use results in more queuing delays and the time per fetch increases slightly.

We also observed that as the restart change probability is increased, the abort rate increases for both systems, e.g., for HICON, it increased from 95% to 120%. The reason is that wasted fetches

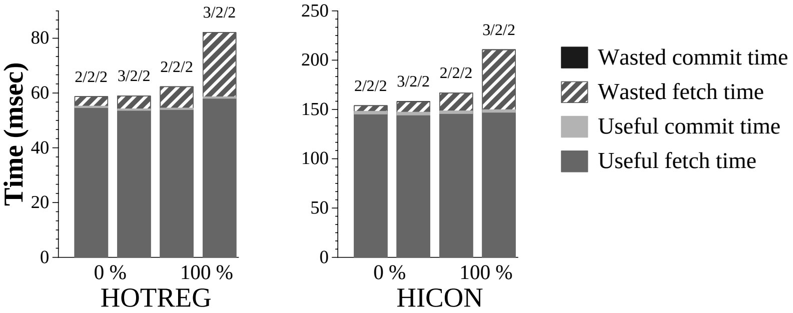


Figure 7-6: Transaction breakdown for 0% and 100% restart change probability in LANs

increase the duration of an aborted transaction T_i relative to successful transactions making T_i more vulnerable to be aborted again. The abort rate increase due to a high restart change probability is higher for the 3/2/2 system since it has more wasted fetches than the 2/2/2 system. This increase in abort rate feeds back into the system and results in raising the number of wasted fetches even more. The process of higher abort rates raising the number of wasted fetches and vice versa does not continue forever since the contribution by both factors keeps diminishing in each feedback cycle. In general, a transaction T_i aborts another transaction only after T_i commits. If this process continued forever, no client transaction would be able to commit. At this point, *some* transaction would be able commit and then progress could be made, i.e., there is a negative feedback due to aborting of transactions. (Two multi-server transactions may abort each other during the two-phase commit protocol due to different ordering at servers. However, the window of vulnerability is very small for such cases. Furthermore, it is very likely that these transactions do not rerun and prepare again in a lock-step manner, e.g., a slight difference in network delays for the two transactions during their reruns will allow one of them to be ordered before the other transaction and commit successfully.)

As discussed earlier, the performance difference between systems that commit transactions at different levels increases whenever the cost of aborts is high and the abort rate is significant. A higher restart change probability value raises the abort costs and hence increases the performance difference between the 2/2/2 and 3/2/2 systems for the HOTREG and HICON workloads.

Percentage of Read-only Transactions

Our workloads contain a mix of 50% read-only transactions. In many applications and environments, a larger percentage of transactions may be read-only. In Figure 7-7, we show how the throughput varies with the percentage of read-only transactions. Since read-only transactions are committed at PL-2 in both systems and these transactions result in lower amount of contention (due to fewer modifications), the throughput difference reduces as the percentage of these transactions is increased, e.g., for the HICON workload, the performance difference between the two systems reduces from 20% to 3% when the percentage of read-only transactions is increased from 0% to 75%; a similar difference is observed in the HOTREG workload as well.

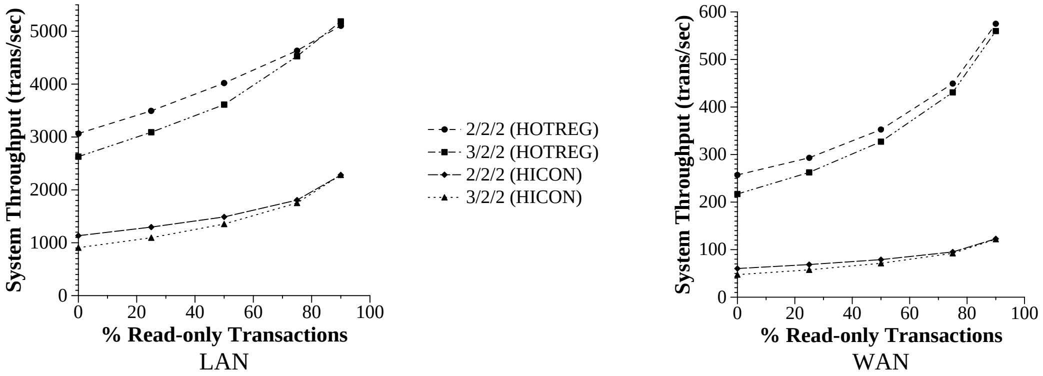


Figure 7-7: Transaction throughput (transaction per sec) as the percentage of read-only transactions is varied

Transaction Length

We also ran an experiment in which the number of objects accessed by a transaction was varied from 100 to 500 (the minimum and maximum accesses were 10% below and above these values respectively, e.g., 450 to 550 accesses for the 500 access case). In order to run this experiment, we doubled the size of each region by a factor of two, i.e., the private region at each server is 50 pages instead of 25, the working set is 1750 pages, and so on. This change was needed because it was not possible to run the HOTREG workload with more object accesses: the private region (25 pages) at each preferred server in our default system was not large enough to allow 80% of accesses to be private region accesses. The increased region size reduces data contention and the performance difference between the 2/2/2 and 3/2/2 systems decreases to 3-5% instead of 7-10% in our original system.

The results of this experiment for HOTREG and HICON are shown in Figure 7-8. Increasing the number of objects accessed by a transaction results in higher contention [GR93] and hence more aborts, e.g., the abort rate in HOTREG goes up from 3% to 60% in LAN environments for the 3/2/2 system. Furthermore, a longer transaction can abort at a later point during its execution and a client has to read/write more objects to rerun the transaction to the same point in the new run. Thus, the performance difference between the 2/2/2 and 3/2/2 systems increases from 2% with 100 object accesses to 16% at 500 object accesses for HOTREG; like previous experiments, the increase in transaction execution time is due to the extra time spent by a client performing wasted fetches. This result is in line with our arguments given earlier, i.e., increased abort costs along with higher abort rates result in a performance degradation of the 3/2/2 system.

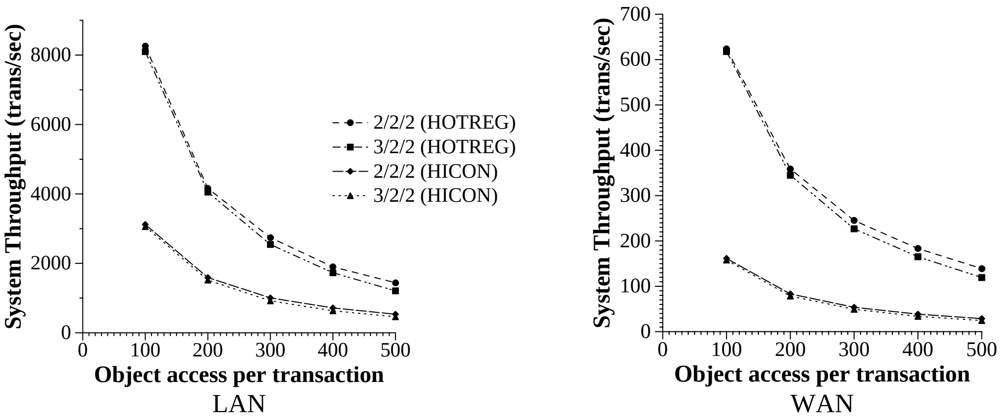


Figure 7-8: Variation in number of object accesses per transaction

Local Work at Clients

To better understand the effect of higher abort costs, we ran another experiment in which a client spends more CPU cycles on each object accessed in a transaction (the time to modify an object is twice the time for reading an object). When a transaction aborts, the amount of work that has to be redone is higher, i.e., abort costs are increased.

Figure 7-9 shows the variation of throughput as the work performed per object is increased; it also shows the breakdown of a transaction's execution time in the HICON and HOTREG workloads for the 0% and 100% cases (we do not show the commit times since they are negligible compared to the other components). In LAN environments, as the work cycles are increased, the transaction's execution time is dominated by the local work component (L) and local wasted work (L_w) constitutes a large fraction of the total wasted work. Thus, the performance difference between the 2/2/2 and the 3/2/2 systems is essentially due to L_w as the CPU work per object is increased (see Figure 7-9). In HOTREG, the abort rate (11-20%) is not sufficiently high to increase the performance difference between the 2/2/2 and the 3/2/2 systems by a large amount. In HICON, a higher abort rate has more significant impact and the throughput difference between the two systems increases.

An interesting phenomenon occurs when the client processor becomes the bottleneck. After this point, the local work component becomes so dominant that the performance difference between the 2/2/2 and the 3/2/2 systems decreases or remains the same. Furthermore, the abort rate increases before the client CPU becomes heavily utilized and then *decreases* after this point. The abort rate is determined by the relative rate at which objects are modified to the rate at which their invalidations become known to other clients. After the client CPU bottleneck point, information is propagated relatively faster to clients: clients modify objects at the same rate but invalidations come *relatively* faster to clients since more "I'm alive messages" are sent by servers (more timeouts at the servers).

Another interesting observation about both workloads in LANs environments is that with more

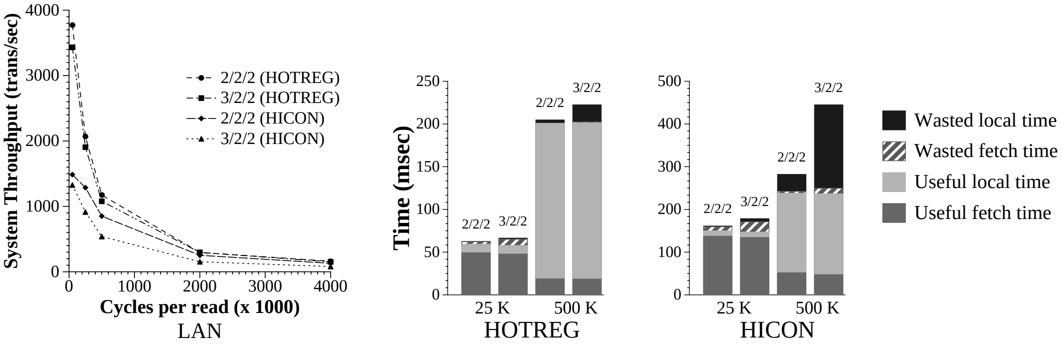


Figure 7-9: Variation in the amount of work done for every object access (LANs)

time spent on local work, the network and disk utilizations are reduced and the time to perform a fetch decreases. As a result, the total useful fetch time spent by a transaction (F_u) reduces, e.g., in Figure 7-9, the useful fetch time for the 500K cycles case is approximately 60% lower than the 25K cycles case.

For WAN environments, the performance difference is largely unaffected until the CPU processing costs become comparable with the work done by a transaction due to fetches. The performance variations are similar to the LAN case after this point, i.e., HICON's performance degradation increases whereas the throughput difference does not vary much for HOTREG.

Clients per Cluster

We varied the number of clients per cluster from 10 to 60; the results are shown in Figure 7-10. For HICON, increasing the number of clients raised the contention level substantially since we did not change the size of the working set at the servers. As a result, the abort rate increased from 30% to 240% causing more wasted fetches and the performance difference increased from 4% to 18%. In HOTREG, we added a private region for every client and the contention level did not change substantially. The performance degradation of the 3/2/2 system relative to the 2/2/2 system varied between 7% and 10%.

Client Caches

To understand the impact of client cache sizes, we varied the size of the client cache from 150 to 3500 pages. As the cache size increases, capacity misses decrease but most of these are replaced by conflict misses in HICON and HOTREG. Thus, there is a small decrease in the number of fetches as the cache size is increased and, more importantly, the reduction in fetches is similar for the 2/2/2 and 3/2/2 systems. As a result, the performance difference between these systems remains the same across a range of client cache sizes.

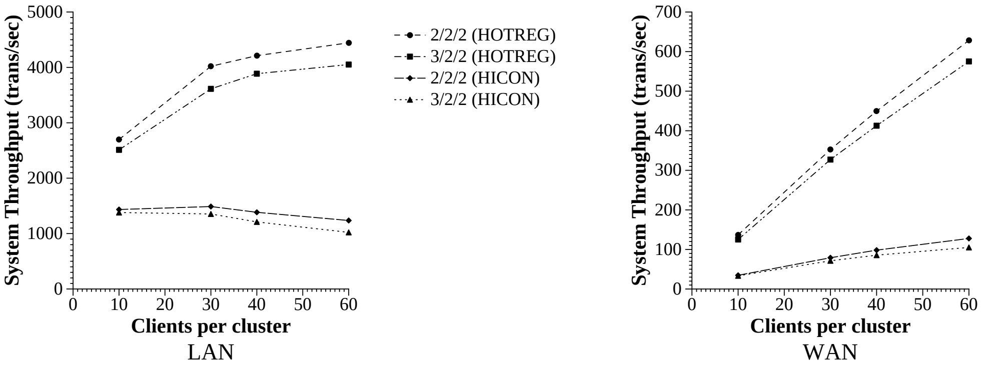


Figure 7-10: Throughput changes with variation in number of clients per cluster

Processor Speeds

We also varied the processor speeds and observed that the change in the performance difference between the 3/2/2 and 2/2/2 systems was not significant for client speeds beyond 300 MIPS (recall that our default system uses a client CPU speed of 600 MIPS). For a 200 MIPS server processor (100 MIPS client processor), we observed that the performance difference was higher (13% instead of 9%) in LAN environments for the HICON workload. The reason is that the server CPU becomes heavily utilized (due to network message handling) and hence there is an increase in the time to fetch a page or commit a transaction. Thus, apart from the increase in the wasted fetch time (F_w), the useful fetch time (F_u) increases as well and results in a higher performance degradation.

Network Variations

We varied the network parameters in two ways. We changed the extra router delays from 0 to 20 msec in LANs and from 40 to 300 msec in WANs; we also varied the network bandwidth allocated for each cluster. With extra router delays, a transaction spends more time in the network resulting in a lower load on resources such as the disk in HOTREG and the network queue in HICON. However, since these resources were utilized to a similar degree in the 2/2/2 and the 3/2/2 systems, reduced utilization helped both systems in a similar manner. Thus, variations in network delays did not change the performance difference significantly.

For similar reasons, variations in network bandwidth did not change the performance difference between the two systems. The utilization of some resources (e.g., disk) may increase at higher bandwidths due to faster communication but the increase is similar for both systems. Furthermore, for workloads such as HICON in which the network is the bottleneck, a faster network helps both systems. Hence, the relative performance remains the same with an increase or decrease in network bandwidth.

7.3.3 Cost of PL-2+ for Update Transactions

To evaluate the cost of PL-2+ for update transactions, we ran experiments with the 2+/2/2 system and compared them with the 2/2/2 and 3/2/2 systems. We observed that the cost of the 2+/2/2 system is similar to that of the 3/2/2 system. Since PL-2+ offers weaker guarantees than PL-3 (PL-2+ can result in inconsistent updates to the database), it is not worthwhile to support PL-2+ for update transactions; the 3/2/2 system should be used instead.

7.4 Cost of Intermediate Isolation Levels

In this section, we evaluate the cost of providing EPL-2+ and PL-2+ using our multistamp-based mechanisms. To evaluate the cost of providing PL-2+ to read-only transactions, we compare the performance of the 3/2/2 and 3/2+/2 systems. To determine the overheads of EPL-2+, we compare the 3/2+/2 system with the 3/2+/2+ system. We also evaluate the cost of EPL-2+ by comparing the performance of CLOCC (3/3/2) and the 3/3/2+ system; the 3/3/2+ system provides strong guarantees (serializability) to all committed transactions and makes it easier for an application programmer to reason about the code (since the transaction always observes a consistent database state).

The results show that cost of providing PL-2+ in the case of LOWCON are negligible; for HICON and HOTSPOT, the 3/2+/2 system has less than 10% throughput degradation compared to the 3/2/2 system. Our results also show that the cost of multistamps and consistency stalls is low, i.e., it is not extensive to provide strong consistency guarantees such as EPL-2+ or EPL-3U to executing transactions.

7.4.1 Overheads of PL-2+ and EPL-2+

Figure 7-11 shows the system throughput and performance loss of different systems compared to the 3/2/2 system. Similar to the results shown in the last section for serializability, the performance penalty of providing PL-2+ relative to PL-2 in LOWCON is negligible. The transaction breakdown graph in Figure 7-4 shows that the extra work due to wasted fetches is negligible in the 3/2+/2 and 3/2+/2+ systems for LOWCON.

For HICON and HOTREG, providing PL-2+ to read-only transactions (3/2+/2 system) results in a significant increase in the number of data conflicts. At PL-2, read-only transactions never abort whereas they can be aborted at PL-2+. In LAN environments, the abort rate for HOTREG increases from 20% to 32% whereas for HICON, it increases from 105% to 174%; the rise is primarily due to the aborts of read-only transactions. These extra aborts and a 20% restart change probability cause more fetches and hence result in a throughput drop from the 3/2/2 system to the 3/2+/2 system. Figure 7-4 shows that wasted fetches account for the increase in transaction latency for the 3/2+/2 system.

Apart from extra aborts, providing PL-2+ has additional costs due to multistamp manipulation

System	**LAN environment**			**WAN environment**		
	LOWCON	HOTREG	HICON	LOWCON	HOTREG	HICON
3/2/2	7344	3647	1355	352.9	325.1	71.4
3/2+/2	7177 (2 %)	3308 (9 %)	1243 (8 %)	346.3 (2 %)	304.4 (6 %)	65.5 (8 %)
3/2+/2+	7173 (2 %)	3335 (9 %)	1243 (8 %)	344.1 (2 %)	301.3 (7 %)	64.8 (9 %)
3/3/2	7020 (4 %)	3304 (9 %)	1227 (9 %)	307.6 (13%)	276.7 (15%)	62.9 (12%)
3/3/2+	7072 (4 %)	3352 (8 %)	1231 (9 %)	310.8 (12%)	271.1 (16%)	62.2 (13%)

Figure 7-11: System throughput (transactions per sec) and performance penalty (relative to the 3/2/2 system) for PL-2+ and EPL-2+.

and consistency stalls when read-only transactions commit. Our multistamp truncation technique ensures that the multistamp size is less than 100 bytes; hence, the cost of processing and propagating them on the network is low. Furthermore, there are few consistency stalls since a client is usually up-to-date with respect to all participants when a read-only transaction commits; during a transaction's execution, the client performs some fetches from different servers and this enables the client to become reasonably up-to-date with respect to those servers, e.g., there are an approximately 4 fetches per transaction in HOTREG. Thus, we observed that fewer than 3% read-only transactions require communication with servers at commit time.

The 3/2+/2 system has similar or better performance than the 3/3/2 system because the client is able to avoid a commit message roundtrip for read-only transactions in these systems whereas CLOCC requires this communication. Figure 7-4 shows that the extra overheads due to consistency stalls for committing read-only transactions is negligible, i.e., the useful commit time (C_u) is similar in the 3/2/2 and 3/2+/2 systems.

Varying Percentage of Read-only Transactions

Since the extra cost in the 3/2+/2 system is imposed by aborts of read-only transactions, we evaluated the sensitivity of the system throughput with respect to the percentage of read-only transactions in the workload.

When there are no read-only transactions, both systems are identical because all update transactions are provided the same guarantees. With a higher percentage of read-only transactions in the workload mix, contention drops significantly; as a result, there are fewer conflict misses at the clients and the system throughput increases. Furthermore, a lower abort rate results in fewer wasted fetches (due to the 20% restart change probability). Thus, lower contention tends to reduce the performance difference between the 3/2+/2 and 3/2/2 systems.

However, a second opposing factor causes an increase in the throughput difference. As the percentage of read-only transactions is increased, a larger percentage of transactions that abort in

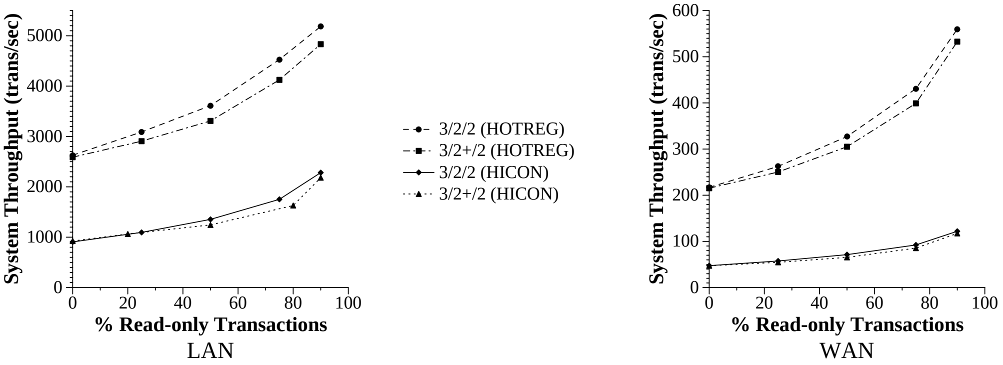


Figure 7-12: Cost of PL-2+ for read-only transactions

the 3/2+/2 system are read-only. Since read-only transactions never abort in the 3/2/2 system, the throughput difference between the two systems tends to increase due to this factor.

The throughput variation is shown in Figure 7-12. At 0% read-only transactions, there is no throughput difference. As the percentage of read-only transactions is increased, the second factor dominates and increases the performance difference between the systems. However, beyond a certain percentage of read-only transactions (e.g., 50% for HICON), the contention is reduced sufficiently and the throughput difference decreases, i.e., the first factor dominates.

Cost of EPL-2+

The difference between the 3/2+/2+ and 3/2+/2 systems in Figure 7-11 shows the cost of providing EPL-2+ to transactions as they execute; the performance difference between the 3/3/2+ and 3/3/2 systems also evaluates the cost of EPL-2+. In both cases, we see that the performance difference is negligible. As stated above, the multistamp processing overheads are low in our implementation. Thus, the extra cost is only due to consistency stalls. The 3/2+/2+ system can have more stalls than the 3/2+/2 system since the former system provides stronger guarantees to transactions as they run; in the 3/2+/2 system, a read-only transaction can stall only at its commit point whereas in the 3/2+/2+ system, any transaction can stall during its execution. We measured the overhead of invalidation-request messages terms of the *stall rate*, i.e., the relative ratio of these messages to the number of fetches in this system. We observed that the stall rate is low for the 3/2+/2+ system and hence it has a negligible impact on performance, e.g., in the worst case, the stall rate was 2.5% (for HOTREG in a WAN environment). In Section 7.5, we present an analysis on how the stall rate changes with a variation of different system parameters.

A consistency stall penalizes performance in a LAN environment less than a WAN environment since the message latency is much lower in LANs. In fact, in such environments, stalls can sometimes be beneficial since they force a client to get invalidation information slightly early, i.e., consistency stalls can keep a client cache even more up-to-date and hence help the client to abort early or avoid some aborts. This benefit is lost in WAN environments where a message roundtrip slows down a transaction substantially.

7.4.2 Overheads of PL-3U, EPL-3U and EPL-3

We now discuss the performance penalties of providing PL-3U and EPL-3U guarantees. Figure 7-13 shows the throughput of various systems that provide PL-3U and EPL-3U guarantees. As before, we have shown the throughput values relative to the 3/2/2 system.

System	**LAN environment**			**WAN environment**		
	LOWCON	HOTREG	HICON	LOWCON	HOTREG	HICON
3/2/2	7344	3647	1355	352.9	325.1	71.4
3/2+/2	7177 (2 %)	3308 (9 %)	1243 (8 %)	346.3 (2 %)	304.4 (6 %)	65.5 (8 %)
3/2+/2+	7173 (2 %)	3335 (9 %)	1243 (8 %)	344.1 (2 %)	301.3 (7 %)	64.8 (9 %)
3/3/2	7020 (4 %)	3304 (9 %)	1227 (9 %)	307.6 (13%)	276.7 (15%)	62.9 (12%)
3/3/2+	7072 (4 %)	3352 (8 %)	1231 (9 %)	310.8 (12%)	271.1 (16%)	62.2 (13%)
3/3U/2	7204 (2 %)	3288 (10 %)	1236 (9 %)	346.6 (2 %)	301.9 (7 %)	65.3 (9 %)
3/3U/3U	7146 (3 %)	3367 (8 %)	1243 (8 %)	342.4 (3 %)	296.0 (9 %)	64.3 (10 %)
3/3/3	7059 (4 %)	3336 (9 %)	1229 (9 %)	299.5 (15 %)	257.3 (21 %)	61.1 (14 %)

Figure 7-13: System throughput (transactions per sec) and performance penalty (relative to the 3/2/2 system) for different isolation guarantees.

The figure shows that the cost of providing PL-3U to read-only transactions is similar to the cost of providing PL-2+ for these transactions (compare the 3/2+/2 and 3/3U/2 systems). Similarly, the performance of the 3/3U/3U and 3/2+/2+ systems is similar showing that the extra cost of EPL-3U over EPL-2+ is low.

Level PL-3U provides stronger guarantees than PL-2+ since PL-3U captures conflicts due to anti-dependencies whereas PL-2+ does not. Capturing constraints due to anti-dependencies results in larger multistamps and hence more truncation. Thus, more read-only transactions can stall at commit time than with PL-2+. However, our results show that these extra stalls are few; in the worst case (HOTREG in WANs), 4% read-only transactions stalled in the 3/3U/2 system instead of 2.3% in the 3/2+/2 system. Again, as for PL-2+, it is usually the case that a client is up-to-date with respect to its participants when the commit point is reached.

The performance of the 3/3U/2 and the 3/3U/3U systems is comparable to CLOCC in LAN environments; in WAN environments, these systems perform better than CLOCC because they avoid a commit roundtrip message for read-only transactions in most cases. Thus, an application programmer may choose PL-3U instead of PL-3 for read-only transactions to achieve better performance; since PL-3U is very close to PL-3, the loss in ease of programming with PL-3U is small.

For executing transactions, we observed that the cost of providing EPL-3U is not high relative to EPL-2+. Even though there are more consistency stalls in the 3/3U/3U system than the 3/2+/2+ case, the number of stalls are still low enough so that system throughput is not significantly affected. In Section 7.5, we compare the stall rates of the 3/3U/3U and 3/2+/2+ systems.

In the PL-3U and EPL-3U implementations, a server may send extra multistamp-request messages at prepare time to other servers if it requires the final multistamp of a committed transaction (see Section 5.4.1). In our simulation experiments, we observed that very few update transactions required such messages in all workloads. In the worst case of HICON, 15% of update transactions (i.e., less than 8% of the total number of transactions) required a message to be sent; for HOTREG, less than 7% of update transaction needed this extra message. Compared to fetches and commits, these messages represent a very small fraction of the total number of messages in the network. Hence, the impact of the extra multistamp-request messages among servers is negligible.

For the sake of completeness, we have also shown the throughput of the 3/3/3 system in Figure 7-13. In this system, all transactions are committed using CLOCC and a transaction's multistamp s computed by merging the multistamps of all transactions that it depends and anti-depends on; in a PL-3U or EPL-3U system, only the multistamps of update transactions are merged. As a result, multistamps are bigger, there is more truncation and hence more consistency stalls. However, these stalls are still low enough to not adversely affect performance. In LAN environments, the extra throughput degradation of this system is negligible compared to the 3/3U/3U system. In WANs, this system has lower throughput than the 3/3U/3U system because an extra commit roundtrip message for committing read-only transactions has a higher cost in these environments.

7.4.3 Comparing PL-2L with PL-2 and PL-2+

We used the HICON and HOTREG workloads to evaluate the costs of our PL-2L scheme. Recall that this scheme also provides EPL-2L to running transactions, i.e., we ran the workloads with the 2L/2L/2L system. The performance of this system is similar to that of the 2/2/2 system. The only extra cost of this system is the extra stalls that may occur during a transaction's execution. As in the EPL-2+ case, we observed that a client sends very few invalidation-request messages compared to fetch requests and hence the performance degradation is low. Thus, PL-2L can be easily provided to applications instead of PL-2 without a significant cost. Of course, a programmer has to be careful about committing update transactions at a level below PL-3 since the database state can be corrupted.

We also ran experiments using the 3/2L/2L system and observed that its performance is similar to that of the 3/2/2 system. It performs about 10% better than the 3/2+/2 system because it never aborts a read-only transaction whereas the latter system can.

7.5 Stall Rate Analysis: Cost of Multistamps

For our multistamp-based implementations to be efficient, it is crucial that the consistency stall rate be low. In this section, we explore the variation of the stall rate when different system parameters such as multistamp size, percentage of multi-server transactions, etc., are changed. We use the 3/2+/2+ system since it stresses the system more than the 3/2+/2 system. We also analyze the relative stall rates due to EPL-2+, EPL-3U and EPL-3 in this section.

A stall has a low performance impact on the system throughput compared to a fetch since an invalidation-request message and its reply are small compared to fetch replies. However, it is desirable to keep the stall rate as low as possible because extra messages increase client/server CPU and network utilization. If any of these resources is already near saturation, a high stall rate can impact performance in an adverse manner. Furthermore, other applications sharing the network will be penalized if the number of consistency messages is high. Thus, even if the throughput difference between two systems is negligible, it is still desirable to have relative few extra messages due to consistency stalls. In this section, we use consistency stall rate as a performance measure for comparing different workload/isolation combinations. Recall that the consistency stall rate is the ratio of invalidation-request messages to the total number of fetches in the system.

In the client-merger scheme presented in Section 5.3.7, a client computes the read-dependency part of a transaction's multistamp before sending a commit message to the participant. We did not use this scheme for computing a transaction's multistamp in our simulator. Instead, we used the following mechanism to generate multistamps at servers based on coarse-grained intersections. When a transaction T_i prepares, the server considers T_i to depend on T_j if T_j has modified page P and T_i has read some object on P. Thus, our multistamps are more conservative than necessary and hence give an upper bound on the number of stalls that can occur with the client-merger scheme. We ran an experiment with fine-grained checks and observed stall rates that were 25% to 30% lower. However, since the stall rate is already low for our stressful workloads, we used coarse-grained checks for all our experiments. Note that validation of transactions is still performed using fine-grained checks.

Here is a brief synopsis of our experimental results:

- Small multistamps of size less than 100 bytes are sufficient to maintain a low consistency stall rates (less than 3-4%).
- The stall rate remains low for a large variety of system parameters. The impact of a stall is very low in LAN environments; in WAN environments, this cost is higher. However, a low stall rate in both environments results in a negligible impact on the overall performance.
- The number of stalls increases with contention but the stall rate does not necessarily increase.

- EPL-3U and EPL-3 may increase the stall rate by a factor of 2 to 5 relative to EPL-2+ but the stall rate value is still low (less than 7% in the worst case).

- Simple aging of multistamps is not sufficient for keeping them small; pruning is also needed.

- Increasing the percentage of multi-server transactions does not necessarily increase the stall rate; a higher percentage of multi-server transactions helps in keeping a client more up-to-date with respect to its connected servers, especially non-preferred servers.

- The stall rate does not change significantly when more clients are added to a cluster or when more clusters are added to the system, i.e., the multistamp-based schemes scale well with the size of the system.

We chose the HOTREG and HICON workloads since they stress the performance of the 3/2+/2+ system more than LOWCON. Our evaluation is pessimistic since we have chosen parameters that are not favorable to the multistamp-based implementations. For example, we have chosen two preferred servers instead of one (which is normally expected); as a result, a large number of client stalls occur due to the fact that a client keeps "switching" between these two servers. Similarly, we used a high percentage of multi-server transactions to increase the distribution of invalidations to a large number of servers. Our workloads, especially HOTREG, have been designed to stress the multistamp truncation mechanism. In HOTREG, 1 in 10 transactions modify the small hot-shared region and there are 30 clients sharing 2 preferred servers; thus, at least one client is modifying the hot region at any given time. Due to contention on this hot-shared region, a large number of invalidations are generated resulting in larger multistamps and hence more truncation. In realistic environments with less stressful conditions, we expect stall rates to be lower than the stall rates shown for HICON and HOTREG.

7.5.1 Consistency Stalls and Contention

The number of stalls is directly related to contention in the workload. Let us consider the case when a client C observes a high number of stalls. Suppose that C stalls with respect to a server S. Since (piggybacked) invalidation messages keep client caches almost up-to-date, it must be the case that C has fetched object *x* that was modified by a recently-committed transaction T_i and T_i must have generated invalidations for client C at server S. If transactions reads the modifications of recently-committed transactions close in time, the workload must have high contention (at least at the page-level since servers maintain page-level cache directories).

The above argument holds in the other direction as well, i.e., more contention leads to more stalls. Due to high contention, a client C can read an object *x* that was recently modified by another transaction T_j and stall if T_j's commit generated an invalidation for C at some other server (C may not be recent enough with respect to that server). Contention also leads to more stalls for another reason. As contention increases, each transaction depends/anti-depends on more transactions and hence multistamps become bigger. This leads to more truncation, a higher multistamp threshold, and hence a higher number of stalls.

It is important to understand that a large number of stalls does *not* necessarily imply a high stall rate. A high contention workload also has a large number of fetches due to conflict misses. Thus, in the HICON workload, we observed that the stall rate was low even though the number of stalls was high. For other workloads, we observed that there are fewer stalls due to lower contention. Thus, the stall rate is low across all workloads.

A client C stalls when it depends on a multi-server transaction and C is not up-to-date with respect to one of the servers being accessed in C's current transaction. A multi-server transaction can stall at two points: when it accesses a server for the first time and when it performs a fetch; we call these *start stalls* and *running stalls*, respectively. However, a single-server transaction can stall only when it starts; it cannot stall after it performs a fetch since the multistamp constraints with respect to other servers are irrelevant for this transaction.

7.5.2 EPL-2+ Stall Rate

Figure 7-14 shows the consistency stall rate of the 3/2+/2+ and 3/3/2+ systems for different workloads. The stall rate is similar for both systems since they have a similar caching behavior; hence, we will just consider the 3/2+/2+ system for our discussion.

System	**LAN environment**			**WAN environment**		
	LOWCON	HOTREG	HICON	LOWCON	HOTREG	HICON
3/2+/2+	0.5%	1.9%	1.6%	0.5%	2.5%	1.8%
3/3/2+	0.5%	1.9%	1.6%	0.5%	2.4%	1.8%

Figure 7-14: Consistency stall rate for EPL-2+

The figure shows that the stall rate is very low for all workloads in LAN and WAN environments. For low contention workloads such as LOWCON, the stall rate is negligible because transactions rarely depend on recently-committed transactions (due to low contention on the shared region).

For a high contention workload such as HICON, the number of stalls is much higher than HOTREG (17500 stalls occur in our experimental setup for HICON as opposed to 3900 for HOTREG). However, HICON also suffers from a large number of conflict misses; as a result, the number of stalls relative to the number of fetches (approximately 23 per transaction) is low. In the HOTREG workload, the small hot-shared region at a server results in a high conflict rate causing dependencies on recently-committed transactions; most of the invalidations in this workload are due to this region. Since the hot-shared region has higher contention than the shared region of LOWCON and most of the hot-shared region can be cached at clients, the stall rate for HOTREG is

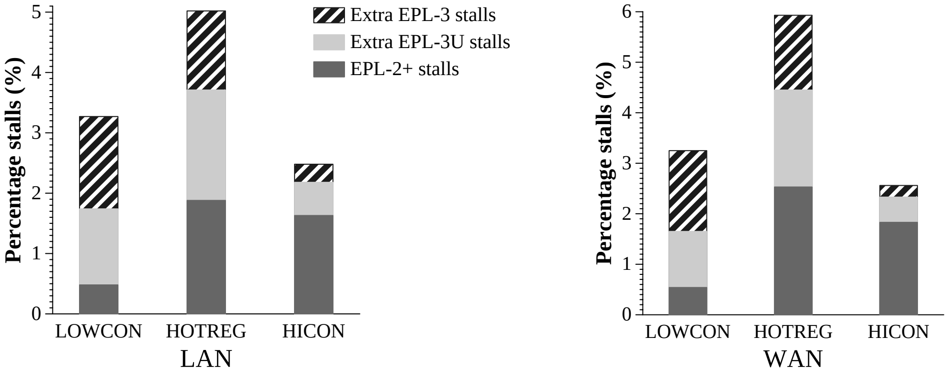


Figure 7-15: Consistency stall rate for EPL-2+, EPL-3U or EPL-3.

higher than that for LOWCON. However, the conflict rate in HOTREG is not as high as the HICON workload since only 10% of the accesses go to this region and it is modified by only 1 in every ten transactions. Thus, a transaction has significantly fewer fetches (approximately 4) resulting in a stall rate similar to HICON. Of course, the stall rate in HOTREG is still very low (below 3%).

7.5.3 Stall Rate Comparison for EPL-2+, EPL-3U and EPL-3

Figure 7-15 presents the stall rates for EPL-2+, EPL-3U and EPL-3; above the EPL-2+ stall rate, we show the extra stalls for EPL-3U and EPL-3. We compared the 3/2+/2+, 3/3U/3U and 3/3/3 systems for our experiments; except for the extra commit message in the 3/3/3 system for read-only transactions, these systems had a similar performance in terms of abort rate, fetches, etc. As expected, EPL-3U has a higher rate than EPL-2+ since it captures anti-dependencies as well as dependencies; the number of transactions whose multistamp is merged into a committing transaction's multistamp doubles for HOTREG and LOWCON. Thus, the multistamp size increases and more truncation takes place, resulting in more stalls. In the case of EPL-3, a transaction T_i's multistamp also contains multistamps of read-only transactions that T_i depends or anti-depends on. For example, in HOTREF and LOWCON, the number of transactions whose multistamps are marged into a committing transaction's multistamp is more than three times than the corresponding number for the EPL-2+ implementation; this results in EPL-3 having a higher stall rate compared to EPL-2+ and EPL-3U.

7.5.4 Size of Multistamps

We varied the size of multistamps to understand the impact of aging and truncation on the stall rate in the 3/2+/2+ system. Figure 7-16 shows the variation of the stall rate as the maximum size for multistamps is increased from zero entries (just the threshold timestamp) to the size reached when just aging is used. (A system that uses just the threshold timestamp is using Lamport clocks [Lam78].) When multistamps are very small (e.g., 0 or 1 entries), most of the information

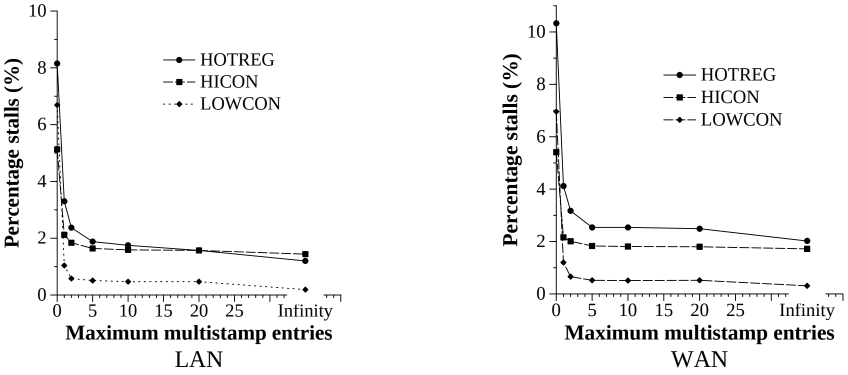


Figure 7-16: Consistency stall rate as the size of multistamps is varied

is truncated resulting in a high stall rate. However, with a slight increase in multistamp size, the information in the multistamps is sufficient to lower the stall rate significantly.

When multistamps are not truncated, they can grow to a size that equals the number of client-server connections in the system, i.e., 960 entries. Figure 7-17 shows that aging reduces the size of multistamps but still they are relatively big. Large multistamps increase network utilization and server CPU overheads substantially resulting in a significant degradation of system performance. For example, in the LOWCON workload for the LAN environment, where network is a bottleneck resource, the throughput of the 3/2+/2+ system that uses only aging is 40% lower than the default 3/2+/2+ system (in which multistamps larger than 5 entries are truncated). These results demonstrate that pruning is needed to ensure that the costs of multistamps is low. As shown in Figure 7-16, pruning to a reasonable size does not increase stall rate significantly. Based on this observation, we chose a multistamp of 5 entries for our default system; the size of 5-entry multistamps is less than 100 bytes.

Workload	**LAN environment**			**WAN environment**		
	LOWCON	HOTREG	HICON	LOWCON	HOTREG	HICON
Entries Size	113 970 bytes	314 2700 bytes	449 3800 bytes	305 2600 bytes	674 6100 bytes	620 5200 bytes

Figure 7-17: Size of multistamps when only aging is used

7.5.5 Increasing Multi-server Transactions

In another experiment, we varied the percentage of multi-server transactions in the system from 10% to 100% in the 3/2+/2+ system; note that our default setting of 20% already denotes a high percentage of multi-server transactions. The stall rate variation is shown in Figure 7-18; we show results for HICON and HOTREG since they are more stressful than LOWCON. There are two

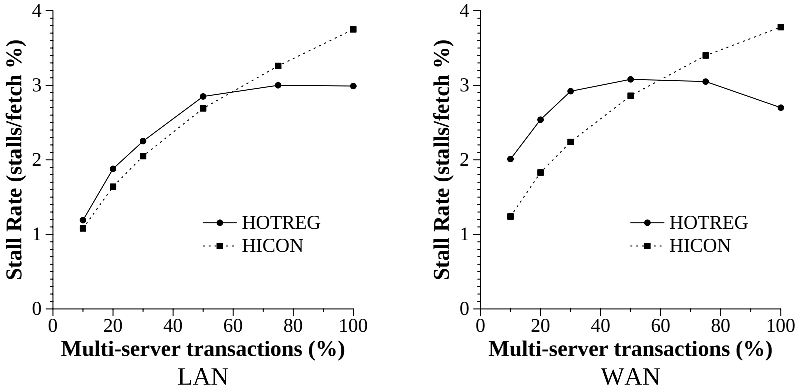


Figure 7-18: Stall rate for EPL-2+ as the percentage of multi-server transactions are increased.

effects that impact the stall rate. First, with a higher percentage of multi-server transactions, more dependencies are generated across servers and more transactions become dependent on multi-server transactions; this can result in bigger multistamps and more truncation. Furthermore, as discussed earlier, a multi-server transaction can stall while it is executing (i.e., running stalls can occur) whereas a single-server transaction can stall only at the beginning. Thus, a higher percentage of multi-server transactions tends to increase the stall rate.

However, a second opposing effect tends to decrease the stall rate as multi-server transactions are increased. A higher percentage of multi-server transactions results in more transactions that involve non-preferred servers. Thus, clients communicate with these servers more often and become more up-to-date with respect to them; as a result, they incur fewer start stalls due to these servers.

In our simulation setup, different variations in stall rate are observed in LAN and WAN environments for the HOTREG workload. In LANs, the relative ratio of the timeout period (1 second) to the transaction execution time (approximately 100 msec) is lower than this ratio in WAN environments; in WANs, the timeout period is 30 seconds and the transaction execution time is approximately 800 msec. Since a client can run more transactions involving non-preferred servers during its timeout period in WANs than in LANs, the possibility of a start consistency stall at a non-preferred server is higher in the WAN case. Thus, at a high percentage of multi-server transactions, the second factor dominates (i.e., increased communication with non-preferred servers is relatively more helpful) and decreases the stall rate for HOTREG in WAN environments.

For HICON, there are significantly more fetches per transaction than in HOTREG resulting in a higher execution time; as a result, the relative ratio of the timeout period to transaction execution time is lower. Hence, clients are mostly up-to-date with respect to non-preferred servers as well. In this case, the first factor dominates and the stall rate increases with an increase in the percentage of multi-server transactions.

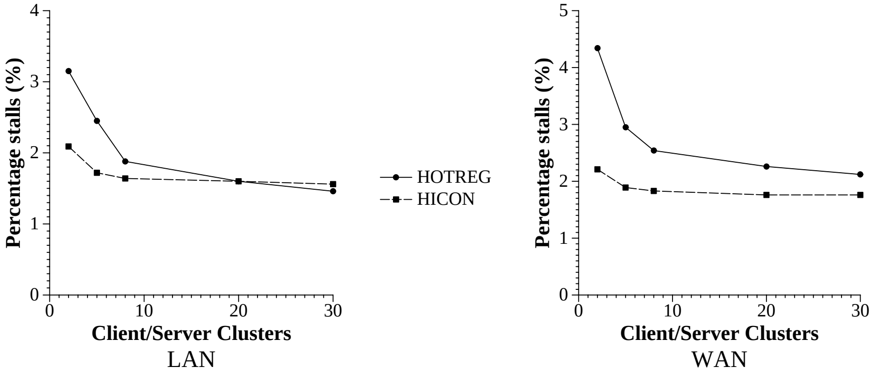
IN LAN environments, the first factor dominates for both workloads and the stall rate increases as the percentage of multi-server transactions is increased.

This experiment shows that even with a high timeout period and a high percentage of multi-server transactions, the stall rate is still low for stressful workloads such as HICON and HOTREG. When the timeout period is high, the extra stalls are primarily due to non-preferred servers: if fewer transactions at a client access non-preferred servers, there is a higher likelihood of a start stall with respect to these servers. However, since only a small fraction of transactions access non-preferred servers, the *overall* stall rate is still low. Thus, the timeout period need not be set very aggressively for maintaining a low stall rate.

To further understand the impact of multi-server transactions, we ran an experiment where a client had more than 2 non-preferred servers; we varied the number of non-preferred servers from 2 to 9, and allowed transactions to use all connected servers. In the worst case of HOTREG in WAN environments, the stall rate increased from 2% to 11.8%. Additional experiments showed that this effect was largely due to transactions that used large numbers of servers rather than the number of connections per client. Of course, as discussed in Section 6.2.1, transactions that use 8 or 9 servers are highly unlikely in practice. Furthermore, even if transactions do access a large number of servers, we do not expect a high update ratio of 20% as used in our experiments, i.e., we expect the stall rate to be low even in such cases.

7.5.6 Scalability

To evaluate the scalability of our mechanisms, we varied the number of groups in the system from 2 to 30, i.e., the largest system had 60 servers and 900 clients. The results for the 3/2+/2+ system are shown in Figure 7-19. In a larger system, multistamps can spread to more servers; merging of entries at a higher number of servers can increase the multistamp size, and hence lead to more truncation and more stalls. However, we observed that the stall rate *decreases* as more clusters are added in the system. The reason is that dependencies across groups are propagated only via non-preferred servers. With the increase in the number of groups, the likelihood of two clients in a



LAN

WAN

Figure 7-19: Stall rate variation with the size of the system

cluster sharing the same non-preferred servers decreases. Suppose that a client C's transaction T_i modifies objects at servers X and Y. Suppose another client D reads T_i's modifications at server X and is informed about T_i's multistamp. In a larger system, it is less likely in a larger system that C and D also share server Y; as a result, T_i's multistamp constraints will not result in D's transaction stalling due to server Y. Thus, this factor dominates over the truncation effect and the stall rate decreases in a larger system.

To better understand the spread of dependency information across servers, we ran an experiment in which one preferred server of each client was chosen in a different cluster than the client's cluster with varying probabilities; in this setup, dependency information is spread across clusters due to preferred as well as non-preferred servers. As in the above experiment, there are two conflicting factors that affect the stall rate as the probability of the second preferred server being in a different cluster is increased. First, the stall rate can increase due to more spreading of information (larger multistamps and more truncation). Second, the stall rate can decrease due to less sharing of the same preferred servers by clients. In the default system, two clients in the same cluster share the same preferred servers; with different topologies, the likelihood of two clients in any cluster sharing the same preferred servers is lower. Depending on the workload and the network environment, we observed that the stall rate remained the same or changed slightly. More importantly, the stall rate remained below 3%.

We also varied the number of clients per cluster from 10 to 80; the results are shown in Figure 7-20. With a larger number of clients, contention levels are higher and hence the stall rate is expected to be higher. However, in LAN environments, the stall rate actually *decreases*. In LANs, the network and the disk become more heavily utilized when more clients are added to a cluster resulting in an increase in network delay due to congestion. Extra delays result in a higher transaction execution time and the relative ratio of the timeout period (for "I'm alive" messages) to the transaction's execution time is reduced. Thus, more timeout messages are sent and clients become relatively

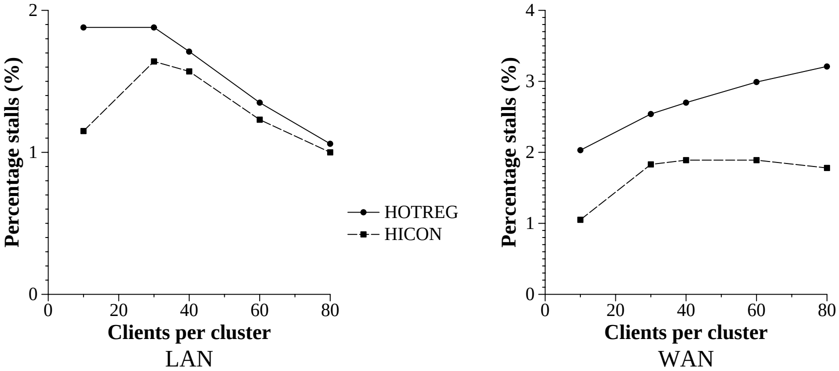


Figure 7-20: Stall rate variation with number of clients per cluster

more up-to-date with respect to servers; thus, the stall rate decreases with as more clients are added to each cluster in the LAN case. In HICON, this effect does is not significant at 10 clients per cluster since the network is not saturated. Thus, the stall rate increases initially in HICON; when the network saturates, the stall rate decreases.

In WAN environments, the network delays do not change substantially as the more clients are added to each cluster. Thus, the above effect is not observed in the WAN case except in HICON when each cluster has 80 clients; in this case, the network is 96% utilized and the stall rate decreases slightly due to extra timeout messages.

The above experiments show that our multistamp-based technique scales well as more clients and servers are added to the system since larger multistamps are not needed to maintain a low stall rate.

Chapter 8

Conclusions

Data consistency is an important requirement in database systems since it allows application writers and end users to make sense of the data in the presence of concurrency and failures. Although strong consistency guarantees such as serializability provide a simple programming model in these systems, they may result in increased resource consumption, communication delays or transaction aborts. Furthermore, such overheads may be unnecessary for applications that do not need strong consistency guarantees for correct operation. To address this problem, current databases allow applications to trade off consistency for a potential gain in performance. Furthermore, an ANSI/ISO standard has been established that defines different levels of consistency (called degrees of isolation); this standard is supported by all commercial database systems. However, the current ANSI standard is ambiguous and an approach suggested to fix this problem is overly restrictive since it disallows efficient optimistic and multi-version mechanisms. Apart from the inadequacy of the existing isolation definitions, there is also a need for high-performance weak consistency mechanisms in distributed client-server systems.

This thesis has focused on the problem of providing data consistency efficiently to applications in database systems. It addresses the limitations of the existing isolation definitions and also presents efficient implementations of various weak consistency levels for client-server systems. It makes contributions in three areas. First, it presents new specifications of the existing ANSI isolation levels that allow a variety of concurrency control implementations. It also presents definitions for other commercial levels using a uniform framework and proposes two new isolation levels that provide useful consistency guarantees to applications. Second, it presents new and efficient implementation techniques based on multipart timestamps for supporting different weak consistency levels in distributed client-server systems. Finally, it presents the results of a simulation study that evaluates the relative performance of different consistency implementations.

8.1 Isolation Level Specifications

We have redefined the existing ANSI isolation levels to allow a wide range of concurrency control implementations including optimism, locking and multi-version schemes in Chapter 3. Our isolation levels are similar to the current ANSI levels and capture their essence thereby making it easy for application writers to use them. Our isolation levels are named PL-1, PL-2, and PL-3, where PL stands for portable level. Like the existing definitions, each isolation level in our specifications captures different kinds of conflicts, e.g., PL-1 captures write-write conflicts, and serializability, or PL-3, captures all types of conflicts. We have removed some of the constraints from the existing definitions since they were applicable only to particular concurrency control schemes (locking) and were overly restrictive for optimistic implementations.

An important property of our specifications is that they allow different guarantees to be provided to running and committed transactions. This flexibility is needed since efficient optimistic implementations may provide strong guarantees such as serializability to committed transactions but weak guarantees to transactions as they execute. The guarantees for committed transactions ensure that the database integrity is not destroyed while guarantees for running transactions ensure that database programmers can write application code with the assumption that the program will not observe violated integrity constraints and behave in an unexpected manner, e.g., crash or display erroneous data on a screen. Our specifications for both types of transactions are similar making it easy for application programmers to use them.

Our specifications use a combination of conditions on serialization graphs and transaction histories. Serialization graphs contain nodes for committed transactions and directed edges corresponding to different types of read and write conflicts. They provide a simple way of capturing multi-object constraints. Similar graphs have been used in the past for serializability semantics-based correctness criteria and extended transaction models. Our approach is the first that applies these techniques to defining ANSI and commercial isolation levels.

We have also specified a variety of guarantees that can be provided to predicate-based operations at weak consistency levels in an implementation-independent manner; a database system can choose the guarantees that it wants to support at each consistency level. Earlier definitions for these operations were either incomplete, ambiguous, or specified in terms of an implementation such as locking or in terms of a particular database language such as SQL.

In Chapter 4, we have presented implementation-independent definitions of various commercial levels such as Cursor Stability, Snapshot Isolation, and Read Consistency; earlier definitions of these levels were either informal or based on a locking implementation. Our specifications for these levels are based on variations of the graphs used for defining the ANSI levels; different types of nodes and edges are added to capture constraints relevant to each level.

We also presented the specifications of two new isolation levels, PL-2+ and PL-2L, that provide useful guarantees to application writers. Level PL-2+ is the *weakest* level that provides consistent reads to transactions; it allows a transaction T_i to commit only if T_i has observed a causally-consistent database state, i.e., if T_i observes the updates of another transaction T_j directly or indirectly, it must not observe an older version of an object that was modified by T_j. Level PL-2+ is also specified using our graph-based technique. An interesting feature of this level is that it disallows all phenomena that Snapshot Isolation, an isolation level used in Oracle, was intended to disallow. Furthermore, since PL-2+ is weaker than Snapshot Isolation, it has the potential of being implemented more efficiently than Snapshot Isolation, especially in distributed client-server systems. Thus, it may desirable to use PL-2+ over Snapshot Isolation.

Our second new level, PL-2L, captures useful properties of the PL-2 lock-based implementation that uses short read-locks and long write-locks. Level PL-2L ensures that a transaction observes a monotonically increasing prefix of the database history as it executes. This level can be used for legacy applications that make monotonicity assumptions about the underlying lock-based concurrency control implementation; when the concurrency control mechanism is changed to (say) optimism, such applications can continue to work correctly using PL-2L. Level PL-2L has also been defined using a variation on serialization graphs and is similar to Oracle's Read Consistency.

8.2 Weak Consistency Mechanisms

In Chapter 5, we have presented optimistic schemes to support new and existing isolation levels in distributed client-server systems. In these systems, servers store persistent objects and client cache them on their machines for good performance. Our mechanisms are based on an optimistic scheme called CLOCC, which has been shown to perform well in client-caching systems for a wide range of workloads and system parameters.

We developed techniques based on multipart timestamps or multistamps for a variety of weak isolation levels. These multistamps efficiently capture different types of conflicts that are relevant to a particular isolation level and are used to warn clients about potential consistency violations. Our protocols are designed to be lazy: a client C sends extra messages to a server only if C does not have information as indicated by the multistamp. Being lazy helps in reducing extra messages that are sent by clients since the relevant consistency information is likely to be present in the client cache when it is needed. Furthermore, it allows servers to piggyback multistamp information on existing messages thereby reducing the overheads of our protocols.

A negative aspect of multistamps is that they do not tend to scale with a large number of clients and servers. We have developed a novel and simple technique called *multistamp truncation* that keeps multistamps small by removing old consistency requirements and replacing them with approximate information; there is little value in propagating old requirements since clients receive consistency information piggybacked on other messages from servers that enables them to satisfy these requirements. An important aspect of our multistamps is that they contain real clock values instead of logical clock values. This approach allows us to make time-based judgements and use the time values to approximate information at various parts of the system, e.g., to determine which tuples are old in a multistamp. For good performance (not correctness), we assume that clocks are loosely synchronized within a few milliseconds or tens of milliseconds; this is a reasonable assumption because protocols such as the Network Time Protocol are able to achieve such levels of synchronization even across wide area networks.

The multistamp-based mechanisms are used for providing PL-2L, PL-2+ and PL-3U guarantees to update and read-only transactions as they commit; they are also used for providing the strong consistency guarantees such as EPL-2+ and EPL-3U to running transactions. Read-only transactions form an important class of transactions and our PL-2+ (or PL-3U) implementation for committing these transactions offers many important advantages over CLOCC while providing strong consistency guarantees. First, the multistamp-based implementations reduce transaction latency since they avoid sending a commit message to servers for committing read-only transactions whereas CLOCC requires such communication for every read-only transaction. Second, the system becomes more scalable since servers perform less work (they do not have to validate read-only transactions). Update transactions have to be validated only against other update transactions thereby reducing the server's load even more. Finally, update transactions are not aborted due to read-only transactions.

Our mechanisms have been designed to impose low network, processor, and memory overheads. Most of the consistency information is piggybacked on existing messages in the system; furthermore, the increase in the size of network messages is low since multistamp truncation allows us to keep small multistamps while adding only a few small messages to the network. Furthermore, the data structures required to maintain multistamps impose low memory overheads at servers and clients.

8.3 Experimental Evaluation

This thesis also evaluates the relative performance of implementations of different isolation levels in distributed client-server systems. Our results are applicable to a data-shipping architecture where the transaction code is executed at client machines. To our knowledge, this is the first published study that compares implementations of different isolation levels in such environments. We used a simulator to compare various consistency schemes in LAN and WAN environments for a range of system parameters using workloads with varying amounts of contention.

We were interested in answering three questions. First, what are the performance gains achieved by executing update transactions at levels below serializability? This is an important issue because committing update transactions at lower levels has a high productivity cost: a database programmer must carefully analyze the application code and ensure that it does not corrupt the database state when executed at a low isolation level. Second, how expensive is it to provide strong consistency guarantees to read-only transactions? Finally, what is the performance penalty of providing consistent views to transactions while they are executing, i.e., what are the space and network overheads imposed by multistamps and how effective is multistamp truncation?

We derived our multi-server workloads based on workloads used in earlier concurrency control studies for single-server systems. Transactions in our studies access a few hundred objects and clients can cache the accessed objects for the duration of a transaction. Our results show that providing strong consistency guarantees to update, read-only, and running transactions are not necessarily expensive. Our experiments revealed the following answers to the questions stated above:

- The cost of providing stronger consistency guarantees including serializability to update transactions is very low for low-contention workloads since the CPU and communication overheads of CLOCC and the multistamp-based schemes are low for normal (i.e., successful) transactions. This is an important result since many applications exhibit low-contention and such workloads are one of the main environments recommended for using weaker isolation levels.

  Even for stressful workloads such as HOTREG and HICON, the cost of providing serializability to update transactions is approximately 10%. As contention is increased, a large number of transactions will abort in any optimistic scheme including CLOCC. However, in CLOCC, a large increase in the abort rate does not result in a corresponding performance degradation because CLOCC has low restart costs (e.g., undo logs at clients help in avoiding some fetches when a transaction reruns) and invalidation messages prevent excessive wastage of work by aborting a transaction early during its execution.

  Our sensitivity analysis results show that the relative performance penalty of serializability is higher compared to PL-2 for update transactions only if the workload has moderate to high contention *and* the cost of restarting a transaction is high.

- The cost of providing serializability to read-only transactions compared to PL-2 is relatively low (approximately 5%) for low-contention workloads in LAN environments. For medium to high-contention workloads, more contention leads to higher performance degradation; however, the performance penalty is less than 10% relative to a system that provides serializability only for update transactions.

  In WAN environments, where message costs are higher, an extra message roundtrip for committing read-only transactions in CLOCC has a higher performance penalty compared to LANs. As a result, even for a workload such as LOWCON, CLOCC has a throughput degradation of approximately 15% relative to a PL-2 system that commits transactions locally at client machines; HOTREG and HICON have overheads of 10-15% in this case. To alleviate this problem, our multistamp-based techniques for PL-2+ and PL-3U can be used since they avoid a commit message roundtrip for read-only transactions in most cases (more than 96% of read-only transactions). Fewer commit message roundtrips for read-only transactions help in improving the system throughput for all workloads. For example, the performance penalty of providing PL-2+ or PL-3U to read-only transactions in LOWCON is approximately 2% compared to a system that provides PL-2 to such transactions. Providing PL-2+ or PL-3U instead of serializability to read-only transactions may be acceptable since these levels provide strong consistency guarantees to such transactions; thus, it should still be relatively easy for programmers to reason about the correctness of the application code.

- The cost of providing strong consistency guarantees such as EPL-2+ and EPL-3U to executing transactions is low. Our experiments show that the multistamp-based implementations are efficient and small multistamps (less than 100 bytes) result in very low stall rates for a wide variation of system parameters. Since fewer than 3% of fetches result in a consistency stall for a system that provides EPL-2+, the throughput impact of multistamps and consistency stalls is negligible. Thus, with very low overheads, a database system can provide a strong guarantee that an executing application code will never observe an inconsistent state of the database.

8.4 Future Work

There are several directions in which this work can be extended. Our isolation conditions have been specified in terms of reads and writes. Researchers have developed correctness criteria based on semantics of operations [Lam76, GM83, Her90, WL85, SS84] rather than simple reads and writes. Such conditions allow more concurrency and better performance while ensuring that an application programmer has a formal model to reason about program correctness. Thus, it will be beneficial to extend our isolation conditions for dealing with complex operations on abstract data types. It would be important to ensure that the new semantic isolation levels be similar to our read/write-based levels so that application programmers do not have to deal with a diverse set of definitions.

Another important issue is how the interactions of various isolation levels affect the database state. Such an understanding of isolation level interactions is crucial for a program's correctness. Suppose that a transaction T_i commits using PL-3 and T_j commits using PL-2. To ensure that T_i observes a serializable state of the database, the application writer for T_j must "know" that the database is updated in a consistent manner. Thus, when a set of objects is read by a transaction, it is difficult to determine the guarantees that are provided to these reads. This problem exists with the current isolation definitions as well.

A possible approach to handle this problem is to label each object with an isolation level to indicate that the object has never been directly or indirectly "affected" by a transaction below that level; the notion of affects is similar to the idea of dependencies and anti-dependencies. When a transaction commits at (say) PL-2+, we can determine if it has read only those objects that have been directly or indirectly affected by PL-2+ or higher update transactions. Such an approach was briefly proposed in [GLPT76] but we are not aware of a full solution developed for that proposal. Labeling of objects to control flow of information for privacy and secrecy reasons has been explored for many years [DD77, Mye99]. It would be interesting to combine those ideas with our isolation definitions. A problem with labeling objects with the minimum isolation level transaction that has ever modified them is that objects may soon get labeled with very low isolation levels. To deal with this problem, a mechanism needs to be provided that allows "lifting" of object isolation levels if applications have extra knowledge about these objects and their read/update history. We can borrow ideas from the information flow control domain since an analogous problem exists there when applications want to release private data (i.e., declassify information) in a controlled manner. Techniques for safe declassification have been proposed in the literature for addressing this problem [Mye99] and it would be interesting to use similar ideas for lifting an object's isolation level.

Our isolation definitions for committed transactions can be used in mobile systems as well. However, they need to be extended to handle "tentatively committed" transactions in such systems. In these systems, clients may be disconnected from the servers and may read/write data stored at its machine or other clients' machines. When a transaction is completes, it is tentatively committed at its client's machine [GKLS94, GBH$^+$96]; when the client reconnects with a server, each tentatively committed transaction is committed if it passes validation checks at the server. Treating these transactions as running transactions can be unnecessarily restrictive. For example, EPL-2 specifications for running transactions disallow reads from other running transactions. When a mobile client is disconnected, it would like to observe its own updates and may even want to read the updates of other nearby clients, i.e., running and tentatively committed transactions are allowed to read from other tentatively committed transactions [TTP$^+$95]. Thus, for mobile systems, we need to divide the category of running transactions into two — transactions that have not reached the commit point and transactions that are tentatively committed; our EPL definitions can be used for the former type of transactions.

This thesis presents efficient and scalable optimistic mechanisms for data-shipping client-server systems. It is unclear which techniques are the best for other system architectures. An important direction of future research is to design and evaluate the performance of concurrency control mechanisms for different types of environments. For example, in a pure function-shipping system, our optimistic techniques may not be the most appropriate. Our schemes take advantage of the client's processing power and offload most of the work from servers to clients; for this reason, the cost of aborts is not very high in our implementations. In pure function-shipping systems, the restart costs can be much higher since an abort would slow down all clients rather than just the client whose transaction aborted. However, optimism may be appropriate in systems that use a combination of data and function-shipping. For example, in a hybrid system where the client performs most of the transaction's work and short queries are executed at a server, our optimistic mechanisms may perform well. Similarly, optimism may be a good approach for three-tier systems, in which servers store persistent objects, clients run applications, and transactions are executed at shared "proxies". These are essentially hybrid systems in which proxies fetch data from the servers and functions are shipped by the clients to the proxies. In such systems, optimism may be the appropriate concurrency control mechanism for managing data among the proxies and the server. Locking may be the best approach to manage the client cache relative to the proxies. In general, hybrid systems require more investigation since a combination of concurrency control techniques may be appropriate for such systems.

This thesis also shows how multistamps can be used capture different types of conflicts efficiently in distributed client-server systems. It would be interesting to use the ideas presented in this thesis for developing efficient consistency mechanisms for other environments such as replicated, mobile, cooperative caching systems (where clients communicate directly with each other for servicing cache misses), and even non-transactional systems. For example, in a replicated system, multistamps may be used for maintaining weakly-consistent replicas at different consistency levels. Similarly, some of the existing non-transactional systems that use multistamps for providing causality guarantees can benefit from our idea of multistamp truncation.

This thesis has shown that the cost of providing serializability in data-shipping client-server systems is not high. It is possible that serializability is inexpensive for some of the common workloads in systems with a different architecture, e.g., pure function-shipping systems, hybrid systems, and three-tier systems. We believe that more studies for evaluating isolation mechanisms are needed since the productivity cost of programming at lower isolation levels can be significant. Quantifying the overheads of serializability for different access patterns is valuable to programmers since they can determine if any significant performance benefit can be achieved for their application by running transactions at lower isolation levels. Such studies are crucial for system developers as well: the performance results can provide insights into the overheads of strong consistency mechanisms thereby allowing database implementors to optimize the system for better performance.

Appendix A

Specifications of Intermediate Levels for Executing Transactions

In Chapter 3, we discussed how different isolation levels for executing transactions can specified similar to the approach used for committed transactions. For the sake of completeness, we now give the specifications of intermediate levels for executing transactions. For all these levels, dirty reads at runtime (phenomenon P1) are disallowed. We only provide guarantees to reads of an executing transaction T_i (recall from Section 3.5 that T_i's predicate-based writes are treated as predicate-based reads). As discussed in Chapter 3, if an executing transaction T_i in history H is provided level *L* during its execution, it must be provided a level at least as strong as *L* when it commits. We also assume that committed transactions in history H are being provided with (at least) level *L* guarantees. It is possible to consider systems in which execution-time guarantees are stronger than commit-time guarantees. Consistency conditions for these systems can be developed in a manner similar to the approach used for committed transactions in mixed systems (Section 3.3). For simplicity, we only consider systems where the commit-time guarantees are at least as strong the execution-time guarantees.

Isolation Level EPL-2+

Level EPL-2+ ensures that an executing transaction observes a consistent committed database state.

This level is specified using the Direct Transaction Graph or DTG presented in Section 3.5.3. The DTG is specified for a history H and an executing transaction T_i (for which isolation guarantees are being provided) and is denoted by DTG(H, T_i). The DTG is exactly the same as DSG with the following addition: it also contains a node for T_i, and all edges corresponding to T_i's reads. EPL-2+ disallows P1 and E-single:

> **E-single: Single Anti-dependency Cycles at Runtime.** A history H and an executing transaction T_i exhibit phenomenon E-single if DTG(H, T_i) contains a directed cycle involving T_i with exactly one anti-dependency edge.

Disallowing phenomenon E-single is equivalent to the no-depend-misses property with respect to transaction T$_i$'s reads, i.e., if T$_i$ depends on a transaction T$_j$, T$_i$ does not miss the effects of T$_j$. This equivalence can be shown in manner similar to Theorem 2+ given in Section 4.1.1.

Isolation Level EPL-3U

Isolation level EPL-3U ensures that the reads of an executing transaction T$_i$ are serializable with all update transactions that have committed.

This level disallows phenomena P1 and E-update (the function *update-transactions(H)* selects only operations executed by update transactions in H):

> **E-update: Single Anti-Dependency Cycles with Update Transactions at Run-time.** A history H and an executing transaction T$_i$ exhibit phenomenon E-update if DTG(update-transactions(H), T$_i$) contains a cycle involving T$_i$ that consists of dependency edges and 1 or more anti-dependency edges.

Similar to the case for G-update, disallowing E-update is equivalent to the no-update-conflict-misses property with respect to T$_i$'s reads. Recall that EPL-3 ensures that T$_i$ observes a serializable state of the database. Isolation level EPL-3U provides lower guarantees than EPL-3 since it only ensures that the observed state is serializable with respect to update transactions only, i.e., the addition of a read-only transaction along with T$_i$ in the DTG may result in a cycle.

Isolation Level EPL-CS

Cursor Stability is an isolation level designed to disallow lost updates. Since this level provides extra guarantees over PL-2 with respect to a transaction's modifications and our execution-time guarantees are provided for reads, we do not need to disallow any phenomenon other than P1 for level EPL-CS.

Isolation Level EPL-2L

At level EPL-2L, a running transaction observes a monotonically increasing prefix of the committed history. To define EPL-2L, we use a modified form of the USG called the *Unfolded Transaction Graph* or *UTG* for a history H and an executing transaction T$_i$. The graph UTG is the same as USG except that the UTG only contains read nodes (and no write nodes) due to T$_i$; since T$_i$'s predicate-based writes are considered as predicate-based reads, nodes and edges corresponding to such "reads" are also added. We define a phenomenon, E-monotonic, that is analogous to G-monotonic; T$_i$ executes at level EPL-2L if phenomena P1 and E-monotonic do not occur:

E-monotonic: Monotonic Reads at Runtime. A history H and an executing transaction T_i exhibit phenomenon E-monotonic if there is a cycle in USG(H, T_i) containing exactly one anti-dependency edge from a read node $r_i(x_j)$ (or r_i(P: x_j, ...)) to some transaction node T_k (*and* any number of order or dependency edges).

Isolation Level EPL-SI

At level EPL-SI, an executing transaction observes a snapshot of the committed database state.

To define EPL-SI, we use a variation on the SSG called the *Start-ordered Transaction Graph* or the *STG*. The STG is the same as the SSG with the following change: it also contains a node for executing transaction T_i, and all edges corresponding to T_i's reads (and as in the DTG, we treat predicate-based writes as predicate-based reads). Like level PL-SI, we define two phenomena that are analogous to G-SIa and GSI-b:

E-SIa: Interference at Runtime. A history H and an executing transaction T_i exhibit interference E-SIa if STG(H,T_i) contains a read-dependency edge from T_j to T_i without there also being a start-dependency edge from T_j to T_i.

E-SIb: Missed Effects at Runtime. A history H and transaction T_i exhibits phenomenon E-SIb if STG(H, T_i) contains a directed cycle involving T_i with exactly one anti-dependency edge.

Phenomenon E-SI consists of E-SIa and E-SIb; level EPL-SI disallows P1 and E-SI.

Isolation Level EPL-FCV

At level EPL-FCV, an executing transaction T_i observes a consistent state of the database such that all T_i does not miss the effects of any transaction that committed before T_i's start point. This level disallows phenomena P1 and ESI-b.

Level	Name	Conditions
EPL-2	Committed Reads	P1
EPL-CS	Cursor Stability	P1
EPL-2L	Monotonic View	P1, E-monotonic
EPL-MSR	Monotonic Snapshot Reads	P1, E-MSR
EPL-2+	Consistent View	P1, E-single
EPL-FCV	Forward Consistent View	P1, E-SIb
EPL-SI	Snapshot Isolation	P1, E-SI
EPL-3U	Update Serializability	P1, E-update
EPL-3	Full Serializability	P1, E2

Figure A-1: Intermediate isolation levels for running transactions

Isolation Level EPL-MSR

At this level, an executing transaction T_i's actions observe a monotonically increasing snapshot of the committed state. We use a graph called the *Start-ordered Unfolded Transaction Graph* (SUTG) that contains all committed transactions and read operations of transaction T_i under consideration (as in the UTG for EPL-2L, we add nodes for T_i's predicate-based writes and treat them as predicate-based reads). A phenomenon E-MSR that is similar to G-MSR can be defined on the SUTG and level EPL-MSR as one that disallows P1 and E-MSR.

Figure A-1 summarizes the intermediate isolation levels for executing transactions.

Appendix B

Optimistic Mechanisms for PL-2L, Causality and PL-3

We now discuss some optimistic schemes that provide different consistency and causality guarantees to executing and committed transactions in distributed client-server systems. In Section B.1, we discuss how the PL-2+ scheme can modified to provide PL-2L and EPL-2L. Section B.2 shows how different causality guarantees can be provided to clients; these schemes are also based on the multistamp-constraint mechanism. We also present results that show the cost of providing different levels of causality. Section B.3 presents a technique that allows clients to efficiently commit read-only transactions with serializability guarantees.

B.1 Optimistic Schemes for Levels PL-2L and EPL-2L

The scheme for EPL-2L is exactly the same as the EPL-2+ technique, with consistency stalls as described in Chapter 5, except that invalidations are handled as in PL-2, i.e., if the current transaction T$_i$ receives an invalidation for an object *x* that it has modified, T$_i$ is aborted; if T$_i$ has simply read *x*, it is not aborted. The reason is that EPL-2L simply requires that T$_i$ does not miss a transaction T$_j$'s effects *after* it becomes dependent on T$_j$; it is acceptable for T$_i$ to have missed T$_j$'s effects earlier.

The above scheme validates T$_i$'s reads as it executes and provides EPL-2L. To provide PL-2L, T$_i$'s writes need to be validate as well. For this purpose, servers perform the checks in Weak-CLOCC at commit time.

In our PL-2L scheme, a read-only transaction T$_i$ can be committed immediately at the client machine when it finishes execution. As in PL-2, such transactions are never aborted; invalidations only abort a transaction if it has modified an obsolete object.

B.2 Causality Guarantees

Causality [Lam78] is a guarantee that clients may find useful. In this section, we discuss three levels of causality — no causality, local causality and global causality, and present schemes that provide them to clients. These causality guarantees are independent of the consistency guarantees being provided and can be supported during a transaction's execution or at commit time.

Local causality guarantees are provided to a transaction T_i executing at client C if the system ensures that T_i does not miss the effects of any transaction T_j that ran at C before T_i *and* all the transactions that T_j depends on. Local causality is useful for a transaction T_i at client C since it allows C to make decisions based on data observed in transactions that executed at C before T_i. If transaction T_i misses the effects of T_j or any transaction T_j depends on, we say that *no causality* guarantees are provided. *Global causality* is a property such that if a client C observes the effects of transaction T_j committed by some other client D, it also observes effects of all earlier transactions of client D and the transactions they depended on. Global causality is the stronger than local causality. Note that locking provides global causality to all uncommitted transactions running above (and including) PL-2.

Although it may seem that not providing any causality guarantees is very weak, such guarantees may not be needed for applications where a client's "memory" is not important. For example, if different users use a computer terminal in a public place to execute their queries, there is no reason why a subsequent query needs to observe the effects of a transaction that was committed earlier by the client. Of course, local causality can be useful in many cases. For example, suppose an employee record E (whose manager is M) is added to an employee database and a manager database. Suppose that a client reads E's record and then runs another transaction asking for all employees working for M; we would like to ensure that E is present in the returned result. Local causality would be required to guarantee this result. Global causality is needed when a client wants *its* transactions to be observed in the order it has committed them. For example, consider a stock reporting system where a client C modifies the stock price of a company *A* and in its next transaction closes the market. If another client D observes the market to be closed and reads the price of *A*'s stock, we would like to ensure that D observes the final stock value; thus, global causality is needed in this application.

We now discuss how different levels of causality can be supported while transactions are executing; similar schemes can be used for committed transactions. We build on the scheme that provides EPL-2+ to running transactions.

Local Causality

Local causality is already provided by our EPL-2L and higher isolation degree schemes for running transactions since the REQ array at a client keeps track of the requirements imposed by *all* previously committed transactions and the transactions they depended on. For isolation level EPL-2, here is a modification to the scheme.

A client C maintains an array called PREVREQ array keeps track of the requirements imposed by all transactions that have previously committed at C. As before, the REQ array keeps track of the requirements placed by all transactions including the current transaction. Instead of using REQ, the PREVREQ array is used to perform the read-dependency checks (at commit time or while running, depending on when the causality guarantees are needed). It is initialized using the REQ array at the beginning of a transaction but not updated while the transaction is executing, whereas the REQ is updated as before. The PREVREQ array ensures that a transaction executing at client C does not miss the effects of any transaction T_i that previously ran at C and the transactions that T_i depends on. However, since this array does not include the constraints introduced by the current transaction, the client is not stalled because of the reads it makes in the current transaction (recall that local causality simply requires the transaction to observe the effects of all transactions that have *previously* committed at C and the transactions they depend on).

No Causality

Our mechanism for EPL-2 does not provide any causality guarantees. For schemes that provide EPL-2L or a higher isolation level, the REQ array at a client keeps track of the multistamp constraints imposed by all transactions that have executed at the client; we need to modify these mechanisms so that REQ only contains constraints imposed by the current transaction's reads and not the reads/writes of previous transactions, i.e., the REQ array should be used just for providing the consistency guarantees requested by client C.

The client maintains a table SSTAMP that maps servers to multistamps; every time the client receives a fetch reply from server X, it merges the multistamp in the message into SSTAMP[X]. The array REQ is empty when the transaction starts. The first time the transaction accesses an object from server X, it uses SSTAMP[X] to update entries for other servers used by the transaction in REQ and then sends invalidation requests to these other servers if necessary. After commit, REQ is merged into SSTAMP[X] for all servers X accessed by the current transaction T_i (to capture T_i's dependencies). In this scheme, at any given point, REQ just stores the requirement imposed by the current transaction; it does not take into account the requirements imposed by a previously committed transaction T_j. Thus, T_i can miss the effects of some transactions that T_j depended on. However, note that T_i *cannot* miss T_j's effects even in this scheme; if T_i's client uncaches an object *x* that was modified by T_j and fetches it again from a server X, X will return the latest version of *x* or block T_i's fetch if T_j is still prepared at X.

Global Causality

Global causality is supported by modifying the EPL-2+ scheme in the following way. A client C keeps track of a multistamp, ALLSTAMP that is obtained by merging the multistamps of all earlier transactions that have been committed by C. When a transaction T_i from C commits, the coordinator sends T_i's final multistamp to C in the commit reply; C merges the multistamp into

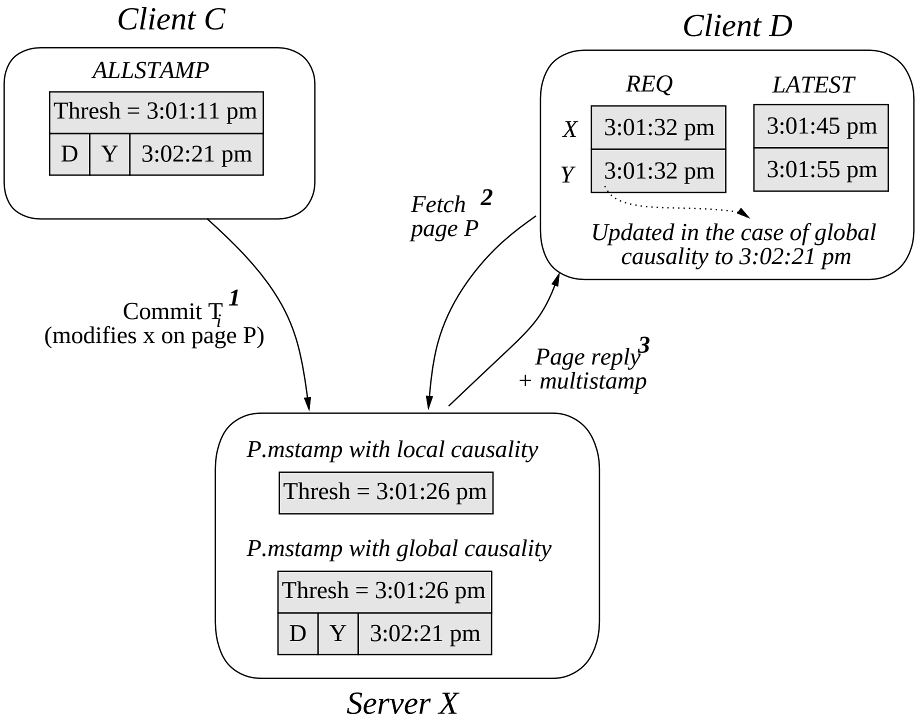


Figure B-1: Global causality can result in extra consistency stalls.

ALLSTAMP [1]. At the end of its next transaction T_j, client C sends ALLSTAMP to T_j's coordinator and the server merges it into T_j's multistamp (i.e., ALLSTAMP is used to "seed" T_j's multistamp); if any transaction T_k reads T_j's effects, T_k cannot miss the updates of any of the earlier transactions committed by client C. As a result, global causality will be provided to T_k.

Figure B-1 shows a case where client C commits a transaction T_i that has modified object *x* on page P. In the case of global causality, we merge C's ALLSTAMP into P's multistamp; for local causality, this merge is not performed. Since P's multistamp has more constraints for global causality, when client D fetches page P from server X, REQ[Y] increases beyond LATEST[Y] causing a consistency stall (assuming D is accessing server Y in the current transaction). In the case of local causality, client D's LATEST array already meets the requirements imposed by P's multistamp; hence, there is no consistency stall.

Performance Cost of Different Causality Guarantees

In this section, we evaluate the cost of providing different levels of causality to clients using the 3/2+/2+ system (the notation for naming systems has been presented in Section 7.1). We have chosen the 3/2+/2+ system since it is the cheapest reasonable system that provides strong isolation guarantees to all types of transactions. We compare the costs of providing different levels of

[1] Note that ALLSTAMP is different from CURRUSTAMP that was presented in Section 5.4.1 since the latter just contains the multistamps of the update transactions that were committed by a client.

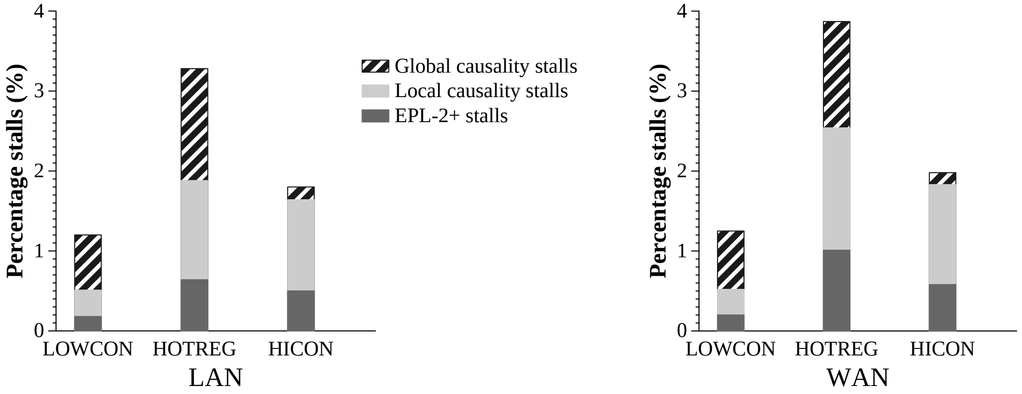


Figure B-2: Breakdown of stall rate for local and global causality.

causality to running transactions for the 3/2+/2+ system; the costs are lower for providing these guarantees at commit time. Note that CLOCC and the 3/3/2+ system provide stronger consistency guarantees than local causality to committed transactions. However, they do not necessarily provide causality guarantees to transactions as they run. We evaluated the cost of providing different causality guarantees to running transactions in the 3/3/2+ system; the results are similar to those for the 3/2+/2+ system.

We observed that the impact of throughput is negligible for providing different levels of causality. Thus, we perform a stall rate analysis to understand the costs of local and global causality.

The stall rates for different levels of causality are given in Figure B-2. The stall rate goes up by a factor of two to three when support for local causality is added to the 3/2+/2+ (or 3/3/2+) systems, and it increases further when support for global causality is added (by less than a factor of two). We now discuss why the stall rate increases in systems with local and global causality.

In a system without local or global causality guarantees, single-server transactions cannot stall, and even in a multi-server transaction, a client's requirements are due only to the servers used in that transaction. Thus, the following situation must occur for a client to stall. Client C fetches page P at server S in a multi-server transaction that involves another server R as well. Furthermore, page P has been modified recently by a multi-server transaction that also involved server R and caused an invalidation for C. The likelihood of a multi-server transaction depending on another "temporally close" multi-server transaction is relatively low because there are 80% single-server transactions in the workload (changes in the stall rate as the percentage of multi-server transactions is varied has been described in Section 7.5.5).

When local causality is supported, there are more stalls since a transaction can stall due to requirements generated in an earlier transaction. For example, without any causality guarantees, single-server transactions never stall but now they can; we observed that with local causality about half the total stalls in HICON and HOTREG were stalls in a single-server transaction. Furthermore, a client C's requirements are no longer due to the servers being accessed in C's current transaction but are due to *all* servers that C has accessed in the past. Since the client's invalidation requirements are higher than in a system without local causality, the stall rate is higher.

The default implementations used in Chapter 7 provide local causality to running or committed transactions. In systems where EPL-2+ or EPL-3U is being provided, the default implementation provides local causality to running transactions; for systems in which PL-2+ or PL-3U is being provided, it guarantees local causality for committed transactions.

Global causality further increases the stall rate because clients now act as "propagators" of multistamps. Without global causality, multistamp information is propagated across servers only through multi-server transactions. Now when a client "switches" servers, it propagates the multistamps generated due to its previous commits to the new servers. Thus, faster propagation causes bigger multistamps resulting in more pruning and hence, a higher number of stalls.

Of course, despite the increase in stalls, the stall rate is still very low and the performance degradation due to local or global causality is minimal. Thus, even global causality can be supported at a low performance cost.

B.3 Efficient Serializability for Read-only Transactions

We now discuss a simple extension to CLOCC that allows read-only transactions to be committed at client machines with PL-3 guarantees. In a single-server system using CLOCC (i.e., all data is stored on one server), a read-only transaction T_q can always be committed by a client locally because invalidation messages are generated in a transaction-consistent manner and the arrival order at the server can be used as the serialization order (no timestamps are needed). The PL-3U scheme also uses arrival order, but without coordinating this order across servers. As a result, it does not provide serializability: two read-only transactions T_q and T_r that commit at client machines are not ordered with respect to each other. CLOCC avoids this problem by ordering all transactions based on their timestamp value. We now present a scheme for committing read-only transactions locally (at client machines) that also provides serializability based on the timestamp order.

For a read-only transaction T_r, suppose that $T_r.maxread$ is the highest timestamp of all the transactions whose effects have been observed by T_r. Transaction T_r's client can commit T_r locally if it can determine that T_r has observed the effects of *all* conflicting transactions that have (or can have) a timestamp less than T_r.maxread; T_r can be serialized at a time slightly higher than T_r.maxread.

To provide serializability without communicating with the servers, we take advantage of the fact that CLOCC maintains a watermark timestamp (see Section 5.1.1); any transaction that tries to validate at a server below the watermark is aborted. To commit T_r locally, its client ensures two conditions. First, it checks that it has received the invalidations of all transactions that have committed at all servers accessed by T_r with a timestamp less than T_r.maxread. Second, it ensures that T_r.maxread is below the watermark of the accessed servers. These tests are called the *read-only checks*. The first condition guarantees that T_r has not missed the effects of any transaction that has committed till now with a timestamp less than T_r.maxread and the second condition ensures that no transaction will commit in the future below T_r.maxread at the relevant servers. Thus, T_r can be serialized at a time slightly higher than T_r.maxread. If the read-only checks fail at T_r client, it must communicate with the servers to validate T_r.

To implement the above strategy, a client maintains an array WMARK, where WMARK[X] stores the last known watermark from server X. In each invalidation message, the server sends its current watermark as well (if there is a prepared transaction below the watermark, the server includes its invalidations as well). For each page P, a server maintains a time *P.modts* that is the maximum timestamp of the transactions that have modified P; this value is also sent to the client in a fetch reply. At the end of a read-only transaction T_r, its client computes T_r.maxread by taking the maximum value of P.modts, for all pages P accessed by T_r. Optimizations similar to the ones designed for the PSTAMP table can be used such that P.modts is only maintained for recently modified pages at the client and the server.

In this scheme, the likelihood that the read-only check fails for transaction T_r is dependent on the relative values of T_r.maxread and the watermarks of different servers accessed by T_r. A low value of server watermarks and a high value of T_r.maxread can cause the read-only check to fail. As discussed in Section 5.1.1, the watermark is set at least δ below a server's current time, where δ is the network delay. Thus, when δ is high (e.g., in a WAN environment) *and* T_r reads from a recently committed transaction, the read-only check is likely to fail. In LAN environments, where delays are low, the watermark can be kept reasonably close to the current time and the read-only check will usually succeed at the client.

Note that reading from a recently committed transaction also implies a workload with moderate to high-contention since two transactions are conflicting on the same object close to each other in real time. Thus, for the common case of low-contention workloads and for LAN environments, the above scheme may actually suffice. This scheme also has the virtue of being very simple with very low overheads in terms of data structures and their manipulation.

The above scheme can be modified for providing EPL-3 to running transactions (a multistamp-based scheme for EPL-3 was presented in Section 5.4.2. At any given point, the client must ensure that a running transaction T_i.maxread is always greater than the watermark values of the accessed servers. Of course, since this condition has to be true while T_i is executing, the communication costs may be higher or T_i may block at a server more often than in the committed transaction case.

Bibliography

[A^{+}76] M. Astrahan et al. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976. Also available in Chapter 1 of Readings in Database Systems, Third Edition, Morgan Kaufmann, 1998.

[AAS93] D. Agrawal, A. E. Abbadi, and A. K. Singh. Consistency and Orderability: Semantics-Based Correctness Criteria for Databases. *ACM Transactions on Database Systems (TODS)*, 18(3):460–486, Sept. 1993.

[ABGS87] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed Multi-version Optimistic Concurrency Control with Reduced Rollback. *Distributed Computing*, 2(1):45–59, 1987.

[ABJ97] V. Atluri, E. Bertino, and S. Jajodia. A Theoretical Framework for Degrees of Isolation in Databases. *Information and Software Technology*, 39(1):47–53, 1997.

[ACD^{+}96] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[ACL^{+}97] A. Adya, M. Castro, B. Liskov, U. Maheshwari, and L. Shrira. Fragment Reconstruction: Providing Global Cache Coherence in a Transactional Storage System. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, Baltimore, MD, May 1997.

[Ady94] A. Adya. Transaction Management for Mobile Objects Using Optimistic Concurrency Control. Master's thesis, Massachusetts Institute of Technology, Jan. 1994. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-626.

[AGLM95] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control using Loosely Synchronized Clocks. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 23–34, San Jose, CA, May 1995.

[AL97] A. Adya and B. Liskov. Lazy Consistency Using Loosely Synchronized Clocks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 73–82, Santa Barbara, CA, Aug. 1997.

[ALBL91] T. Anderson, H. Levy, B. Bershad, and E. Lazowska. The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, Santa Clara, CA, April 1991.

[ANK$^+$95] M. Ahamad, G. Neiger, P. Kohli, J. Burns, and P. Hutto. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing*, 9(1):37–49, 1995.

[ANS92] *ANSI X3.135-1992, American National Standard for Information Systems – Database Language – SQL*, November 1992.

[BBG$^+$95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 1–10, San Jose, CA, May 1995.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[BK91] N. S. Barghouti and G. E. Kaiser. Concurrency Control in Advanced Database Applications. *Computing Surveys*, 23(3):269–317, September 1991.

[BK96] J. Basu and A. Keller. Degrees of Transaction Isolation in SQL*Cache: A Predicate-based Client-side Caching System, May 1996. Available at http://www-db.stanford.edu/pub/keller/.

[BOS91] P. Butterworth, A. Otis, and J. Stein. The GemStone Object Database Management System. *Comm. of the ACM*, 34(10):64–77, October 1991.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[CALM97] M. Castro, A. Adya, B. Liskov, and A. Myers. HAC: Hybrid Adaptive Caching for Distributed Storage Systems. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 102–115, St. Malo, France, Oct. 1997.

[CDN93] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington D.C., May 1993.

[CFZ94] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 359–370, Minneapolis, MN, June 1994.

[CG85] A. Chan and R. Gray. Implementing Distributed Read-Only Transactions. *IEEE Transactions on Software Engineering*, 11(2):205–212, Feb. 1985.

[CO82] S. Ceri and S. Owicki. On the Use of Optimistic Methods for Concurrency Control in Distributed Databases. In *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 117–129, Asilomar, CA, Feb. 1982.

[CR94] P. Chrysanthis and K. Ramamritham. Synthesis of Extended Transaction Models using ACTA. *ACM Transactions on Database Systems (TODS)*, 19(3):450–491, Sept. 1994.

[CS89] W. W. Chang and H. J. Schek. A Signature Access Method for the Starburst Database System. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 145–153, Amsterdam, Netherlands, August 1989.

[Dat90] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, Fifth edition, 1990.

[DD77] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Comm. of the ACM*, 19(11):624–633, November 1976. Also published as IBM RJ1487, December, 1974.

[FCAB98] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *Proceedings of ACM SIGCOMM'98*, pages 254–265, Vancouver, Canada, Feb. 1998.

[FCL97] M. J. Franklin, M. J. Carey, and M. Livny. Transactional Client-Server Cache Consistency: Alternatives and Performance. *ACM Transactions on Database Systems*, 22(3):315–363, Sept. 1997.

[GBH$^+$96] J. Gray, J. G. Bennett, P. Helland, P. E. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, May 1996.

[Ghe95] S. Ghemawat. The Modified Object Buffer: A Storage Management Technique for Object-Oriented Databases. Technical Report MIT/LCS/TR-666, MIT Laboratory for Computer Science, Sept. 1995.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Canada, June 1996.

[GKLS94] R. Gruber, F. Kaashoek, B. Liskov, and L. Shrira. Disconnected Operation in the Thor Object-Oriented Database System. In *Proc. of IEEE Workshop on Mobile Computing Systems and Applications*, Dec. 1994.

[GKM96] C. Gerlhof, A. Kemper, and G. Moerkotte. On the Cost of Monitoring and Reorganization of Object Bases for Clustering. *SIGMOD Record*, 25(3):22–27, September 1996.

[GLPT76] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Database. In *Modeling in Data Base Management Systems*. Amsterdam: Elsevier North-Holland, 1976. Also available in Chapter 3 of *Readings in Database Systems, Second Edition*, M. Stonebraker Editor, Morgan Kaufmann, 1994.

[GM83] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[GR93] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1993.

[Gru89] R. Gruber. Optimistic Concurrency Control for Nested Distributed Transactions. Technical Report MIT/LCS/TR-453, MIT Laboratory for Computer Science, Cambridge, MA, 1989.

[Gru97] R. Gruber. *Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases*. PhD thesis, M.I.T., Cambridge, MA, 1997.

[GW82] H. Garcia and G. Weiderhold. Read-Only Transactions in a Distributed Database. *ACM Transactions Database Systems*, 7(2):209–234, June 1982.

[Hae84] T. Haerder. Observations on Optimistic Concurrency Control Schemes. *Information Systems*, 9(2):111–120, June 1984.

[Her90] M. P. Herlihy. Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. *ACM Transactions on Database Systems*, 15(1):96–124, March 1990.

[HP86] R. C. Hansdah and L. M. Patnaik. Update Serializability in Locking. In *International Conference on Database Theory*, pages 171–185, Rome, Italy, Sept. 1986.

[IBM99] IBM Corporation. IBM DB2 Version 5.2 Universal Database: Administration Guide, Chapter 10, 1999.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of 19th Int'l Symp. on Computer Architecture*, pages 13–21, Queensland, Australia, May 1992.

[KR81] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[KS91] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Thirteenth ACM Symposium on Operating Systems Principles*, pages 213–225, Asilomar Conference Center, Pacific Grove, CA., Oct. 1991.

[KSS97] H. Korth, A. Silberschatz, and S. Sudarshan. *Database System Concepts*. McGraw Hill, 1997.

[LAC$^+$96] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 318–329, Montreal, Canada, June 1996.

[Lam76] L. Lamport. Towards a Theory of Correctness for Multi-user Data Base Systems. Report CA-7610-0712, Mass. Computer Associates, Wakefield, MA, Oct. 1976.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. of the ACM*, 21(7):558–565, July 1978.

[LCJS87] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 111–122, Austin, TX, November 1987.

[Lis93] B. Liskov. Practical Uses of Synchronized Clocks in Distributed Systems. *Distributed Computing*, 6(4):211–219, Aug. 1993.

[LLSG92] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

[LSWW87] B. Liskov, R. Scheifler, E. Walker, and W. Weihl. Orphan Detection. Programming Methodology Group Memo 53, Laboratory for Computer Science, MIT, Cambridge, MA, February 1987.

[LW84] M.-Y. Lai and W. K. Wilkinson. Distributed Transaction Management in Jasmin. In *Proceedings of the 10th International Conference on Very Large Data Bases*, pages 466–470. Singapore, Aug. 1984.

[MH86] M. McKendry and M. Herlihy. Time-Driven Orphan Elimination. In *Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems*, pages 42–48. IEEE, January 1986.

[Mil92] D. L. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. Network Working Report RFC 1305, March 1992.

[Mil96] D. L. Mills. The Network Computer as Precision Timekeeper. In *Proc. of Precision Time and Time Interval (PTTI) Applications and Planning Meeting*, pages 96–108, Reston, VA, Dec. 1996. Also available at http://www.eecis.udel.edu/ mills/ntp.htm.

[MK94] W. J. McIver and R. King. Self Adaptive, On-Line Reclustering of Complex Object Data. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 407–418, Minneapolis, MN, May 1994.

[Moh90] C. Mohan. Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems. In *Proceedings of the 16th VLDB Conference*, pages 406–418, Brisbane, Australia, Aug. 1990.

[Mye99] A. C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, Jan. 1999.

[Ngu88] T. Nguyen. Performance Measurement of Orphan Detection in the Argus System. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1988.

[O'N86] P. O'Neil. The Escrow Transactional Method. *ACM Transactions on Database Systems (TODS)*, 11(4):405–430, Dec. 1986.

[Ora95] Oracle Corporation. Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7, July 1995.

[Ous90] J. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proc. of USENIX Summer Conference*, pages 247–256, Anaheim, CA, June 1990.

[OVU98] M. T. Ozsu, K. Voruganti, and R. Unrau. An Asynchronous Avoidance-based Cache Consistency Algorithm for Client Caching DBMSs. In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 440–451, New York, NY, Aug. 1998.

[Pap79] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[PST$^+$97] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, St. Malo, France, Oct. 1997.

[Ree78] D. P. Reed. Naming and Synchronization in a Decentralized Computer System. Technical Report MIT/LCS/TR-205, Laboratory for Computer Science, MIT, Cambridge, MA, 1978.

[Reu82] A. Reuter. Concurrency on High-Traffic Data Elements. In *Proc. of ACM Symposium on Principles of Database Systems*, pages 83–92, Los Angeles, CA, Mar. 1982.

[RKS93] R. Rastogi, H. F. Korth, and A. Silberschatz. Strict Histories in Object-Based Database Systems. In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 288–299, Washington, D.C., May 1993.

[RT90] E. Rahm and A. Thomasian. A New Distributed Optimistic Concurrency Control Method and a Comparison of its Performance with Two-Phase Locking. In *Proceedings of Tenth International Conference on Distributed Computing Systems*, Paris, France, May 1990.

[SBCM95] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-Phase Commit Optimizations in a Commercial Distributed Environment. *Distributed and Parallel Databases*, 3(4):325–360, Oct. 1995.

[Sea99] Seagate Technology Inc. Cheetah 9LP disk drive, http://www.seagate.com/, Feb 1999.

[SS84] P. Schwarz and A. Spector. Synchronizing Shared Abstract Types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.

[SS96] M. Spasojevic and M. Satyanarayanan. An Empirical Study of a Wide-Area Distributed File System. *ACM Transactions on Computer Systems*, 14(2):200–222, May 1996.

[Sta89] J. W. Stamos. A Low-Cost Atomic Commit Protocol. Tech. Report RJ7185, IBM Almaden, CA, December 1989.

[SWY93] H.-J. Schek, G. Weikum, and H. Ye. Towards a Unified Theory of Concurrency Control and Recovery. In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 300–311, Washington, D.C., May 1993.

[TN92] M. Tsangaris and J. F. Naughton. On the Performance of Object Clustering Techniques. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 144–153, San Diego, CA, June 1992.

[TPC92] TPC: Transaction Processing Performance Council. TPC-A Standard Specification, Revision 1.1, TPC-C Standard Specification, Revision 1.0. http://www.tpc.org, 1992.

[TTP$^+$95] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, and M. J. Spreitzer. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. 15th ACM Symp. on Operating System Principles (SOSP),* pages 172–183, Copper Mountain Resort, CO, Dec. 1995.

[vEBBV95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. 16th ACM Symp. on Operating System Principles (SOSP),* pages 40–53, Copper Mountain Resort, CO, Dec. 1995.

[Wei87] W. E. Weihl. Distributed Version Management for Read-only Actions. *IEEE Transactions on Software Engineering*, SE-13(1):55–64, January 1987.

[WL85] W. Weihl and B. Liskov. Implementation of Resilient, Atomic Data Types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, April 1985.

[YBS91] M. R. Y. Breitbart, D. Georgakopoulos and A. Silberschatz. On Rigorous Transaction Scheduling. *IEEE Transactions on Software Engineering*, 17(9):954–960, Sept. 1991.

[ZCF97] M. Zaharioudakis, M. J. Carey, and M. J. Franklin. Adaptive, Fine-Grained Sharing in a Client-Server OODBMS: A Callback-Based Approach. *ACM Transactions on Database Systems*, 22(4):570–627, Dec. 1997.