TRANSACTION SYNCHRONIZATION IN KNOWLEDGE BASES:
Concepts, Realization and Quantitative Evaluation

by

Vinay K. Chaudhri

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Transaction Synchronization in Knowledge Bases:
Concepts, Realization and Quantitative Evaluation

by
Vinay K. Chaudhri
Doctor of Philosophy, 1995
Department of Computer Science
University of Toronto

Abstract

Large knowledge bases that are intended for applications such as CAD, corporate repositories or process control will have to be shared by multiple users. For these systems to scale up, to give acceptable performance and to exhibit consistent behavior, it is mandatory to synchronize user transactions using a concurrency control algorithm. The transactions in knowledge bases often access a large number of entities and perform complex inferences that may last for a long period of time. In such a situation, using conventional concurrency control methods, which require a transaction to hold its locks until it has acquired all the locks it will ever need, do not lead to good performance. This thesis examines the problem of concurrency control for such long transactions in a knowledge base setting.

Using a directed graph as a general model of a knowledge base, we develop an algorithm, called the Dynamic Directed Graph (DDG) policy, that allows release of locks by a transaction before it has acquired all the locks that it will ever need. Furthermore, it deals with a knowledge base whose graph may contain cycles and may receive insertions and deletions of nodes and edges. We develop a theory of correctness for schedules in such databases and use it to prove the correctness of the DDG policy and two other locking policies. In addition, we analyze the well-structured-ness and the deadlock-freedom of the DDG policy. The thesis also includes the design of a prototype implementation of the DDG policy and performance results for three real knowledge bases. These results show that the DDG policy can be implemented without unacceptably high overhead, and for the workloads found in "real-world" knowledge bases, the proposed algorithm can indeed perform better than the conventional methods, such as two-phase locking.

Acknowledgments

I am grateful to my adviser Prof. John Mylopoulos for accepting me as a student, nurturing my graduate student career by offering the best of the opportunities, always maintaining a pragmatic research direction, giving freedom to pursue my interests and for setting up an intellectual framework to conduct an exciting research program.

I am indebted to my co-adviser Prof. Vassos Hadzilacos for patiently guiding me through the theoretical aspects of concurrency control. I would not have been able to develop the right framework, identify relevant theorems and construct the correctness proofs without Vassos's help. In addition, his numerous painstaking readings of the drafts of the thesis substantially improved the quality of the work.

I am grateful to Prof. Ken Sevcik for his contributions to the performance analysis part of my thesis. Our endless arguments on several of the basic assumptions that are taken for granted in the database performance community put this work on a solid foundation. Ken's strong support for the thesis topic combined with the pleasant interactions we had were instrumental in making it easy for me to progress through the graduate program.

I am thankful to the members of the thesis committee, Prof. Peter Boulton, Prof. Mark Chignell, Prof. Hector Levesque and Prof. Alberto Mendelzon for reading the thesis, giving constructive suggestions and for offering perspectives that I had not thought of. I would also like to thank the external reader Prof. Dennis Shasha for his appraisal of this work.

I would like to express special thanks to Prof. G. Scott Graham for his significant help with the simulation methodologies. I also appreciated the opportunity that he and Prof. Sevcik gave me to participate in teaching courses on performance analysis that helped me in developing a strong background for completing this thesis.

Prof. Mike Carey and Prof. Miron Livny of University of Wisconsin were extremely generous by making available some of their software. Mike was lighteningly prompt in answering questions about the fine details of his work and in explaining many assumptions in performance analysis.

Thousands of machine cycles that I burnt in the performance analysis were made available on their machines by Prof. Songnian Zhou and Prof. Dave Wortman. Ben Gamsa and Chris Siebenmann were forgiving every time I filled up the disk or hogged a workstation. Jingwen Wang deserves special thanks for taking care of the load sharing problems the minute they arose.

The work on performance analysis benefited significantly by using three real knowledge base applications. These applications were provided by Dr. Huaiqing Wang and Dr. Bryan Kramer (APACS), Deborah L. McGuinness (Wines) and John Chionglo (TOVE).

I wish to thank Prof. Mike Carter and Prof. Burhan Turksen for their help in the initial phases of my Ph.D. program. Mike was especially supportive by giving both administrative and academic help in what I consider some of my lowest moments.

I am grateful to Prof. Russ Greiner and Prof. Charles Elkan for the interactions I had with them in the first year of my graduate program here. Russ initiated me into the field of knowledge representation and gave an excellent start to my graduate work. Numerous discussions with Charles were instrumental in defining my research directions. Also quite influential were Arvind Gupta and Bart Selman who answered many of my questions when I was still trying to figure out who to work with.

I would like to thank my colleagues Thodoros Topaloglou and Manolis Koubarakis for teaching me finer aspects of the language Telos which was the starting point for this work. I was very appreciative of the warm welcome to the group extended by Brian Nixon. Adel Shrufi deserves many thanks for creating opportunities for lively discussions. The other members of the group, Lawerence Chung, Igor Jurisica, David Lauzon, Dimitris Plexousakis, Martin Stanely and Eric Yu made the working environment pleasurable. Thanks to Tony Plate for his help with the typesetting of this version of the thesis.

I had good fortune of having several supportive friends who made non-academic life in Toronto something to cherish. I thank Heiner Belefeldt, Tony Bonner, David and Roxanne Carlislie, Anirudha Chatterjee, Isabel Cruz, Janice Dumont, Sherif Ghali, Salman Ghani, David Gunby, Rob Hanks, Karim Harzalla, Srinivas Kandala, Neil Kennedy, Yves Lesperance, Sheila McIlraith, Arshad Moolia, Glenn Moulaison, Bob Ott, Mark Ozon, Jacob Pabbathy, Javier Pinto, Sridhar Sambatur, Prabir Sarkar, Richard Scherl, Jeff Siskind, Manfred Stede, Anne Saddlemayer, Sue Webb, Karen Williams and Daniel Zilio for their friendship and support. I also wish to thank our administrative staff: Veronica Archibald, Maria Cuda, Norma Dotto, Marina Haloulos, Martha Hendriks and Kathy Yen for all their help over the years.

Financial support was generously made available by University of Toronto and the Department of Computer Science.

I would like to thank my former advisers and mentors Prof. T. P. Bagchi and Prof. N. P. Mehta for the excellent training they gave me in the most formative period of my academic career.

Last but not the least, I would like to thank my parents for permitting me to pursue graduate work. This endeavor would not have been possible without help of my sister Meera and brother-in-law Ashwani who shouldered responsibilities that were mine. It is to them that I dedicate this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

As we build large knowledge bases, it will no longer be viable or desirable to maintain multiple copies of the same knowledge base for each one of its users, nor will it be feasible to restrict access to the knowledge base to one user at a time. Instead, it is expected that multiple users will share a single knowledge base that receives queries and updates and interleaves their execution against the knowledge base, thereby optimizing the deployment of computing resources, both CPU cycles and storage space.

To make the problem more concrete, consider a system using large knowledge base, part of which is stored in primary storage and the rest in secondary storage. If query/update requests from users are processed sequentially, the system will remain idle while waiting for a disk access to complete (see Figure 1.1(a): dark dots and vertical lines correspond to the requests of two different users). However, if requests are processed in an interleaved fashion, i.e., concurrently as suggested in Figure 1.1(b), the idle periods can be reduced resulting in higher system throughput[1] and a faster response time to the user (see Figure 1.2). This improvement in throughput or response time will grow with the number of users until resources available to the system become saturated.[2] Beyond this point, interleaved execution of user requests does nothing to enhance the system performance and may cause it to deteriorate because of the overhead of concurrency control.

In this thesis, we study the problem of concurrency control or transaction synchronization for knowledge bases. In this chapter, we first define a knowledge base and then characterize the concurrency control problems that we address. We give an overview of our approach for solving the problem, a summary of our contributions and then end the chapter with an outline of this thesis.

1.2 What is a Knowledge Base?

Without indulging in a discussion on the philosophical meaning of *knowledge* or a *knowledge base*, we present here three alternative definitions of a knowledge base and then quickly

[1] Measured by the number of user requests executed on average per unit of time.

[2] Concurrent processing can lead to arbitrarily large improvements in performance — sometimes by as much as of the order of *one hundred times* (Gray 1992). Even when there are no disk waits, a round robin scheduling discipline that allows interleaved processing of user requests is a general purpose scheduling algorithm that is commonly used (Lavenberg 1983).

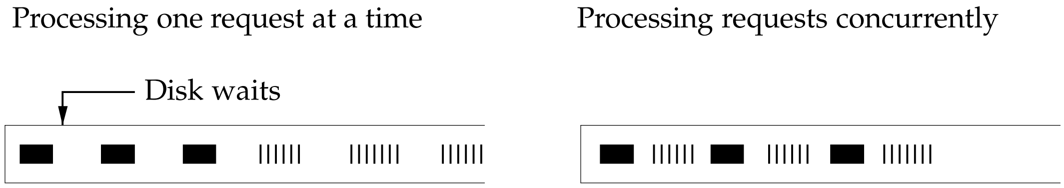


Figure 1.1: Concurrent processing reduces idle times

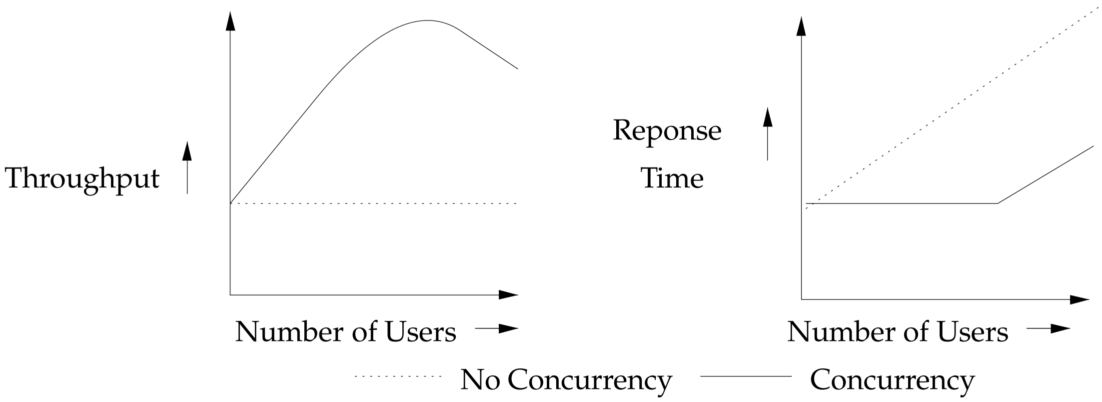


Figure 1.2: Concurrent processing improves performance

highlight the features of the systems considered in this thesis.

In the knowledge representation (KR) community, a widely accepted definition of a knowledge base is as follows (Levesque and Brachman 1985):

> A knowledge base has explicit structures representing the knowledge of the system which determine the actions of the system. It is not the use of a certain programming language or a data-structuring facility that makes a system knowledge-based.

In the database community knowledge bases are perceived with a much narrower perspective. For example, one of the popular definitions for a knowledge base management system (KBMS) is as follows (Ullman 1988):

> A KBMS is a system that provides (1) what a DBMS provides (support for efficient access, transaction management, etc.) and (2) a single, declarative language to serve the roles played by both the data manipulation language and the host language in a DBMS. A knowledge system is one that supports only (2); i.e., it is a programming system with a declarative language.

Interestingly, the above definition is contrary to the view adopted in the KR community where a knowledge base is not defined in terms of the features of the programming environment. Let us now consider a third definition for a knowledge base (Chaudhri et al. 1994):

> A knowledge base is a database that has a rich data model and adopts modeling features from object-oriented, deductive, temporal and spatial data

> models. A knowledge base often possesses complex semantic structure and has rich processing requirements involving sophisticated retrieval and inference procedures.

This definition views a knowledge base as a post-relational database system that provides advanced data modeling abilities as well as richer processing capabilities.

Needless to say, each of the above definitions presents a valid perspective of a knowledge base and contributes to the viewpoint adopted in this thesis. The most important characteristic of a knowledge base for our purposes is explicit representation of knowledge, and therefore, any system that explicitly encodes the knowledge and the semantic structure of an application domain will be called knowledge based. Examples of explicit representation of knowledge are found in data models that are, for example, object-oriented or deductive. This requirement can also be satisfied by many conventional relational database systems, for example, a system representing bill of materials information. In a later section of this chapter, we argue that such systems can usually be viewed as directed graphs. Therefore, from now on, we use the terms *database* and *knowledge base* interchangeably with the understanding that our results are general and applicable to a broad class of applications.

1.3 An Example Knowledge Base

To be more precise, knowledge bases are assumed to support an object-oriented representational framework with an assertional sub-language used for both deductive rules and constraints. They may also support facilities for representing special kinds of knowledge (for example, spatial knowledge, incomplete knowledge, etc.). Examples of such systems are Telos, (Mylopoulos et al. 1990), CLASSIC (Borgida et al. 1989), CANDIDE (Beck, Gala and Navathe 1989) and several other deductive (Naqvi and Tsur 1989) and object-oriented systems (Zdonik and Maier 1989).

As an example, consider a knowledge base that has a class `Employee`, which is a specialization of class `Person`. The class `Employee` has attributes `Name`, `Manager` and `Salary`. Each `Salary` attribute indicates the salary of an employee during a certain time interval (for example, `before 1989`). `Emp01` is an instance of `Employee` and is an identifier for the employee `Adam` whose `Manager` is `John`. The salary of `Adam` takes two different values in the history — `20000` during time interval `I1` and `30000` during time interval `I2`. These intervals in the history, `I1` and `I2`, are left unspecified. There is an integrity constraint that requires that the value of the salary must always increase, and therefore, we can infer that `I1` must precede `I2`. In addition, it is assumed that the knowledge base contains the following deductive rules:

`DR1`: $\forall p/\text{Employee}, \forall s/\text{Salary}, \forall t1/\text{TimeInterval}$
$\quad \texttt{Salary}(p,s)(\texttt{at } t1) \wedge \texttt{GreaterThan}(s, 25000) \Rightarrow \texttt{WellPaid}(p)\ (\texttt{at } t1)$
`DR2`: $\forall p/\text{Employee}, \forall t1, t2/\text{TimeInterval}$
$\quad \texttt{WellPaid(p)}\ (\texttt{at } t2) \wedge \texttt{StartsBefore}(t2, t1) \Rightarrow \texttt{WellPaid}(p)\ (\texttt{at } t1)$

The first rule states that if an employee's salary is over 25000 then the employee is well-paid. The second rule asserts that if an employee is well-paid during a time interval $t2$, she remains well-paid during any time interval that starts after $t2$. A part of this knowledge base can be seen as a labeled directed graph as seen in Figure 1.3. From now on, we refer to this knowledge base as KB_1.

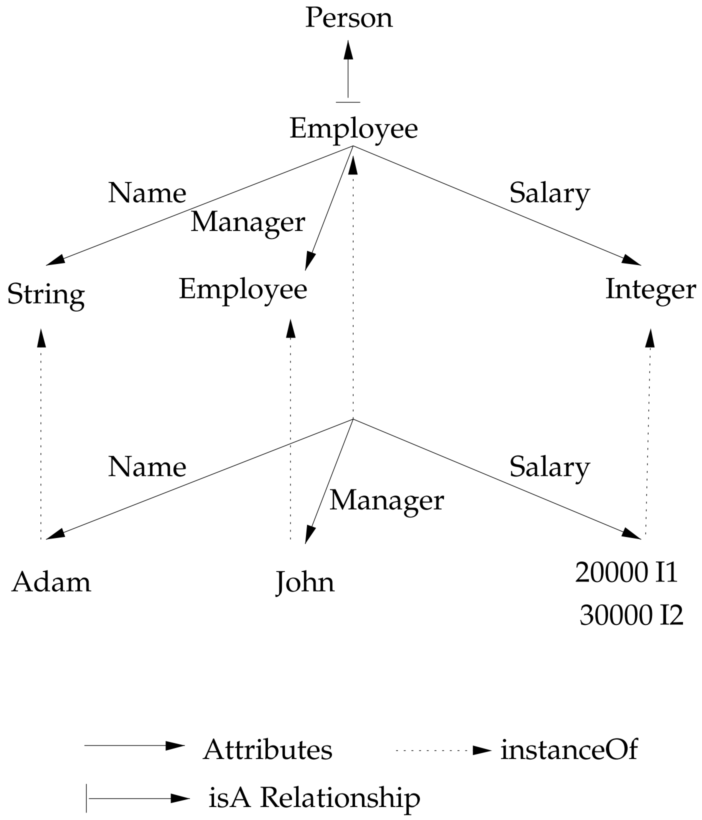


Figure 1.3: The knowledge base KB_1

1.4 Perils of Arbitrary Interleaving

Suppose two users want to simultaneously access KB_1. The first user, called T_1, wants to change the definition of `DR1` to `DR3`, asserting that if an employee is a manager then she is well-paid. If `Manager`(p, q) represents that q is a manager of p, then this rule may be stated as follows:

$$\texttt{DR3: } \forall p, q/\textsf{Employee}, \forall t1/\textsf{TimeInterval}$$
$$\texttt{Manager}(p, q)(\texttt{at } t1) \Rightarrow \texttt{WellPaid}(q)\ (\texttt{at } t1)$$

The second user, called T_2, wants to find all `WellPaid` employees during the time interval `I2`.

T_1 and T_2 will perform these operations using `UNTELL,` `TELL` and `ASK` commands (Mylopoulos et al. 1990). Consider the following interleaving of their operations:

1. T_1 executes an `UNTELL` removing `DR1` from the knowledge base.

2. T_2 executes an `ASK` to find all `WellPaid` employees. Currently, the knowledge base contains only `DR2`. Therefore, T_2 gets an answer that there are no `WellPaid` employees.

3. T_1 completes its job by a `TELL` command and adds `DR3`.

If we execute the operations of T_1 and T_2 without any interleaving and in the order T_1 after T_2, we get the answer `Adam`. If the order is T_2 after T_1, we get the answer `John`. The answer returned in the above execution does not correspond to a state of the knowledge base that was intended by the user, and therefore, such an interleaving is *incorrect*. The notion of *correctness* of concurrent executions has been formalized through the concept of *serializability* (Eswaran et al. 1976; Bernstein et al. 1978; Papadimitriou 1979; Gray and Reuter 1993). A *transaction* is the execution of a user program on a knowledge base. Two executions of the same set of transactions are *equivalent* if they have the same operations, they leave the knowledge base in the same state, and if each operation returns the same value in both executions. An interleaved execution of transactions is *serializable* if it is equivalent to some serial execution of the same collection of transactions. The preceding example shows that not all concurrent executions of transactions are serializable. The mechanism that controls the order in which the operations of concurrent transactions are processed, so that the overall execution is serializable, is called the *concurrency control algorithm* or *policy*.

We would like to emphasize that in database concurrency control we are interested in parallel execution of the operations of different users as opposed to parallelizing the execution of one user transaction. The latter problem has been considered by several other researchers (Raschid, Sellis and Lin 1988; Filman 1989; Ishida, Yokoo and Gasser 1990; Miranker 1991). The next sub-section describes a standard concurrency control algorithm from databases and identifies the problems in applying it to knowledge bases directly.

1.5 Problem in Using Conventional Concurrency Control Algorithms

There is a vast body of literature on concurrency control algorithms (Papadimitriou 1986; Bernstein, Hadzilacos and Goodman 1987; Gray and Reuter 1993). There are three broad classes of such algorithms: locking, timestamps and serialization graphs. For each of these classes, there are variations based on multiple versions and certifiers. Locking-based algorithms have been most successful in practice and their performance is better understood. They also have special solutions for graph structures — the abstraction of knowledge bases that appears to be the most appealing. Therefore, we have adopted the locking class of methods for knowledge bases.

Most commercial database systems use a concurrency control algorithm called two-phase locking. This is a general-purpose algorithm that does not make any assumptions about the underlying data. Two-phase locking (2PL) (Eswaran et al. 1976) works as follows:

> **TP1.** Associated with each data item is a distinct "lock". A transaction must acquire a lock on a data item before accessing it.
> **TP2.** While a transaction holds a lock on a data item, no other transaction may access that data item.
> **TP3.** A transaction cannot acquire any additional lock once it has released some lock (hence the name two-phase locking).

In a simple generalization, a transaction may acquire two types of locks: *shared* and *exclusive*. An item that is only read needs to be locked in shared mode, and item that is written is locked in exclusive mode. Two transactions may simultaneously hold a lock on the same data item only if both lock it in the shared mode.

It can be shown that 2PL ensures serializability. In the example of the previous section, T_1 will lock `WellPaid` before changing its definition so that T_2 will not be able to read the partially updated value. Executing transactions under 2PL prevents incorrect execution.

If a transaction must acquire a lock (because of the rule TP1), but cannot do so (because of the rule TP2), it must wait until the transaction that owns that lock releases it. It is easy to construct scenarios in which locks are acquired in such a manner that a *deadlock* arises (Yannakakis 1982b): a cyclical sequence of transactions each waiting for the next to release a lock it must acquire. Such deadlocks may be resolved by choosing one of the transactions, aborting it (i.e., undoing any effects it had on the knowledge base state), releasing its locks and restarting it at a later time.

The time a transaction T acquires its last lock is known as T's *locked point*. 2PL requires that a transaction must hold all its locks until it has reached its locked point. In order to make recovery from failures easier, most commercial systems use a version of 2PL, known as *strict* 2PL, in which all the locks must be held until the transaction commits (Bernstein, Hadzilacos and Goodman 1987). This has serious performance implications for the type of transactions likely to be applied to knowledge bases since they often access a large number of data items. For example, in the knowledge base KB_1, while proving a goal through backward chaining, a transaction is likely to access all the items that are below that goal in the inference graph, potentially a set including all deductive rules. In such situations, if we use strict 2PL, transactions will end up locking large portions of the knowledge base for long periods of time, thus significantly reducing the concurrency. (For the rest of the thesis, we consider only strict 2PL unless stated otherwise.)

The above noted problem has been known in the literature as the *long-lived transaction* (LLT) problem (Gray 1981). In general, a long transaction is caused because transactions may access a large number of objects, perform lengthy computations, pause for input, or because of a combination of these factors. The LLT problem in knowledge bases is primarily due to transactions accessing a large number of entities and performing lengthy computations. In the next section, we review several approaches for dealing with this problem.

1.6 Approaches to Deal With Long Transactions

Detailed surveys of concurrency control methods that go beyond two-phase locking can be found elsewhere (Skarra and Zdonik 1989; Berghouti and Kaiser 1991). In this section, we consider methods that provide more concurrency than two-phase locking, and thus, help in coping with the problem of long transactions.

1.6.1 Relaxing the Consistency Requirements

One class of solutions works by relaxing the serializability requirement of transactions (Garcia-Molina 1983; Lynch 1983; Farrag and Ozsu 1989; Garcia-Molina and Salem 1987). Intuitively, the idea is to split up LLTs into shorter steps that are executed as individual transactions. After each step, locks can be released, and other transactions can then access the resources used by the LLT. The price paid is that the LLT is no longer serializable, which in general, is not acceptable.

A popular way to relax consistency in commercial systems is to compromise between correctness and performance. This is usually done by defining degrees of isolation (Gray and Reuter 1993) weaker than serializability. Serializability, as defined in Section 1.4, and achieved by retaining all the locks until the locked point of a transaction, is the highest degree of isolation and is called 3° isolation. In 2° isolation, read locks on the data items are released as soon as the item has been read instead of keeping them until the transaction's locked point. Thus, if a transaction reads an item more than once, it may get different results. Even weaker notions of consistency are defined along similar lines.

In this thesis, we assume that the transactions are required to be serializable. Therefore, none of the above approaches applies.

1.6.2 Using Multiple Versions

It is possible to increase concurrency if we make use of multi-versioned data (Chan et al. 1990; Reed 1983; Bernstein, Hadzilacos and Goodman 1987; Carey and Muhanna 1986). A multi-version concurrency control algorithm maintains old versions of updated data. Conflicts can be avoided by allowing read only transactions to see old versions. A disadvantage of this approach is that one has to maintain and manage the old versions. On the other hand, such an approach can be quite attractive if the database maintains the old versions for its own purposes anyway and the versions do not become an extra overhead for concurrency control. This technique, however, does not work when there are no read-only transactions.

1.6.3 Using Information About Data Types

In an abstract data type, one can use the information about data types to achieve greater concurrency (Schwarz and Spector 1984). For example, in the case of a data type queue, if there is more than one entry in the queue, two transactions, one of which adds an item and the other that removes an item may be allowed to run concurrently even though both of them are writes. A similar analysis can be done for other data types. This approach has to be tailored to a specific application.

1.6.4 Extensions of 2PL

Some recent methods have tried to extend two-phase locking to allow transactions to concurrently access the entities that are locked (Salem, Garcia-Molina and Shands 1994; Agrawal and Abbadi 1990). For example, under an *altruistic locking policy* (Salem, Garcia-Molina and Shands 1994), a transaction T_1 is allowed to lock an item (assuming all locks are exclusive) that is locked by T_2 but has been *donated* by T_2. In addition, all items locked by T_1 should have been donated by T_2 or T_2 must have released some lock. Thus, T_1 is concurrently able to access the entities that are locked by T_2, but the set of entities that it may access is limited until T_2 unlocks an item.

Similarly, under an *ordered shared locking policy*, a transaction T_1 is allowed to hold a conflicting lock with another transaction T_2, by putting T_1 *on hold* for T_2. Furthermore, T_1 cannot release any lock until T_2 releases some lock (thus, T_1 is on hold until T_2 releases some lock). Here again, T_1 is able to concurrently access the items that are locked by T_2 in a conflicting mode, but its commit may be delayed until T_2 releases some lock.

1.6.5 User Controlled Concurrency

To get around the problem of long transactions, some systems make a copy of the whole database and give it to each of its users. Users independently make changes to the database and submit it to a central server that tries to detect inconsistencies. If it finds any inconsistencies, it passes them along to the persons who made the changes. Otherwise, it propagates the changes to all users. Once a week, a "dump" of the database is made so that new machines can load this dump and don't have to perform all the operations locally to catch up. Such an approach has been adopted in the CYC knowledge base project (Guha and Lenat 1994; Guha 1994). Similar approaches have been advocated by others (Korth, Kim and Banchilhon 1988; Adams and Miller 1989; Mays et al. 1991). The basic assumptions in this approach are that it is easy to make a copy of the knowledge base and that the cost of detecting and repairing inconsistency is much smaller as compared to the cost of preventing it by ensuring serializability. Such an approach is feasible for a small knowledge base but it is unclear if it will scale up to more realistic knowledge base sizes that are to be used in a production environment.

1.6.6 Viewing the Database as a Directed Graph

In this approach, assuming that the database is structured as a directed graph, one can allow release of locks before a transaction reaches its locked point. For example, one such policy, called the Directed Acyclic Graph (DAG) policy[3] may be specified by the following rules (Silberschatz and Kedem 1980; Yannakakis 1982a):

> **D1.** A transaction T may begin execution by locking any entity.
> **D2.** Subsequently, T can lock an item if T has locked all the predecessors of that entity in the past and is currently holding a lock on at least one of them.
> **D3.** A transaction T may lock an entity only once.

Unlike 2PL, the above version of the DAG policy is deadlock-free, that is, if transactions follow the DAG policy then a deadlock never arises. A different version of the policy (Kedem and Silberschatz 1983) in which shared and exclusive locks are allowed is not deadlock-free. Furthermore, the DAG policy allows a transaction to release certain locks before it has acquired all the locks it will ever need. The freedom of transactions to release locks earlier often results in a greater degree of concurrency than would be possible under two-phase locking.

1.7 Approach of the Thesis

Our approach is based on the observation that, given a knowledge base, such as KB_1, we can selectively use semantic relationships and represent them as a directed graph. For example, the directed graph shown in Figure 1.4(a) is one of the possible ways to view KB_1 as a graph. In this graph, each class is represented by a node, and there is an edge (A, B) from node A to node B if B is a subclass of A. Each instance is represented by a node, and there is an edge between a class and each of its instances. In Figure 1.4(b), we show a dependence graph representation of the deductive rules in the knowledge base KB_1 (Ullman 1988). In the dependence graph, each predicate is represented by a node, and there is an edge (A, B) between two nodes A and B if B appears in the body of A. This dependence graph can be extended to incorporate class hierarchies and temporal knowledge (Plexousakis 1993) or to deal with subsumption relationships (Borgida and Patel-Schneider 1994).

[3] The DAG policy when applied to graphs that are restricted to trees is also known as the Tree policy.

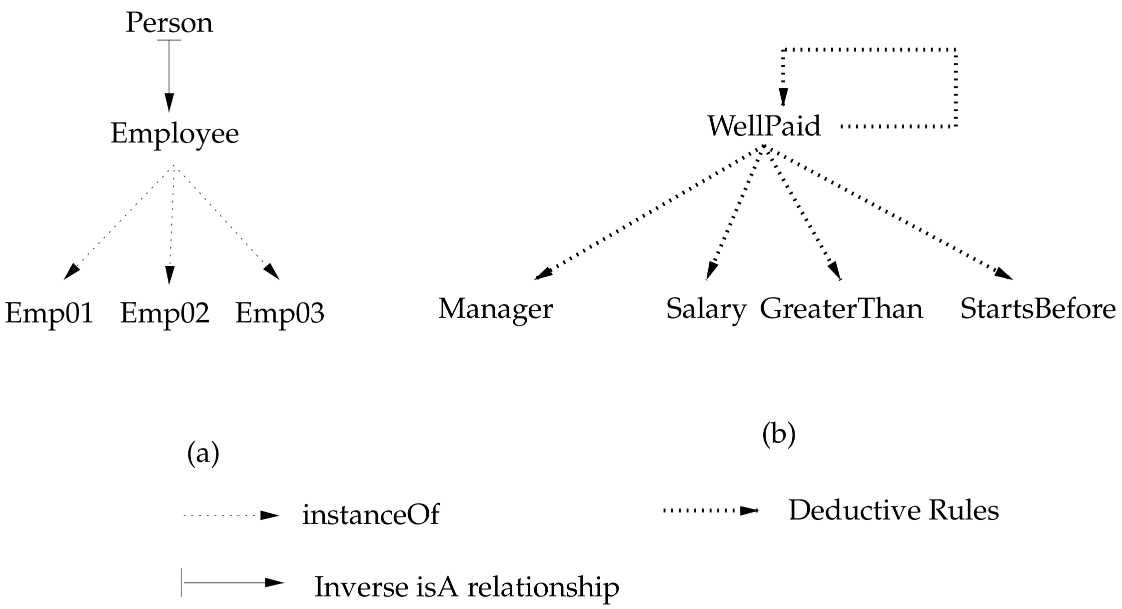


Figure 1.4: Directed graph representation of the knowledge base KB_1

Directed graph structures may be found in other knowledge bases. For example, consider a spatial knowledge base that stores the aerial view of a city. The city is broken down into suburbs, suburbs into blocks, and blocks into buildings. Each of these areas is represented by a box in Figure 1.5(a). In an efficient implementation to query such a knowledge base (Ellis 1995), one constructs a hierarchy over the inclusion relationship between the boxes (Figure 1.5(b)). In such a representation, the knowledge base reduces to a directed graph (in fact to a tree).

There can be more than one way to construct a directed graph corresponding to a knowledge base. The purpose of the above examples is to show that it is natural to view a knowledge base as a directed graph using one or more of the semantic relationships in terms of which the knowledge base is structured. Unless otherwise stated, we assume that we are given a directed graph representation of the knowledge base without necessarily knowing how such a representation was obtained.

The observation that most knowledge base features can be visualized as graphs has been the driving force in our research. Therefore, we have taken the DAG policy as a starting point for our work. However, the DAG policy assumes that there are no cycles in the underlying structure and the structure does not undergo any change (Yannakakis 1982a). Unfortunately, the graphs that model features that arise in knowledge bases often contain cycles (e.g., in the inference graph generated for a collection of recursive rules) and undergo change (e.g., when the rule definitions are changed or rules are added or deleted). This means that the DAG policy cannot be directly applied to knowledge bases.

1.8 Contributions of the Thesis

The primary contribution of this thesis is the development of an algorithm called the Dynamic Directed Graph (DDG) policy that can handle cycles, insertions and deletions in a knowledge base. The DDG policy is an extension of the DAG policy. It includes

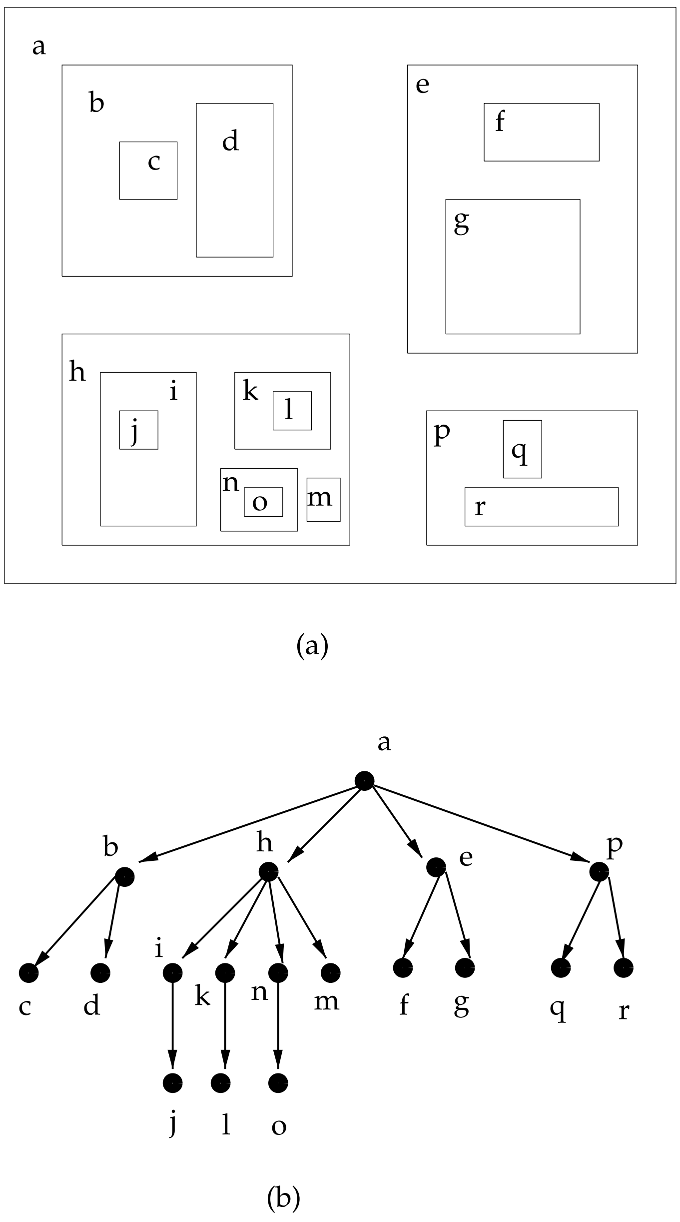


Figure 1.5: A Spatial knowledge base and its graph representation

structure maintenance operations to support the changing graph and its locking rules are more general than the locking rules of the DAG policy to take care of cycles, insertions and deletions in the graph. The algorithm supports both shared and exclusive locks and has a desirable property that it is *well-structured*. The DDG policy with only exclusive locks is *deadlock-free*, but the DDG-SX policy, in which both shared and exclusive locks are permitted, is not deadlock-free.

To analyze the correctness of the DDG policy, we develop a general theory of non-serializable schedules found in databases that undergo insertions and deletions. We show that in such systems, a *canonical schedule*, a schedule in which all transactions except one are executed serially, is more general than a canonical schedule found in databases that do not undergo insertions and deletions. We show the utility of this result by using it to prove the correctness of different versions of the DDG policy and two other locking policies.

The algorithmic results of this thesis are based on the observation that the directed graphs are a compelling abstraction for a large class of knowledge bases. The results have, however, a general applicability. This is because the DDG policy is independent of the source from which the directed graph is derived. Thus, the directed graph could come from a system which does not satisfy any definition of a knowledge base, and the DDG policy will still be applicable.

We simulate the performance of the DDG policy and compare it to the performance of 2PL. While simulating the DDG policy, we considered several issues that arise in implementing it. We give a detailed design of a prototype implementation and indicate the computations at various stages of transaction execution. We identify efficient algorithms that are necessary for performing these computations (for example, to compute *dominator tree* and to compute *strongly connected components*). We indicate the assumptions that need to be made about the transactions and give a mechanism that controls the release of locks by a transaction before it commits.

The evaluation of the algorithm is done in the context of three real applications (one of these is more extensively treated than the other two). We characterize the workload using a two-class system in which Class 1 transactions consist of long transactions that access a large number of entities, and Class 2 transactions consist of short transactions that perform short lookups and updates. Under these conditions we show that the Class 2 response time obtained using the DDG policy is significantly better than the corresponding response time obtained using 2PL without substantial penalty in the response time of Class 1 transactions. In addition, we also show how to select a subset of the semantic relationships in the knowledge that should be used for concurrency control and the effect of traversal strategies on transaction response time. These results are backed by an extensive robustness analysis.

1.9 Thesis Outline

In Chapter 2, we describe the DDG policy and its formal properties. A detailed reading of this chapter will be of interest to readers who wish to go deep into the correctness issues. A casual reader can obtain a general understanding of the material in Chapter 2 by reading Sections 2.1, 2.3, 2.5.1, 2.5.2, the algorithm description in Section 2.5.3 and finally Section 2.7. In Chapter 3, we give a detailed design of the implementation, which includes several algorithms that are used to compute the properties of the knowledge base. This chapter will be of interest to readers who wish to understand the intricacies of building a system using the ideas presented in Chapter 2. Our experimental setup for studying the performance of the DDG policy is described in Chapter 4. In Chapter 5, we study the performance of the DDG policy for three real knowledge bases. Both Chapters 4 and 5 are of general interest and are easily accessible to a casual reader. In Chapter 6, we give a summary of the thesis and draw general conclusions from our research.

As pointed out in the previous section, the results of Chapter 2 are not restricted to knowledge bases and are applicable to any system in which the information is represented as directed graphs. In Chapter 5, we consider three case studies and instantiate the abstract directed graph abstraction to real knowledge bases. That's when the relevance and applicability of this work to knowledge bases becomes more immediate.

Some of the preliminary results from Chapters 2 and 5 have already been published as conference papers (Chaudhri, Hadzilacos and Mylopoulos 1992; Chaudhri et al. 1994). A paper describing the details of theoretical results from Chapter 2 has been accepted for publication (Chaudhri and Hadzilacos 1995). A summary of the results of the thesis has been published as part of a larger paper giving system description of a knowledge base management systems (Mylopoulos et al. 1992a). Another paper giving a consolidated summary of the dissertation is available as a technical report (Chaudhri and Mylopoulos 1995).

Chapter 2

The Dynamic Directed Graph Policy

In order to formally analyze the correctness of a concurrency control algorithm, it is essential to have a model that is easy to understand, and preferably, independent of the data model or knowledge representation scheme being used. We begin this chapter in Section 2.1 by describing such a model of a knowledge base. We state the assumptions about the knowledge base and the transactions. In Section 2.3, we describe a restricted version of the Dynamic Directed Graph (DDG) policy which is one of the key ideas developed in this chapter. We prove the correctness of the DDG policy (Theorem 2) using the theorem on canonical schedules (Theorem 1). (As an aid to follow the results of this chapter, in Figure 2.1 we show the relationships amongst some of the key theorems.) The proof of the theorem on canonical schedules (Theorem 1), another important contribution of this chapter, is presented later in the chapter in Section 2.5.3, when we generalize our model to deal with both *shared* and *exclusive* locks (Theorem 6). In addition to the correctness of the DDG policy, we consider several variations of it (Theorems 5 and 7) and also analyze its deadlock-freedom and well-structured-ness (Theorems 3 and 4). Towards the end of the chapter, in Section 2.6, we use the canonical schedules theorem to prove the correctness of the dynamic tree policy and the altruistic locking policy (Theorems 8 and 9), thus demonstrating its general applicability. We conclude the chapter with a summary and discussion of the results.

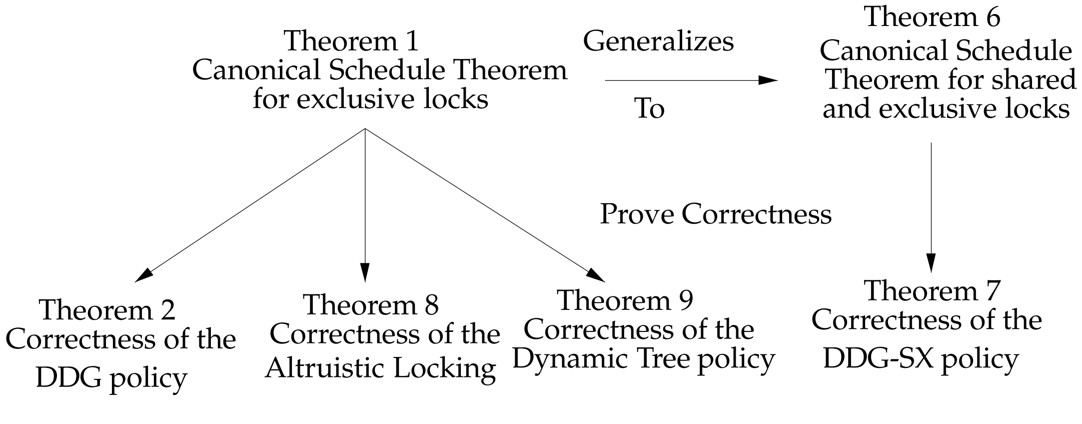


Figure 2.1: Relationships amongst some key theorems

$T_1 = \langle$`(DELETE (WellPaid,Greaterthan)) (DELETE (WellPaid,Salary))`
`(INSERT (WellPaid,Manager))`$\rangle$

Figure 2.2: An example transaction

2.1 A Model of Knowledge Bases

In describing our model of the knowledge base and in our algorithms we refer to a knowledge base as a database. We made this choice because the theory and algorithms that we have developed are fundamental to database concurrency control. The term knowledge base is used whenever we refer to a specific instance of a database.

The database is a collection of entities drawn from a set U. The set U is a universal set which contains all the entities that may exist in the database over its life time.

When the database is viewed as a directed graph $G = (V, E)$, where V is a set of nodes A_i (for example `Employee` in Figure 1.5), and E is a set of edges which are ordered pairs (A_i, A_j) of nodes (for example, (`Person, Employee`)), the term *entity* is used to denote both nodes and edges. We assume that the presence of an edge in the database implies that the database also contains the entities corresponding to both of its end points. Furthermore, when a node is deleted, it is not an end point for any edge. For such a database, the set U consists of all the nodes and edges that may exist in the database over its life time.

A database is defined by:

— a selection G of entities from the set U, $G \subseteq U$ and

— an assignment of values to the entities in G.

Each distinct selection of entities is called a *structural* state of the database. Corresponding to each structural state there can be several different assignments of values to the entities. Each such assignment is called a *value* state. The consistency constraints of the database define a set CS of *consistent states*.

For example, in the database of Figure 1.4(a), the selection of entities consists of nodes `Employee`, `Person`, `Emp01`, `Emp02`, `Emp03` and the edges `(Person, Employee)`, `(Employee, Emp01)`, `(Employee, Emp02) and (Employee, Emp03)`. The addition or deletion of a new class, for example, `Student`, or a new instance, for example, `Emp04`, will lead to a new structural state of the database. Changing the value of an attribute of an instance will lead to a new value state of the database.

A user interacts with the database by means of transactions. Each transaction is a sequence of `TELL, UNTELL, RETELL` and `ASK` operations (Mylopoulos et al. 1990). These operations are implemented by means of more primitive operations. For example, in terms of a graph representation in Figure 1.4(b), the transaction T_1 of Section 1.4 (shown in Figure 2.2) consists of several primitive operations: delete the edges `(WellPaid,Salary)`, `(WellPaid,GreaterThan)` and insert the edge `(WellPaid,Manager)`.

At an abstract level, the database may be changed by performing one or more operations from a set of operations O. In our model we take O =\{`A,I,D`\}, where `A,I` and `D` are the abbreviations for `ACCESS, INSERT` and `DELETE` respectively. The `INSERT` and `DELETE` operations change a structural state of the database by inserting or deleting an entity. The `ACCESS` operation changes a value state by assigning a different value to some entity in the database. We assume that an access operation on an entity e is the indivisible execution of the instructions $t_i \leftarrow e$ [read e]; $e \leftarrow f_i(t_1, \ldots, t_i)$[update e], where t_i's are local variables of T and f_i is an uninterpreted function symbol.

Formally, a *step* is a pair (`a` e), where `a` is an operation (from the set `I, A, D`) and e is an entity. A *transaction* is a finite sequence of steps over $O \times U$.

Let G be a structural database state. An `ACCESS` or `DELETE` (`INSERT`) operation is *defined* in G if it operates on an entity that exists (does not exist) in G. Thus given a sequence of operations S, we can inductively compute a new structural database state $S(G)$ that results by *applying* S to G. $S(G)$ is *undefined* if at least one operation in S is not defined in the database state in which it is executed.

A *transaction system* is a collection τ of transactions. A *schedule* S of a transaction system τ is an ordering of the steps of some transactions of τ that preserves the order of operations of each transaction. (We only consider some transactions of τ, because τ may be infinite, and in any schedule we are interested in studying the execution of some of the transactions.) Some such orderings may not be possible. For example, with G_1, G_2, G_3 and G_4 as structural database states, consider the interleaving:

$$
\begin{array}{lll}
G_1 \xrightarrow{T_{p1}} G_2 & & G_3 \xrightarrow{T_{p2}} G_4 \\
& G_2 \xrightarrow{T_q} G_3 &
\end{array}
$$

In this interleaving, T_p is executed in two parts, T_{p1} and T_{p2}. T_{p1} is executed in the structural database state G_1 and T_{p2} in the structural database state G_3. For this schedule to be defined, both $T_{p1}(G_1)$ and $T_{p2}(G_3)$ should be defined whereas $T_p(G_1)$ need not be defined. Intuitively, this means that the operations should be defined in the structural state in which they are executed and not necessarily in the state in which the transaction is initiated. We formalize this notion by defining *proper* schedules. A schedule S is *proper* for a structural state G, if $S(G)$ is defined.

As discussed earlier, a transaction transforms a database state by using uninterpreted functions. We assume that, when executed in isolation, for any interpretation of the function symbols, a transaction maps the set CS into itself. A schedule S of a set of transactions is *correct* if S also maps CS into itself. According to this definition, in a model that has only `ACCESS` operations, *correctness* is equivalent to *serializability*; that is, there is a serial schedule S' such that if S and S' start from the same initial state, they leave the database in the same final state and return the same values to the user. The same result holds true for a database that contains `INSERT` and `DELETE` operations. This can be seen by augmenting each entity in U with a boolean variable whose value is **true** if that entity exists in the database and **false** if it does not. Then, any structural state of G can be represented by U and an appropriate boolean value associated with each entity in U. In such a database, an insert operation on an entity A first reads the boolean value corresponding to A, and if found **false**, it writes the value of the boolean variable corresponding to A to **true**. Similarly, a delete operation on an entity A first reads the boolean value corresponding to A and if found **true** writes it to **false**. Thus, a model with `INSERT` and `DELETE` operations reduces to a model in which correctness is equivalent to serializability.

The notion of *conflicts* in a transaction system is captured by the predicate *conflicts*$((o_1\, A_1), (o_2\, A_2))$, where o_1 and o_2 are operations on the entities A_1 and A_2 respectively. This abstraction helps us to develop a general theory of serializability which is applicable to systems with different notions of conflict. For example, when the database is a directed graph, *conflicts*$((o_x\, A_x), (o_y\, A_y))$ is true if A_x and A_y have an entity in common. For example, if the entity A_x is a node A_1, and the entity A_y is an edge (A_1, A_2), then the operations (`D` A_1) and (`A` (A_1, A_2)) conflict.

Serializability of a schedule S can be decided as follows. Construct a directed graph $D(S)$ by associating a node with each transaction T_i and including an arc (T_i, T_j) if in schedule S, T_i executes an operation $(o_i\ A_i)$ before T_j executes an operation $(o_j\ A_j)$ and *conflicts*$((o_i\ A_i), (o_j\ A_j))$ is true. The schedule S is serializable if $D(S)$ is acyclic.

Locked transactions are transactions with two additional operations, `LOCK` and `UNLOCK` (abbreviated by `L` and `U` respectively) in their repertoire. We use (`U` *) to denote the release of all the locks by a transaction. Thus the set of allowed operations is now, O_L ={`A`,`I`,`D`,`L`,`U`}. A locked transaction is a sequence over $O_L \times U$. We say that a locked transaction T has locked A through step i if for some $j < i$, the jth step of T is (`L` A) and there is no k with $j < k < i$ such that the kth step of T is (`U` A). A locked transaction is *well-formed* if whenever the ith step of T is $(o\ A)$, then A is locked through step i; that is, an operation can take place only if the corresponding entity is locked. From now on, we consider only well-formed locked transactions.

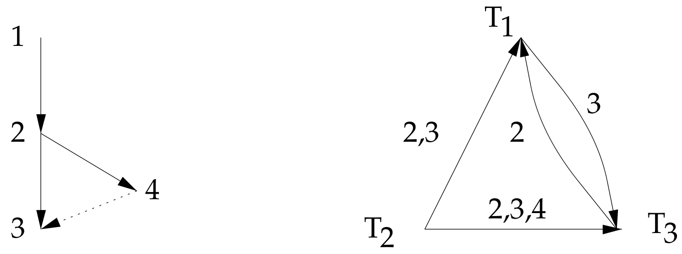
A *legal schedule* of a set of locked transactions is a schedule that respects the locks of the transactions; that is, a transaction cannot lock an entity that is already locked by some other transaction. A transaction system τ is *safe* if any legal and proper schedule of τ is correct.

A *locking policy* P is a relation such that $P(T, \overline{T})$ only if transaction T is a subsequence of a well-formed locked transaction $\overline{T}$. Intuitively, a locking policy tells us how to put locks in a given transaction. If $P(T, \overline{T})$ holds then $\overline{T}$ is one of the ways of locking T according to the locking policy. $P(T, \overline{T})$ is a many-to-many relation, and therefore, it is not possible to pre-specify *one* locked transaction for any given transaction. Instead, the locked transaction is dynamically computed depending on the structural state of the database when each step of the transaction is executed. A locking policy P is *safe* if for any transaction system $\tau = \{T_1, \ldots, T_m\}$ and $\overline{\tau} = \{\overline{T}_1, \ldots, \overline{T}_m\}$ where $P(T_i, \overline{T}_i)$ for all $1 \leq i \leq m$, the locked transaction system $\overline{\tau}$ is safe.

The *interaction graph* $G(\tau)$ of a transaction system τ is an undirected graph with one node corresponding to each transaction T_i of τ and an edge between any two nodes whose corresponding transactions have steps o_1 and o_2 such that *conflicts*$(o_1\ o_2)$ is true.

A *partial schedule* of a set of locked transactions $\{T_1, \ldots, T_m\}$ is a prefix of any schedule of these transactions. We say that an entity A is *mentioned* or *referenced* by a transaction T if T has a step involving A. We denote by $R(T)$ the set of entities referenced by a transaction T. Let L_T(step i) be the set of entities locked by transaction T through, but not including, step i. Furthermore, let R_T(step i) be the set of entities referenced by transaction T up to, and including, step i.

Let us summarize here the differences between our model and that of Yannakakis (Yannakakis 1982a). First, we distinguish between the structural and value states of the database. In the model of a static database, such a distinction was not necessary, because the database is always in the same structural state. Second, we explicitly distinguished `INSERT` and `DELETE` operations from a `WRITE` operation based on which we defined proper schedules. In case of a static database, every schedule is proper. Finally, we have defined locking policy as a relation in contrast to the static database where it was defined as a function. This is because for a given transaction, there can be more than one distinct mapping to the locked transactions depending on the structural state in which it is executed. The DDG policy is an example of one such policy.



(a) The knowledge base (b) The serializability graph $D(S_{DAG})$

T_1:	(L 1) (A 1)		(L 2,3) (A 2,3) (U 1,2,3)	
T_2:	(L 2,3,4) (I (4,3)) (U 2,3,4)			
T_3:		(L 2,4) (A 2,4) (U 2)		(L 3) (A 3) (U 3,4)

(c) The schedule S_{DAG}

Figure 2.3: An example of non-serializable schedule produced by naive application of DAG policy

2.2 A Naive Application of the DAG Policy to Dynamic Graphs

In this section, we illustrate by means of an example that a naive application of the DAG policy might produce incorrect results in the face of updates to the underlying DAG structure. Figure 2.3(a) shows a knowledge base which is manipulated by three transactions T_1, T_2 and T_3 running according to the locking rules D1-D3 of the DAG policy. The schedule S_{DAG} produced by these transactions is shown in Figure 2.3(c). We use (L 2,3,4) as a compact representation of three operations: (L 2), (L 3) and (L 4). The other operations in this schedule should be interpreted in a similar way. When T_1 begins execution, the edge $(4, 3)$ is non-existent. The DAG policy pre-computes the locked transaction, requiring T_1 to be holding a lock on node 2 at the time it acquires a lock on node 3. In the meantime, T_2 inserts the edge $(4, 3)$ and T_3 completes a part of its execution. If T_1 continues using the lock steps that it has pre-computed, we will get the schedule S_{DAG} as shown in Figure 2.3(c). The serializability graph of this schedule, shown in Figure 2.3(b), contains a cycle, and thus S_{DAG} is not serializable. With this motivation, let us give the description of the DDG policy, which in addition to handling the databases represented by cyclic graphs, overcomes this problem of insertion and deletion of nodes and edges in the database.

2.3 Description of the DDG Policy

We first define some properties of directed graphs that are necessary for specifying our algorithm. A *root* of a directed graph is a node that does not have any predecessors. A directed graph is *rooted* if it has a unique root and there is a path from the root to every other node in the graph. A directed graph is *connected*, if the underlying undirected graph is connected. A *strongly connected component* (SCC) G_i of a directed graph G is a maximal set of nodes such that for each $A, B \in G_i$, there is a path from A to B. An SCC is *non-trivial* if it has more than one node. An *entry point* of an SCC, G_i, is a node B such that there is an edge (B, A) in G, A is in G_i, but B is not in G_i. Thus, if a node is an SCC by itself, its entry points are simply its predecessors.

The dominator D of a set of nodes W is a node such that for each node $A \in W$, either every path from the root to A passes through D or D lies on the same strongly connected component as A. Thus, in a rooted graph, the root dominates all the nodes in the graph including itself. All nodes on a strongly connected component dominate each other.

We assume that once an object has been deleted from the database, it may not be inserted into it again. We call this assumption *unique object identifier* assumption. We highlight the importance of this assumption in Section 2.5.1.

The locking rules of the DDG policy assume that the underlying undirected graph is always connected and rooted. In the first subsection, we show how any arbitrary graph is converted to and maintained in this restricted form. In the second subsection, we specify the locking rules of the DDG policy.

2.3.1 Restricting the Database to a Rooted and a Connected Graph

Restricting the database to a rooted and a connected graph is a two-step process. First, the database has to be in this form to start with, and second, this form has to be maintained as the database undergoes insertion and deletion of nodes and edges. The first step is implemented by *preprocessing* rules and the second step by *structure maintenance* rules. The justification for this requirement will be given later in Section 2.3.3.

Preprocessing Rules

The preprocessing rules are applied when the database is compiled. These rules take as input a directed graph G representing the database and generate as output a graph $\overline{G}$, which is rooted and connected. They also compute some information that is later used by the locking and the structure maintenance rules. Here are the preprocessing rules:

P1. Partition G into $G_i(V_i, E_i)$, $1 \leq i \leq a$, where the underlying undirected graph of each G_i is a connected component of the underlying undirected graph of G. For each component, identify the non-trivial SCCs G_{ij}, $1 \leq j \leq b_i$.
P2. Compute the *sources* of G as follows: For each connected component G_i of G, condense each non-trivial SCC into one node. The edges that were incident to any of the nodes on an SCC are shown as the edges incident to the *condensed node* corresponding to that SCC. Similarly, the edges that were incident away from some node in an SCC are shown as the edges incident away from the condensed node corresponding to that SCC. Identify the roots s_{ij}, $1 \leq j \leq l_i$, of the condensed graph of G (the condensed graph has at least one root because it is acyclic). If any node from s_{ij}, $1 \leq j \leq l_i$, corresponds to a condensed non-trivial SCC, we replace it by an arbitrarily chosen node from the same SCC. We call nodes s_{ij}, $1 \leq j \leq l_i$ as the sources of G.
P3. For each $G_i(V_i, E_i)$, add a *control node* c_i. Add edges (c_i, s_{ij}), for all $1 \leq i \leq a$, $1 \leq j \leq l_i$. Add another control node $\mathcal{C}$ and the edges $(\mathcal{C}, c_i)$, for all $1 \leq i \leq a$.
P4. Call the resulting graph $\overline{G}(\overline{V}, \overline{E})$. Thus,

$$\overline{V} = V \cup \{\ c_i \mid 1 \leq i \leq a\ \} \cup \{\mathcal{C}\}$$
$$\overline{E} = E \cup \{\ (c_i, s_{ij}) \mid 1 \leq j \leq l_i, 1 \leq i \leq a\ \} \cup \{\ (\mathcal{C}, c_i) \mid 1 \leq a\ \}$$

Structure Maintenance Rules

Insert and delete operations applied to $\overline{G}$ may cause it to become disconnected or to acquire new sources. Furthermore, the information about the connected components of the graph needs to be updated. All this is enforced by the following rules.

M1. When a new source, s_i, is created in a connected component, G_j, add the edge (c_j, s_i). (The new sources are computed as in the preprocessing rule P2.)
M2. When an existing source, s_i, is removed from component G_j, remove the edge (c_j, s_i). If the removal of s_i results in the creation of new sources, then do as in M1. (The new sources are computed as in the preprocessing rule P2.)
M3. Two connected components G_i and G_j are merged. This will happen if an edge (A, B) is inserted with $A \in G_i$ and $B \in G_j$. Let $s_{i1}, s_{i2}, \ldots s_{il_i}$ and $s_{j1}, s_{j2}, \ldots s_{jl_j}$ be the sources of G_i and G_j respectively. Remove c_j (and therefore the edges, $(\mathcal{C}, c_j)$ and (c_j, s_{jm}) $1 \leq m \leq l_j$). Recompute the sources of the merged component as: $\{s_{i1}, s_{i2}, \ldots s_{il_i}\} \cup \{s_{j1}, s_{j2}, \ldots s_{jl_j}\} - \{B\}$. Add edges from c_i to each source in the new set of sources.
M4. A connected component $G_i(V_i, E_i)$ is split into components $G_{j1}(V_{j1}, E_{j1})$ and $G_{j2}(V_{j2}, E_{j2})$. Compute new sources and add and delete appropriate edges.
M5. As there are updates in the graph, keep updating the information on strongly connected components.[1]

When an edge is inserted, it may lead to activation of rules M1, M2, M3 and M5. When an edge is deleted, it may lead to activation of rules M1, M2, M4 and M5. A node insertion may require rule M1. Since deletion of a node will be preceded by deletion of edges incident to that node, it does not require any of the above rules. From now on, we assume that the database is always connected and rooted.

2.3.2 Locking Rules for a Transaction T

The rules presented in this section are the core of the DDG policy as they specify how a transaction T should acquire locks. We assume that a *lock operation that locks an SCC locks all the nodes on that SCC together.*

Locking Rules

L1. Before a transaction T performs any `INSERT, DELETE` or `WRITE` operation on a node A (an edge (A, B)), T has to lock A (both A and B).
L2. A node that is being inserted can be locked at any time.
L3. Each node can be locked by a transaction T at most once.
L4. A transaction T may begin by locking any SCC.
Subsequently,

[1]One can use incremental graph algorithms. Such algorithms can dynamically maintain certain kinds of information about a graph in the face of updates to the graph without recomputing the information from scratch (Italiano 1986; Italiano 1988). More information about the specific algorithms is given in Chapter 3, Section 3.3.2.

L5. All the nodes of an SCC are locked together in one step provided all the entry points of that SCC in the *present state* of $\overline{G}$ have been locked by T in the past and T is now holding a lock on at least one of them.

For example, the transaction T_1, that was considered in Section 1.4, and is shown again in Figure 2.2 could start by locking `WellPaid`. Then it could lock `Salary` and then the edge `(WellPaid,Salary)`. After deleting this edge, T_1 could release the lock on `Salary` and proceed to lock `Greaterthan`. Thus, the transaction is able to acquire locks even after releasing some of the locks — a clear improvement over two-phase locking.

In the example of Figure 2.3, T_1 will not be allowed to lock node 3 in the new state of the database unless it has locked node 4 as well. This prevents the incorrect schedule S_{DAG}. In Section 2.4.2, we prove that any schedule produced by the DDG policy is correct.

2.3.3 Discussion of the DDG Policy

In this section, we discuss various aspects of the DDG policy.

Let us first give some justification for the preprocessing and structure maintenance rules. If the graph corresponding to a database is not connected, a transaction can span more than one component. To guarantee the correctness of all the schedules in such a situation, we will have to ensure that the transactions that access some components in common follow the same serialization order in these components. This could be achieved by maintaining a graph external to the database, in which there is a node T_i corresponding to each transaction T_i, and an edge (T_i, T_j) if T_i precedes T_j in some component. Assuming the executions are serializable within each connected component, the schedules produced in such a situation will be correct if this external graph is acyclic. This external graph is not necessary if there is only one connected component in the database. Thus, if we keep the underlying graph of the knowledge base connected at all times, we save the cost of maintaining and checking cycles in this external graph. On the other hand, we incur some cost in preprocessing and structure maintenance that is incurred only when there are insertions or deletions in the graph. As the changes in the graph are infrequent, our proposed design results in a net saving, and therefore, is computationally more efficient.

In the preprocessing, we assume that there will be only two levels of control nodes — one level of control node for each connected component and one control node for the whole graph. The number of levels of these nodes can be varied to improve the performance of the database. For example, we could have just one control node for the whole database or we could have $\lfloor \log_2 n \rfloor + 1$ levels by introducing a node for each pair of components and then a node at each higher level for each pair of nodes at an immediately lower level, where n is the number of connected components in the database. These two design options will permit different levels of concurrency and will have different running costs and overheads. The exact improvement will depend on the environment in which the knowledge base will be used.

Let us now discuss our locking rules. To complete successfully, a transaction that operates under the DDG policy must begin by locking a node that is a dominator of all the nodes that it wishes to lock. This is not inconsistent with the locking rule L4 which just says that to lock the first node, no other condition needs to be satisfied.

The locking rule L5 deals with both trivial and non-trivial SCCs. If a node is an SCC by itself, L5 reduces to saying that a transaction T can lock a node A if all predecessors of A in the present state of the database have been locked in the past and T is presently holding a lock on at least one of them.

Sometimes an edge insertion may lead to the creation of a new non-trivial SCC in the database by merging two SCCs. Suppose T_1 is a transaction that inserts an edge which leads to a new non-trivial SCC G_{ij}. It is possible that another transaction T_2 is holding locks on some of the nodes in G_{ij}. T_2 continues to hold locks on those nodes, but to obtain a lock on any other node in G_{ij}, it has to apply L5 with respect to the current state of the database (i.e. after the edge insertion). If it fails to satisfy L5, it must abort and start again.

There are two key differences between the locking rules of the DAG policy and those of the DDG policy. First, the locking rules of the DDG policy are applied to the current state of the graph, whereas the locking rules of the DAG policy are applied to the state of the graph when the transaction begins execution, which is also the current graph. In the case of the DAG policy, there is no need to distinguish between the initial and the current state of the graph, because the graph never changes. As we saw in the schedule S_{DAG} of the previous subsection, this results in a non-serializable behavior in the face of updates. Second, the locking rules of the DDG policy provide a solution for databases represented by cyclic graphs. In general, using the locking rules of the DAG policy, it is not possible to lock any node on a cycle. The DDG policy solves both of these problems.

In contrast to 2PL, the locks under the DDG policy need not be held until the transaction reaches its locked point. Because of this, the DDG policy permits more concurrency than 2PL. The early release of locks is an especially desirable property for traversal transactions that access a large number of data items and are often found in knowledge based systems. On the other hand, the DDG policy may require a transaction to lock more nodes than it would lock under 2PL. For example, consider the database of Figure 2.3(a) after insertion of the edge (4,3). On this database, consider a transaction that wants to access nodes 2 and 3 only. This transaction must also lock node 4 (to satisfy locking rule L5) even though it does not need it. Thus, some transactions must lock more items than they actually need. This effect is quantitatively evaluated in Chapter 5 (see Figure 5.10 and Section 5.4.3).

2.4 Properties of the DDG Policy

We begin this section by stating a theorem that applies to databases that receive insertions and deletions of entities. We then use this theorem to show that the DDG policy always produces correct schedules. Finally, we prove that the DDG policy is deadlock-free and does not impose any restriction on the set of data items that may be accessed by a transaction.

2.4.1 Theorem on Canonical Schedules

In this section, we state a theorem (Theorem 1) that characterizes unsafe transaction systems. Intuitively, Theorem 1 says that if a locked transaction system is not safe then it is *always* possible to construct a *canonical* non-serializable schedule, which is legal and proper, and in which all transactions except one are executed serially. The result of Theorem 1 is the generalization of a similar result for transaction systems that do not contain insert/delete operations (called static systems) (Yannakakis 1982a) to systems that contain insert/delete operations (called dynamic systems).

Theorem 1 will be used several times in this thesis. The utility of Theorem 1 comes primarily from the serial nature of the schedule S'. Due to the serial nature of S', we are able to isolate the execution of each transaction and analyze it independently from others, thus, simplifying the analysis. Let us now state the theorem.

Theorem 1 *A transaction system τ is not safe if and only if there are pairs (T_1, A_1), ...,(T_k, A_k) of locked transactions $T_1, \ldots, T_k$ in τ ($k > 1$) and not necessarily distinct entities, $A_1, \ldots, A_k$, and some $c, 1 \leq c < k$, such that*

(1) There is a subsequence $(\tilde{T}_1, \tilde{A}_1), \ldots, (\tilde{T}_{\tilde{k}}, \tilde{A}_{\tilde{k}})$ of $(T_1, A_1), \ldots, (T_k, A_k)$ such that $\tilde{T}_1, \ldots, \tilde{T}_{\tilde{k}}$ form a cycle in the interaction graph $G(\tau)$. Formally, $\tilde{A}_i \in R(\tilde{T}_i) \cap R(\tilde{T}_{i+1})$ with arithmetic mod k.

(2) In T_c, the entity A_k is locked after A_c is unlocked.

(3) Consider the partial schedule $S^{'}$ as follows. Let $T_c^{'}$ be the prefix of T_c up to but excluding the (L A_k) step, and $T_i^{'}$, for $i \neq c$, be the prefix of T_i up to and including the (U A_i) step. $S^{'}$ is the serial schedule of $T_1^{\prime}, \ldots, T_k^{\prime}$ in this order. $S^{'}$ satisfies following conditions:

(a) $S^{'}$ is legal.

(b) $S^{'}$ is proper.

(c) $D(S^{'})$ is acyclic, the indices of $T_1^{'}, \ldots, T_k^{'}$ are in their topological order in $D(S^{'})$, and $T_k^{'}$ is a unique sink in $D(S^{'})$.

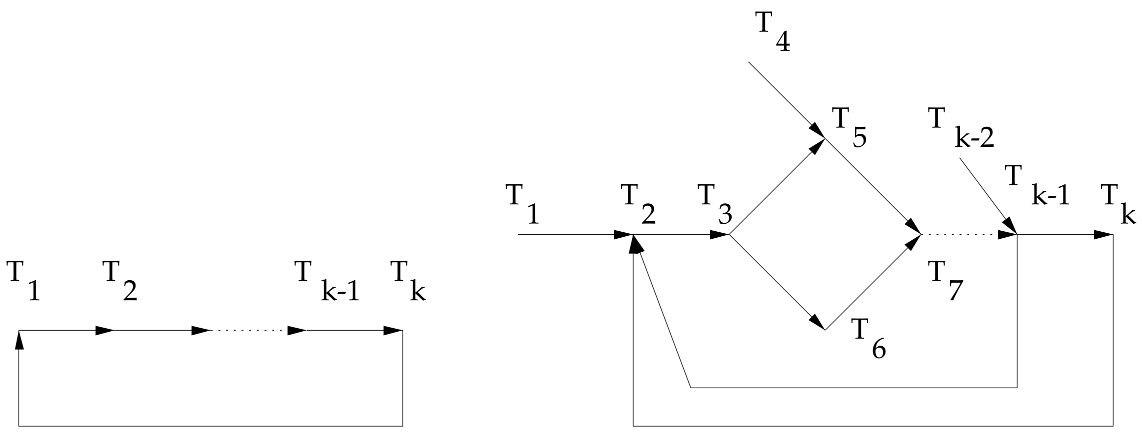
(d) $S^{'}$ can be extended to a complete legal and proper schedule, that is, can avoid deadlock.

Theorem 1 gives conditions under which a transaction system may produce incorrect schedules. The condition 1 of the theorem requires existence of a set of transactions that lie on a cycle in the interaction graph of the transaction system. Intuitively, this condition captures the existence of a cyclical dependency amongst the transactions. If there is no such dependency, then there is no way there could be a cycle in the serializability graph leading to non-serializable behavior. Condition 2 requires that at least one transaction must be non-two-phase in the sense that it must lock an entity after it has released some lock. Using this condition, it is trivial to claim the correctness of a system in which every transaction follows two-phase locking. Condition 3 requires the existence of a special kind of prefix $S^{'}$ of a complete legal and proper non-serializable schedule: a prefix in which every transaction is executed serially. Intuitively, this condition imposes a partial order on the transaction in $S^{'}$. When $S^{'}$ is extended by one lock operation as defined in condition (2), it results in a cycle in the serializability graph giving non-serializable schedule.

PROOF OF THEOREM 1.

Theorem 1 is a special case of Theorem 6 which is described in Section 2.5.3. The present theorem deals with only exclusive locks in contrast to Theorem 6 which deals with both shared and exclusive locks. As a result, condition 3c of Theorem 6 is more general than the condition 3c of the present theorem. Since there is a large overlap in the proofs of the two theorems, we do not give the proof here, but just show in Observation 2 (page 40) that when only exclusive locks are permitted, the condition 3c of Theorem 6 reduces to the condition 3c of the present theorem.

END OF THEOREM 1. □



(a) Static system (b) Dynamic System

Figure 2.4: Example serializability graphs for canonical schedules

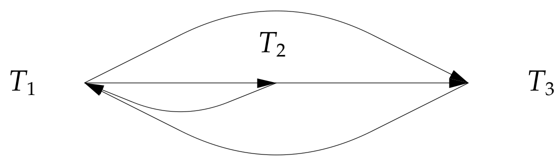
Observation 1 *The condition 3a of Theorem 1 may be formally stated as follows:*

$$
\begin{array}{lcl}
R_{T_i}(\mathtt{U}\,A_i) \cap \left\{ \cup_{j=1}^{i-1} (L_{T_j}(\mathtt{U}\,A_j) - \{A_j\}) \right\} & = & \emptyset \quad \text{for } 1 < i < c \\
R_{T_i}(\mathtt{L}\,A_k) \cap \left\{ \cup_{j=1}^{c-1} (L_{T_j}(\mathtt{U}\,A_j) - \{A_j\}) \right\} & = & \emptyset \quad \text{for } i = c \\
R_{T_i}(\mathtt{U}\,A_i) \cap \left\{ \left[\cup_{j=1}^{c-1} (L_{T_j}(\mathtt{U}\,A_j) - \{A_j\}) \right] \cup \right. & & \\
\qquad L_{T_c}(\mathtt{L}\,A_k) \cup \left. \left[\cup_{j=c+1}^{i-1} (L_{T_j}(\mathtt{U}\,A_j) - \{A_j\}) \right] \right\} & = & \emptyset \quad \text{for } c < i \leq k
\end{array}
$$

END OF OBSERVATION 1. □

The key difference between the above theorem and its analogue for static databases (Yannakakis 1982a) derives from the fact that the properness of schedules does not now come for "free": Special precautions must be taken to ensure that the canonical non-serializable schedule is, indeed, proper. This difference influences the theorem in three ways. First, all the transactions that are involved in the canonical schedule do not necessarily participate in a cycle in the interaction graph. Therefore, in condition 1 of the theorem, we need to consider a sub-sequence of the transactions $T_1, \ldots, T_k$. Second, the transaction T_c that locks A_k and creates the non-serializable schedule (condition (2) above) is not necessarily the first transaction in the sequence $T_1^{'}, \ldots, T_k^{'}$. This is because the properness of the transactions $T_c, \ldots, T_k$ may depend on the entities inserted/deleted by one or more of transactions $T_1, \ldots, T_{c-1}$. Finally, as specified in condition 3c, $D(S^{'})$ can have more than one source. For static databases, $D(S^{'})$ consists of a simple path which is closed by a back edge when T_c locks A_k (Figure 2.4(a)), whereas, for the dynamic databases the corresponding serialization graph is not necessarily a simple path (Figure 2.4(b)). The topological ordering of transactions (property 3(c)) becomes useful in the correctness proofs.

As an example, consider the schedule S_p and its serializability graph shown in Figure 2.5. Suppose at the beginning of this schedule the database is empty. For S_p to be proper, we need all three of the transactions. Even though T_1 and T_2 form a cycle in the interaction graph, any schedule produced by their interleaving is not proper, because for T_1 to be able to complete, it must be able to delete the entity c, which must exist in the database. We



(a) Serializability Graph $D(S_p)$

T_1: `(I a)                                   (A b) (D c)`
T_2: `      (A a) (I b)`
T_3: `                      (A a) (D b) (I c)`

Figure 2.5: A proper schedule S_p

can see that $D(S_p)$ is not just a simple path closed by a back edge but has two parallel paths (from T_1 to T_3) and two back edges ((T_2, T_1) and (T_3, T_1)) that create a cycle.

2.4.2 Correctness

We now use Theorem 1 to prove the correctness of the DDG policy. By virtue of the preprocessing and structure maintenance rules, the set of transactions are restricted in such a way that the graph always remains rooted and connected.

Let $\mathcal{N}$ denote the set of all nodes that might exist in the database. Then the set of all possible entities, U, is $\mathcal{N} \cup (\mathcal{N} \times \mathcal{N})$. Our model does not allow multiple edges between the nodes, but supports labeled edges in an indirect way as follows. An `ACCESS` operation can be thought of as checking the existence of an edge and returning the value of the label if one exists. The set of transactions consists of all finite sequences over $O \times U$.

We first prove a lemma about the properties of the *DDG* policy.

Lemma 1 *Let T be any transaction following the DDG policy, B be the first entity locked by T, and G be the state of the graph when T begins execution. In the absence of any other concurrent transactions:*

(a) All the entities locked by T are either dominated by B in G or have been inserted by T.

(b) For each A in $R(T)$, all nodes that are ancestors of A and descendants of B in G, have been referenced by T when it locks A.

PROOF OF LEMMA 1.

(a) By induction on i, we prove that the i-th node locked by T has the property stated in (a).

Base case. $i = 1$. T can begin by locking any node (by rule L4). If B is an SCC by itself, then B dominates itself. If B is on a non-trivial SCC, then along with B we also lock all nodes on that SCC, which dominate each other. This proves the claim for the base case.

Induction step. Suppose the first m nodes locked by T are dominated by B.

Let the $(m+1)$st node locked by T be C. If C is inserted by T then we are done. Otherwise, T must have locked all the entry points $C_1, \ldots, C_l$ of v in the past (by rule L5). Every path from the root to C must go through one of the entry points $C_1, \ldots, C_l$. Since by induction hypothesis every path from root to any of $C_1, \ldots, C_l$ must go through B, any path from root to C must go through B as well. Hence C must be dominated by B as well.

This proves the claim for the $(m+1)$st entity locked by T.

Hence by induction, all the entities locked by T are either descendants of B in G or have been inserted by T.

The lemma implies a weaker result: All the entities locked by T are either descendants of B in G or have been inserted by T.

(b) By induction on i, we prove that the i-th node locked by T has the desired property.

Base case. The lemma is trivially true for the first entity locked by T.

Induction step. Suppose for each A of the first m nodes locked by T, all nodes that are ancestors of A and descendants of B in G, have been referenced by T when it locks A.

Let the $(m+1)$st node locked by T be C. If C is an SCC by itself, T must have locked all entry points of C, which are also its predecessors, in the past (by rule L5), and the lemma is true for C by induction hypothesis. If C belongs to a non-trivial SCC G_{ij}, entry points of G_{ij} which must include all predecessors of C that do not lie on G_{ij} have been locked in the past (by rule L5); and its predecessors that lie on G_{ij} are locked together with C in the same step (by rule L5), thus satisfying the claim of the lemma. If C is being inserted, it does not have any predecessors, and therefore, condition (b) of the lemma is trivially satisfied.

Hence by principle of induction, for each A in $R(T)$, all nodes that are ancestors of A and descendants of B in G, are referenced by T when it locks A.

END OF LEMMA 1. $\square$

Before we proceed with the proof of correctness of the DDG policy, let us first give some intuition into how the proof is structured. We begin by constructing a non-serializable schedule as in Theorem 1. Then we argue that the first node locked by each transaction $T_i^{'}$ must be a descendant of A_k in G_i, for $c \leq i \leq k$, where G_i is the state of the database in which $T_i^{'}$ begins execution. This leads to a contradiction, because $T_c^{'}$ locks a predecessor of A_k and hence must begin from a predecessor of A_k in G_c. In order to establish the claim that the first node locked by transaction $T_i^{'}$ must be a descendant of A_k in G_i for $c \leq i \leq k$, we need to show two sub-claims. First, we need to show that A_k exists in the database state in which each of $T_c^{'}, \ldots, T_k^{'}$ executes. We accomplish this in Lemma 2 by using the fact that when $T_c^{'}$ finishes execution it holds a lock on a parent of A_k. And second, we need to show that if $T_i^{'}$ begins from an ancestor of A_k in G_i, for $c \leq i \leq k$, it violates the condition 3c of Theorem 1. We accomplish this using Lemma 8 and the observation that $T_c^{'}$ holds a lock on a predecessor of A_k.

Theorem 2 *The DDG policy is safe.*

PROOF OF THEOREM 2.

Suppose the DDG policy is not safe. Then we can choose transactions $T_1, \ldots, T_k$, entities $A_1, \ldots, A_k$ and construct the corresponding schedule S' as in Theorem 1. The serialization graph $D(S')$ of the partial schedule S' is a directed acyclic graph over $T'_1, \ldots, T'_k$, with $c < k$ and T'_k as a unique sink.

We assume that as $T'_1, \ldots, T'_k$ execute, by virtue of the preprocessing rules and structure maintenance rules, the database remains rooted and connected. This is necessary, otherwise, the concept of dominator is not defined.

For a node A and a structural database state G, define $F(A, G)$ as the set of entry points of the SCC in which A lies in G. For $1 \leq i \leq k$, let G_i be the structural state of the database when T'_i begins execution, and B_i be the first entity locked by T'_i. Let G_{k+1} be the structural database state when T'_k finishes its execution.

Since A_k is not the first entity locked by T_c (see property 2 of the canonical schedule in Theorem 1), and T_c is about to lock A_k when the database is in structural state G_{k+1}, $F(A_k, G_{k+1}) \cap L_{T_c}(\mathtt{L}\ A_k) \neq \emptyset$. This follows from the locking rules L5 and that T_c could not be inserting A_k using rule L2, because if that were the case it would mean that T_k deleted A_k (because S' is proper), and an entity that has been deleted cannot be re-inserted (recall the unique object identifier assumption). Let $f \in F(A_k, G_{k+1})$ be a node on which T_c is holding a lock at the time it locks A_k. T_c has held this lock since it executed its prefix before the ($\mathtt{L}$ A_k) step. This lock is held throughout the execution of $T'_{c+1}, \ldots, T'_k$ in S'.

PROOF OF THEOREM 2 (TO BE CONTINUED).

Before we proceed further, let us prove some properties of S'.

Lemma 2 *A_k is a successor of f in all of $G_c, \ldots, G_{k+1}$.*

PROOF OF LEMMA 2.

If A_k is an SCC by itself in G_{k+1} then f must be a predecessor of A_k in G_{k+1} (locking rule L5). T_c has held a lock on f since it executed its prefix T'_c. Since T_c locks A_k in the step immediately following T'_c, it could not have locked A_k in T'_c because L3 prevents a transaction from locking a node more than once. Therefore, T'_c did not insert or delete the edge (f, A_k) (because doing so would require locking of A_k as specified in rule L1). Since T_c has held a lock on f since it executed its prefix T'_c, no transaction from $T'_{c+1}, \ldots, T'_k$ could have deleted or inserted the edge (f, A_k) (because doing so would require locking of f as required by locking rule L1). Hence A_k must be a successor of f in the structural database states $G_c, \ldots, G_{k+1}$.

If A_k is not an SCC by itself and belongs to a non-trivial SCC G_{pq} in G_{k+1}, T_c locks all nodes in G_{pq} in step ($\mathtt{L}\ A_k$) (locking rule L5). We claim that T'_k must also lock all the nodes in G_{pq} (proven below). A_k is an entity that is locked by T'_k and by T'_c in the step immediately following T'_k (condition 3 of Theorem 1). Since T_c locks all nodes in G_{pq} in the step immediately following T'_k, and since T'_k also locked all nodes in G_{pq}, any node in G_{pq} would satisfy the property required of A_k. Therefore, let A_k be a successor of f. Then using an argument similar to the one used in the above paragraph, no transaction from $T_c, \ldots, T_k$ could have deleted or inserted the edge (f, A_k), and therefore, A_k must be a successor of f in the structural database states $G_c, \ldots, G_{k+1}$.

It remains to be proven that $T_k^{'}$ locks all nodes in G_{pq}. If A_k belongs to G_{pq} in G_k, then since $T_k^{'}$ locks A_k, $T_k^{'}$ must have locked all nodes in G_{pq} (by rule L5). If A_k does not belong to G_{pq} in G_k then A_k must become a part of G_{pq} as $T_k^{'}$ adds or deletes some edges to G_k. Suppose that the sequence of structural database states as $T_k^{'}$ inserts/deletes nodes and edges is $G_k^1, \ldots, G_k^h = G_{k+1}$. Consider the earliest structural database state G_k^{i+1} in which G_{pq} is created. We consider two sub-cases:

(i) G_{pq} is created by deletion of some edges from an already existing SCC in G_k^i, in which case $T_k^{'}$ must have locked all the nodes in G_{pq} (locking rule L5).

(ii) G_{pq} is created due to addition of an edge in G_k^i. Suppose the set of nodes in G_{pq} are $C = \{C_j \mid 1 \leq j \leq d\}$and an edge (C_d, C_1), is added to G_k^i that leads to the creation of G_{pq} in G_k^{i+1}. Prior to the addition of the edge, there must be a path from each of $C_1, \ldots, C_d$ to C_d in G_k^i, because otherwise the addition of the edge (C_d, C_1) will not lead to the creation of an SCC that contains all the nodes in the set C. Therefore, each $C_j, 1 \leq j \leq d$, is an ancestor of C_d in G_k^i.

We prove by induction on the length of the shortest path from $C_j, 1 \leq j \leq d$, to C_d in G_k^{i+1} that $T_k^{'}$ must lock all $C_1, \ldots, C_d$.

Base case. Consider C_d for which the length of the shortest path from C_d in G_k^{i+1} is 0. Since $T_k^{'}$ inserts the edge (C_d, C_1), it must lock C_d. This proves the base case.

Induction step. Suppose $T_k^{'}$ locks all $C_j, 1 \leq j \leq d$, such that the length of the shortest path from C_j to C_d in G_k^{i+1} is less than m.

We show that $T_k^{'}$ must lock all $C_j, 1 \leq j \leq d$, such that the length of the shortest path from C_j to C_d in G_k^{i+1} is equal to m. Let C_j be a node for which the length of the shortest path to C_d is m. Consider one such path and a node A such that $(C_j, A) \in G_k^{i+1}$ and the length of the shortest path from A to C_d is $m - 1$. We claim that $A \in G_{pq}$, because there is a path from C_j to A in G_k^{i+1} and a path from A to C_j via C_d in G_k^{i+1}. By induction hypothesis, $T_k^{'}$ locks A. Either (C_j, A) must be in G_k^i as well (may even lie on the same SCC as C_j) or the edge (C_j, A) must have been added by $T_k^{'}$. In the former case, $T_k^{'}$ must lock C_j due to locking rule L5 and in the later case, it must lock C_j due to locking rule L1. Therefore, $T_k^{'}$ must lock C_j.

Hence, by induction $T_k^{'}$ locks all of $C_1, \ldots, C_d$. Thus, $T_k^{'}$ locks all the nodes in G_{pq}.

END OF LEMMA 2. □

Lemma 3 *If a node A is a proper descendant of A_k in some $G_i, c < i \leq k+1$, then for all $j, c \leq j < i$, if $A \in G_j$, A is a proper descendant of A_k in G_j.*

PROOF OF LEMMA 3.

Suppose for contradiction, that the lemma is false, and consider the smallest $j, c \leq j \leq k$, such that $A \in G_j$, A is not a descendant of A_k in G_j, but A is a descendant of A_k in G_{j+1}. To accomplish this, $T_j^{'}$ must insert an edge from a descendant B of A_k in G_j to an ancestor C of A in G_j. Thus, $T_j^{'}$ must lock B and C (by locking rule L1). B_j must dominate, and thus be an ancestor of, B and C in G_j (Lemma 1(a)). Since C is not a descendant of A_k in G_j, B_j could not be a descendant of A_k in G_j. Since B_j dominates a descendant B of A_k in G_j, and B_j is not a descendant of A_k in G_j, and therefore, not on the same SCC as A_k, B_j must be a proper ancestor of A_k in G_j, otherwise, there is a path from root to B in G_j that passes through A_k, but does not pass through B_j. As $T_j^{'}$ locks B, it must lock A_k which is an ancestor of B and a descendant of B_j (by Lemma 1(b)). We now show that it is impossible for $T_j^{'}$ to lock A_k, yielding the desired contradiction.

For $j = c$, $T_j^{'}$ cannot lock A_k, because we know that T_c locks A_k in the step immediately after $T_c^{'}$, and it cannot lock an item more than once (by rule L3). For $c < j \leq k$, $T_j^{'}$ cannot lock A_j either because of the reasons as follow. Since $A_k \in G_j$ (due to Lemma 2), $T_j^{'}$ could not be inserting A_k. Furthermore, since A_k is not the first entity locked by $T_j^{'}$, to lock A_k, $T_j^{'}$ must have locked all nodes in $F(A_k, G_j)$ (by rules L5); but $f \in F(A_k, G_j)$ (by Lemma 2) and T_c holds a lock on f. Thus for all $c \leq j \leq k$, $T_j^{'}$ cannot lock A_k. Hence A must be a descendant of A_k in G_j.

END OF LEMMA 3. □

PROOF OF THEOREM 2 (CONTINUED).

We prove by backward induction on $i, c < i \leq k$, that B_i must be a descendant of A_k in G_i.

Base case: Consider $i = k$.

As noted earlier, $F(A_k, G_{k+1}) \cap L_{T_c}(\texttt{L}\ A_k) \neq \emptyset$ and $f \in F(A_k, G_{k+1})$ is a node on which T_c holds a lock at the time when it locks A_k. By Lemma 2, $f \in F(A_k, G_k)$. From condition (3a) of Theorem 1, $L_{T_c}(\texttt{L}\ A_k) \cap R_{T_k}(\texttt{U}\ A_k) = \emptyset$ (see Observation 1). Since $f \in L_{T_c}(\texttt{L}\ A_k)$, T_k did not reference (and therefore lock) f before the step $(\texttt{U}\ A_k)$. Therefore, $F(A_k, G_k) \not\subseteq R_{T_k}(\texttt{L}\ A_k)$ and hence A_k must be the first entity locked by $T_k^{'}$ (by rules L4,L5). This proves the claim for the base case.

Induction step: Suppose the claim is true for all $T_i^{'}$, $c + 1 < m < i \leq k$; that is, $B_{m+1}, \ldots, B_k$ are descendants of A_k in $G_{m+1}, \ldots, G_k$ respectively.

Consider the transaction $T_m^{'}, c < m < k$. From $D(S^{'})$, pick a transaction $T_i^{'}$ such that $i > m$ and $(T_m^{'}, T_i^{'}) \in D(S^{'})$ (such a transaction must exist by condition 3c of Theorem 1 and because $m \neq k$). By the induction hypothesis, B_i is a descendant of A_k in G_i. To reach a contradiction, suppose that B_m (the first node locked by $T_m^{'}$) is not a descendant of A_k in G_m.

First we show that $T_m^{'}$ does not lock any descendant of A_k in G_m. For, suppose the contrary, and let A be a descendant of A_k in G_m that $T_m^{'}$ locks. B_m must be a dominator of A in G_m (by Lemma 1(a)). Furthermore, B_m must be an ancestor of A_k in G_m, because otherwise there is a path from the root to A in G_m which does not pass through A_k. (Recall that the database is always rooted and connected, by rules P1-P4, M1-M5. Also, B_m and A_k do not lie on the same SCC in G_m because B_m is not a descendant of A_k in G_m.) To lock A, $T_m^{'}$ must lock A_k (due to Lemma 1(b)). Since A_k is not the first entity locked by $T_m^{'}$ (otherwise $B_m = A_k$ and B_m is descendant of A_k), $T_m^{'}$ must lock all nodes in $F(A_k, G_m)$. As $f \in F(A_k, G_m)$ (by Lemma 2) and f is already locked by T_c, $T_m^{'}$ cannot lock f, and therefore, it cannot lock A_k. This proves that $T_m^{'}$ cannot lock any descendant of A_k in G_m.

Next we show that $T_i^{'}$ locked only descendants of A_k in G_m or the nodes that it inserted. From locking rule L2, $T_i^{'}$ must lock the nodes that it inserts. Furthermore, since $T_i^{'}$ began execution by locking B_i, $T_i^{'}$ locked only descendants of B_i in G_i. By induction hypothesis, B_i is a descendant of A_k, and therefore, $T_i^{'}$ locked only descendants of A_k in G_i. The nodes that are descendants of A_k in G_i were descendants of A_k in G_m, if they existed in G_m (due to Lemma 3). Therefore, $T_i^{'}$ locked only descendants of A_k in G_m and the nodes that it inserts.

Thus, $T_m^{'}$ did not lock any descendants of A_k in G_m, and $T_i^{'}$ locked only descendants of A_k in G_m and the nodes that it inserted. The nodes inserted by $T_i^{'}$ cannot be locked by $T_m^{'}$, because they did not exist in G_m. Therefore, $R(T_m^{'}) \cap R(T_i^{'}) = \emptyset$. But this is a contradiction because $(T_m^{'}, T_i^{'}) \in D(S^{'})$ (due to the choice of T_i). Therefore, B_m must be a descendant of A_k in G_m. Thus, the claim is true for $T_m^{'}$ and hence all $T_{c+1}^{'}, \ldots, T_k^{'}$.

Let $T_i, i > c$, be a transaction such that $(T_c^{'}, T_i^{'}) \in D(S^{'})$ (such a transaction must exist due to condition 3c of Theorem 1). As proven above, B_i is a descendant of A_k in G_i. Let A be an entity that is locked by both $T_c^{'}$ and $T_i^{'}$. Thus, A must be a descendant of B_i in G_i (by Lemma 1(a)), and therefore, A is a descendant of A_k in G_i. Due to Lemma 3, A is a descendant of A_k in G_c as well. Since $T_c^{'}$ locks a predecessor f of A_k in G_c (due to Lemma 2), B_c must be an ancestor of A_k in G_c (due to Lemma 1(a)). Since $T_c^{'}$ locks A, it must also lock A_k which is an ancestor of A in G_c and a descendant of B_c in G_c (due to Lemma 1(b)). We know that T_c executes the ($\mathtt{L}$ A_k) step after the prefix $T_c^{'}$, and since it cannot lock an entity more than once (by rule L3), it could not have locked A in the prefix $T_c^{'}$. Thus, such an entity A cannot exist, contradicting the definition of A. Hence the assumed non-serializable schedule cannot exist and the DDG policy must be safe.

END OF THEOREM 2. □

2.4.3 Deadlock-freedom

A deadlock is a situation where a cyclical sequence of transactions are each waiting for the next to release a lock it must acquire (Holt 1972). Such deadlocks may be resolved by choosing one of the transactions, aborting it (i.e., undoing any effects it had on the database state), releasing its locks and restarting it at a later time. Thus, if a locking policy is deadlock-free, it would mean that it will never have to abort any transaction. The DDG policy has this desirable property.

Let us consider a typical case of deadlock for a set τ of transactions. Deadlock arises in a partial schedule S of τ when every transaction in the next step wants to lock an entity, that is already locked by some other transaction. This means that there is a set of transactions $\{T_1, \ldots, T_k\} \subset \tau$ such that the next step of T_i is ($\mathtt{L}$ A_i) where A_i is currently locked by T_{i+1} (where we take $k + 1 = 1$).

Let us prove that the DDG policy avoids deadlocks.

Theorem 3 *The DDG policy is deadlock-free.*

PROOF OF THEOREM 3.

Suppose, in a schedule S, for $1 \leq i \leq k$, T_i blocks waiting for a lock on A_i which is currently held by T_{i+1} (with arithmetic mod k), when the database is in state G_i. Suppose that for some i, the transaction $T_i^{'}$ obtained from T_i by moving the ($\mathtt{U}$ A_{i-1}) step right before the ($\mathtt{L}$ A_i) step is also a transaction of the DDG policy. Then the partial schedule S could be extended to a non-serializable schedule of the system $[\cup_{j \neq i}\{T_j\}] \cup \{T_i^{'}\}$; $T_i^{'}$ can execute the ($\mathtt{U}$ A_i) step and then $T_{i-1}, T_{i-2}, \ldots, T_1, T_k, \ldots, T_i^{'}$ can finish in this order giving a serializability graph that has a cycle. But this contradicts the safety of the DDG policy. Therefore, for each each i, A_{i-1} (with arithmetic mod k) must be an entry point of the SCC to which A_i belongs.

Thus, A_k is a predecessor of A_1 in G_1 (or an entry point of the non-trivial SCC to which it belongs), A_1 a predecessor of A_2 in G_2 (or an entry point of the non-trivial SCC to which it belongs), etc. The relationship amongst $A_1, \ldots, A_k$ is maintained in G_k, because if that is not the case for some A_i, T_i would be able to lock A_i and there would be no deadlock.

This implies that these entities $A_1, A_2, \ldots, A_k$ lie on a cycle in G_k. The DDG policy locks all node on a cycle in one step which could not be locked by different transactions simultaneously. Therefore, the assumed deadlocked schedule cannot exist, and hence, the DDG policy is deadlock-free.

END OF THEOREM 3. □

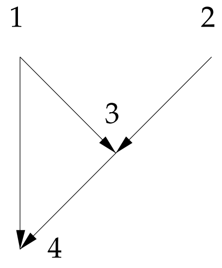


Figure 2.6: Multiple-sourced DAG

2.4.4 Well-structured-ness

A locking policy P is *well-structured* if it allows a transaction to access an arbitrary set of entities in the knowledge base when there is no other concurrent transaction. Formally, P is well-structured if in a database state G and any transaction T, such that T is defined in G[2], there is a locked transaction $\overline{T}$ such that $P(T, \overline{T})$ and $\overline{T}$ is also defined in D.

As an example of a locking policy that is not well-structured, consider the multiple source DAG (MSDAG) policy (Yannakakis 1982b). To state the rules of this locking policy, we need the following definition. A node A *separates* nodes B and C in a directed graph G, if after deleting A from G, there is no path between B and C in the underlying undirected graph of G. The locking rules of MSDP for a transaction T can now be stated as follows:

MSD1. Before accessing an entity A, T must lock A.
MSD2. T must not have previously locked A.
MSD3. The first lock of T is arbitrary.
For every subsequent lock,
MSD4. An entity A can be locked by T only if

(a) T has previously locked all predecessors of A for which deletion of A does not separate them from the first entity locked by T, and

(b) T holds a lock on at least one predecessor of A.

For example, consider the multiple sourced DAG shown in Figure 2.6. The MSDAG policy does not allow a transaction T that locks nodes 2, 3 and 4 because of the following reasons. If T begins by locking node 2 using rule MSD1 then using rule MSD4a it can lock node 3 without having to lock node 1 (deletion of node 3 separates nodes 1 and 2). It cannot lock node 4, because deletion of node 4 does not separate node 1 from node 2, and therefore to lock node 4, T must lock node 1. T cannot lock node 1, because it cannot be locked by using any of the locking rules. Similarly, if T begins by locking node 1 using rule MSD1, and as before, using rule MSD4a it can lock node 3 without having to lock node 2. It cannot lock node 4, because deletion of node 4 does not separate node 2 from node 1, and therefore to lock node 4, T must lock node 2. T cannot lock node 2, because it cannot be locked using any of the locking rules. Thus, this locking policy artificially limits the set of nodes that may be accessed by a transaction. The following theorem shows that the DDG policy does not have such limitation and is another justification for our preprocessing and structure maintenance rules.

[2]Recall that a transaction T is defined in a knowledge base state G if it does not insert (delete, access) an entity that already exists (does not exist) in the database.

Theorem 4 *The DDG policy is well-structured.*

PROOF OF THEOREM 4.

It is possible for a transaction T to lock an arbitrary set of entities in the graph if we can identify one entity starting from which the locking rule L5 will be satisfied for all the entities in the transaction (provided there is no other concurrent transaction). Since the rules P1-P4 and M1-M5 ensure that the graph is always rooted and connected, the node $\mathcal{C}$ of $\overline{G}$ satisfies this property for all nodes in the graph. Hence, the DDG policy is well-structured.

END OF THEOREM 4. □

From the proof of this theorem we can see the importance of the preprocessing and structure maintenance rules. Keeping the graph rooted and connected at all times ensures that the locking policy is well-structured.

2.5 Extensions of the DDG Policy

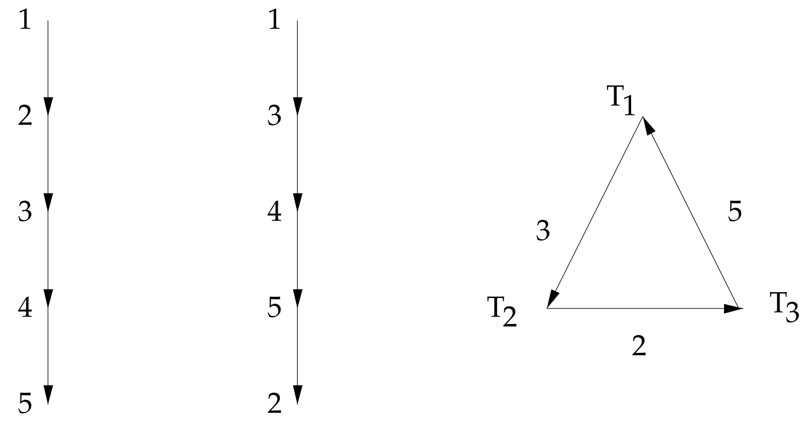
In this section, we consider several extensions of the DDG policy. In particular, we consider the unique object identifier assumption and its influence on the design of the policy (Section 2.5.1), permitting concurrency within the cycles (Section 2.5.2), and allowing shared and exclusive locks (Section 2.5.3).

2.5.1 Unique Object Identifier Assumption

In our model, we assumed that once an object is deleted from the database, it may not be inserted again. We call this the unique object identifier assumption (Section 2.3). This assumption led to a simple locking rule (rule L2) for the nodes that are inserted. In this section, we consider a situation in which we do not make the unique object identifier assumption.

For example, suppose in the database shown in Figure 2.7(a), we execute the schedule S_{uid} shown in Figure 2.7(d). In S_{uid}, T_2 deletes node 2 which is later inserted by transaction T_3 leading to the database state shown in Figure 2.7(b). If this is allowed to happen it leads to a non-serializable schedule when T_1 tries to access node 5 (see the serializability graph $D(S_{uid})$ shown in Figure 2.7(c)). One can verify that in S_{uid}, all locks are obtained in accordance with the DDG policy as stated in Section 2.3.

Under the unique object identifier assumption, node 2 cannot be re-inserted, and therefore, the non-serializable schedule shown in Figure 2.7 cannot occur. If we do not wish to make this assumption, we need to modify the rules of the DDG policy to ensure correctness. One possible approach is to require that any transaction T that inserts a node A must begin by locking the root of the database. (A more efficient solution would require such a transaction to start from the dominator, in the present state of the database, of all the nodes that are locked by active transactions.) This ensures that T is serialized after any active transaction that might have existed before A was deleted. Thus, in the example considered in the previous paragraph, T_3 would need to start by locking node 1 and would be serialized after transaction T_1, thus avoiding the non-serializable schedule S_{uid}. If all transactions that insert a node into the graph have to start from the root, it can become a bottleneck especially when there are several insertions at the leaf level. This is the best one



(a) Initial state (b) Final State (c) $D(S_{uid})$

T_1:(L 3,4)(U 3) (L 5)
T_2: (L 1,2,3)(I (1,3))(D (2,3))(D (1,2))(D 2)(U *)
T_3: (L 5)(L 2)(I 2)(I (5,2))(U *)

(d) The schedule S_{uid}

Figure 2.7: Significance of unique object identifier assumption

can do unless one makes more assumptions. This is because in the absence of any other information, we cannot assume that the node A that is being inserted was not deleted in the past and none of the active transactions referenced A (before A was deleted).

In general, it is reasonable to make the unique object identifier assumption, and therefore, the DDG policy as presented in Section 2.3 should meet the needs of most applications.

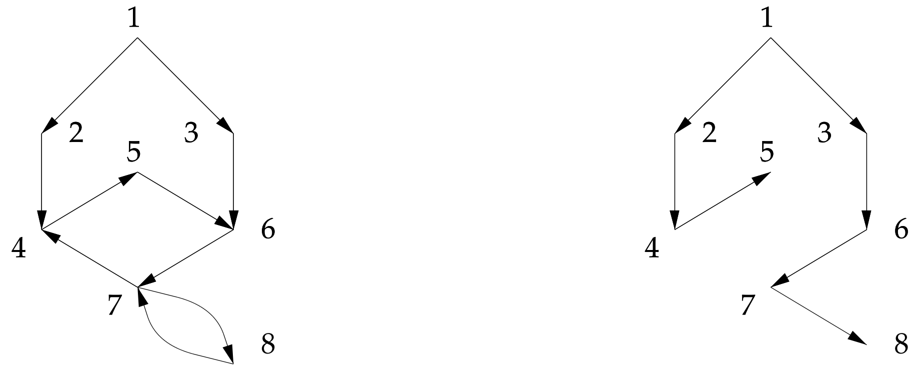
2.5.2 A More Liberal Variant of the DDG policy

Locking rule L5 of the DDG policy requires that all the nodes on a non-trivial SCC should be locked together in one step. This does not permit any concurrency within a non-trivial SCC. A natural question is whether this is the best one can do or whether there is a locking policy that can allow concurrency within non-trivial SCCs.

Here we describe a locking policy DDG′ that allows concurrency within cycles. Under the DDG′ policy, we transform the database so that it can be represented by a directed acyclic graph using the following preprocessing rule. (Assume that the database is initially rooted and connected.)

> **P5.** For each strongly connected component G_{ij} in G, if there is an edge (u, v) such that $u, v \in G_{ij}$, and v is the successor of an entry point of G_{ij}, drop (u, v) from G. After dropping this edge, if the nodes contained in G_{ij} still form a non-trivial SCC, recursively apply this rule until the nodes contained in G_{ij} are SCCs by themselves.

Once P5 has been applied to all the strongly connected components in the database, let the resulting transformed graph be G^T. Every time there is an insertion or deletion of



(a) An example database (b) The transformed database

T_1:(L 2,4) (U 2) (L 5) (U *)
T_2: (L 3,6) (U 3) (L 7) (U *)

Figure 2.8: Permitting concurrency within non-trivial SCCs

an edge from G we apply the following structure maintenance rule to maintain the acyclic structure of the G^T:

M5a. Apply the rule P5 to the updated *SCC* and incorporate the changes into G^T.

Now the locking rule L5 reduces to saying that a transaction T can lock a node if all its predecessors in the present state of the G^T have been locked in the past and the transaction is presently holding a lock on at least one of them.

Let us now see how this approach permits concurrency within non-trivial SCCs. For example, consider the database G shown in Figure 2.8. It is transformed into G^T (by using P5) as shown in Figure 2.8(b). G contains one non-trivial SCC, namely {4, 5, 6, 7, 8}. The DDG′ policy permits transactions $T_1 = \langle(\mathtt{A}\ 2, 4, 5)\rangle$ and $T_2 = \langle(\mathtt{A}\ 3, 6, 7)\rangle$ to concurrently access the nodes on this non-trivial SCC. However, transactions $T_3 = \langle(\mathtt{A}\ 2, 4, 5, 6)\rangle$ and $T_4 = \langle(\mathtt{A}\ 3, 6, 7, 4)\rangle$ cannot concurrently access this non-trivial SCC. Since node 1 is the only node that dominates both nodes 4 and 6, each of them must begin execution by locking node 1. Suppose T_3 locks node 1 before T_4. T_3 will be able to lock the entry points 4 and 6 before T_4 and it will not release locks on them until it has finished processing the whole non-trivial SCC. Thus, T_3, will be the only transaction executing within the non-trivial SCC. Similar arguments can be made when T_4 locks node 1 before T_3. Thus, the DDG′ policy permits concurrency within a non-trivial SCC to a limited extent. The DDG′ policy satisfies the same properties as the DDG policy:

Theorem 5 *The DDG′ policy is a safe, deadlock-free and well-structured policy.*

PROOF OUTLINE.
The *DDG′* policy is the *DDG* policy applied to the graph G^T, and therefore, safety, deadlock-freedom and well-structured-ness follow from Theorems 2, 3 and 4.

END OF THEOREM 5. □

One way to compare the DDG and DDG′ policies is to observe that both policies transform the database into an acyclic graph. In case of the DDG policy, to obtain an acyclic graph we condense each non-trivial SCC into a single node, and therefore, the DDG policy does not permit any concurrency within the non-trivial SCCs. In contrast, in case of the DDG′ policy, we drop some edges from the original graph to obtain an acyclic graph, and therefore, concurrent access on some of the nodes on a non-trivial SCC is possible.

Let us analyze the relative merits of the DDG and the DDG′ policies. The DDG policy is intuitive and simple as it treats non-trivial strongly connected components as a unit of locking. Furthermore, in applications such as recursive processing, it is highly likely that a transaction will access *all* the entities on a cycle and probably access them *more than once*. Since a transaction is allowed to lock an entity only once[3] (L3), it will have to retain all the locks on a non-trivial SCC until it has finished processing it. Thus, if a non-trivial SCC has only one entry point, then there would be no difference in the concurrency permitted by the two policies. Furthermore, when a non-trivial SCC has more than one entry point, concurrency within a non-trivial SCC is possible only if some nodes can be accessed without having to lock all the entry points. In the example discussed earlier in this section, in order to lock 5, T_1 needed to lock only one of the entry points, that is, node 4. But this may not be the case for many nodes in a non-trivial SCC. Therefore, we feel that, in general, we cannot take advantage of the additional concurrency allowed by the DDG′ policy. Based on these arguments, we decided to lock non-trivial SCCs as one node.

2.5.3 Extension to Shared Locks

Let us consider an extension to our model by distinguishing between read and write actions. This leads to a locking policy, called DDG-SX, which permits more concurrency than the DDG policy. The set of operations is now: $O = \{\mathtt{I,D,R,W}\}$, where `I`,`D`,`R` and `W` are abbreviations for `INSERT, DELETE, READ` and `WRITE` respectively. The locked transactions consist of operations: $O_L = \{\mathtt{I,D,R,W,LS,LX,U}\}$. `LS` denotes the acquisition of lock in shared mode, `LX` denotes the acquisition of lock in exclusive mode, and as before, `U` represents an unlock operation. We use generic notation `L` to denote either `LS` or `LX`.

We need to re-define legal schedules and well-formed transactions to take into account the new operations defined in the previous paragraph. In a legal schedule, two (or more) transactions may simultaneously hold a lock on the same entity only if both of them lock it in shared mode. In a well-formed transaction, before performing an `INSERT, DELETE` or `WRITE` operation on an entity, a transaction must lock it in exclusive mode; before performing a read operation on an entity, a transaction must lock it in shared or exclusive mode.

In this model, $\mathit{conflict}((o_x\ A_x),(o_y\ A_y))$ is true if one of o_x and o_y is not `READ` and A_x and A_y have an entity in common.

When both shared and exclusive locks are permitted, a sufficient (but not necessary) condition for the serializability of a schedule S (called D-serializability) has been studied elsewhere (Papadimitriou 1979). A directed graph $D(S)$ is constructed from the schedule S. The nodes of $D(S)$ are the transactions of S; there is a directed edge from T_i to T_j if T_i performs an action o_x on some entity A_x before T_j performs an action o_y on entity A_y in schedule S, and $\mathit{conflict}((o_x\ A_x),(o_y\ A_y))$ is true. If $D(S)$ contains no cycles, then the schedule S is serializable. In general, this condition is not necessary; it is necessary, however, if all

[3]This condition is necessary to guarantee the safety of *any* locking policy (Yannakakis 1982a).

```
T1: (LS 1) (LX 2) (U 1) (LS 3,4)    (LX 5) (U *)
T2:                       (LS 3,4)       (LX 5) (U *)
```

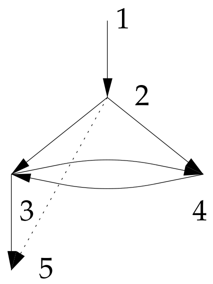


Figure 2.9: An example application of the DDG-SX policy

transactions read an entity before updating it, and that is the assumption we adopt for the rest of this section. (`INSERT` and `DELETE` operations are treated as updates.)

An analogous result to Theorem 1 holds in this model too: A transaction system τ is not safe if and only if there is a complete non-serializable legal and proper schedule, a prefix of which is almost serial, that is, all transactions but one are executed without interruption. There is, however, one important difference: $D(S^{'})$ may not necessarily have a unique sink. We first state the rules of the DDG policy with shared and exclusive locks, called DDG-SX policy, then state and prove more general version of Theorem 1 and then use it to prove the correctness of the DDG-SX policy.

We first state the locking rules of the DDG-SX policy.

Locking Rules for a Transaction T

L1. Before a transaction T performs any `INSERT, DELETE` or `WRITE` operation on a node A (or an edge (A, B)), T has to lock A (both A and B) in exclusive mode. Before T performs a `READ` operation on a node A (an edge (A, B)), it has to lock A (both A and B) in either mode.

L2. A node that is being inserted can be locked at any time.

L3. Each node can be locked by a transaction at most once.

L4. A transaction T may begin by locking any SCC.

Subsequently,

L5. All nodes on an SCC are locked together in one step if:

L5a. All entry points of that SCC in the *present state* of $\overline{G}$ have been locked by T in past, and T is now holding a lock on at least one of them, and

L5b. For every node A on this SCC that is a successor of an entry point, and every path $A_1, \ldots, A_p, A, p \geq 1$, in the present state of the underlying undirected graph of $\overline{G}$, such that T has locked A_1 (in any mode), and $A_2, \ldots, A_p$ in shared mode, T has not unlocked any of $A_1, \ldots, A_p$.

As an example application of the DDG-SX policy consider the database and the transactions shown in Figure 2.9. T_1 begins by locking node 1 (L4) in shared mode and then locks node 2 in exclusive mode (L5). It locks the nodes 3 and 4 which form a strongly connected component in shared mode in one step (L5). It is able to do so because the condition L5b is satisfied for each path from node 1 to node 3 and 4. It locks node 5 in exclusive mode and finishes execution. T_2 begins by locking both nodes 3 and 4 (L4) and then locks node 5 and finishes execution. If T_1 adds the edge $(2, 5)$ (L1), then T_2 will be unable to lock node

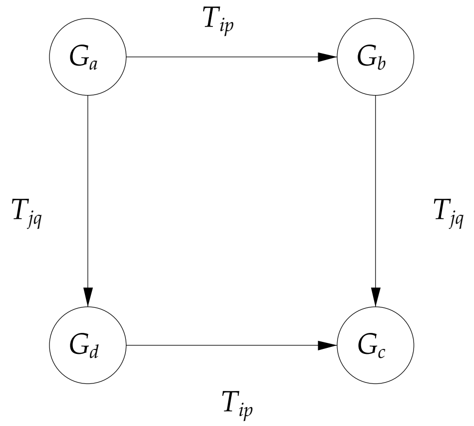


Figure 2.10: An example of commuting transactions

5 because in order to do that it must lock node 2 which is a predecessor of node 5 in the current state of the graph (L5a). T_2 must abort and start from node 2.

The DDG-SX policy permits more concurrency than the DDG policy but has the drawback that it is not deadlock-free. To see this, consider the graph shown in Figure 2.8(b). Two transactions T_1 and T_2 can start by locking node 1 in `S` mode. T_1 may lock 2 in `X` mode before T_2, whereas T_2 may lock 3 in `X` mode before T_1. If T_1 needs to lock 3 next and T_2 needs to lock 2, they will be in a deadlock, each waiting for the other to release a lock. Thus, the extra concurrency offered by the DDG-SX policy over the DDG policy is obtained at the cost of losing deadlock-freedom.

Let us now state the theorem on canonical schedules when shared and exclusive locks are allowed. But in order to do that, we need the following lemma.

Lemma 4 *Let S be a legal and proper schedule of a transaction system τ. In S, let T_{ip} and T_{jq} be two contiguous subsequences of two distinct transactions T_i and T_j respectively such that T_{jq} appears immediately after T_{ip}. If $(T_i, T_j) \notin D(S)$, the schedule obtained by commuting T_{ip} and T_{jq} is legal and proper.*

PROOF OF LEMMA 4.

Let G_a and G_b be the structural database states in which T_{ip} respectively begins and finishes execution. Similarly, let G_b and G_c be the structural database states in which T_{jq} respectively begins and finishes execution. (We show this in Figure 4 where nodes are structural database states; An edge between two nodes G_1 and G_2 labeled by a transaction T implies that execution of T in structural database state G_1 is defined and leads to the structural database state G_2.) If $(T_i, T_j) \notin D(S)$, then T_{ip} and T_{jq} do not perform any conflicting operations. In particular, they do not insert/delete any entity in common. Thus the entities required for the execution of T_{jq} to be proper in G_b exist (do not exist) in G_a as well. Hence $T_{jq}(G_a)$ is defined. Let $G_d = T_{jq}(G_a)$. By a similar argument we can show that $T_{ip}(G_d)$ is defined. As argued earlier, the sets of entities inserted/deleted by T_{ip} and T_{jq} are disjoint, the structural state obtained by executing T_{ip} in the state G_d will be the same as G_c. Therefore, the schedule obtained by this commutation is proper. Furthermore, since T_{ip} and T_{jq} do not perform any conflicting operations, this commutation also preserves legality.

END OF LEMMA 4. □

Theorem 6 *A transaction system τ is not safe if and only if there are pairs (T_1, A_1), ...,(T_k, A_k) of locked transactions $T_1, \ldots, T_k$ in τ ($k > 1$) and not necessarily distinct entities, $A_1, \ldots, A_k$, and some $c, 1 \leq c < k$, such that*

(1) There is a subsequence $(\tilde{T}_1, \tilde{A}_1), \ldots, (\tilde{T}_{\tilde{k}}, \tilde{A}_{\tilde{k}})$ of $(T_1, A_1), \ldots, (T_k, A_k)$ such that $\tilde{T}_1, \ldots, \tilde{T}_{\tilde{k}}$ form a cycle in the interaction graph $G(\tau)$. Formally, $\tilde{A}_i \in R(\tilde{T}_i) \cap R(\tilde{T}_{i+1})$ with arithmetic mod k.

(2) In T_c, the entity A_k is locked after A_c is unlocked.

(3) Consider the partial schedule $S^{'}$ as follows. Let $T_c^{'}$ be the prefix of T_c up to but excluding the ($\mathtt{L}$ A_k) step, and $T_i^{'}$, for $i \neq c$, be the prefix of T_i up to and including the ($\mathtt{U}$ A_i) step. $S^{'}$ is the serial schedule of $T_1', \ldots, T_k'$ in this order. $S^{'}$ satisfies following conditions:

(a) $S^{'}$ is legal.

(b) $S^{'}$ is proper.

(c) $D(S^{'})$ is acyclic, $T_1^{'}, \ldots, T_k^{'}$ is a topological sort of $D(S^{'})$ and there exists a non-empty set of transactions $\mathcal{K} = \{T_{\underline{k}+1}^{'}, \ldots, T_k^{'}\}$, such that each of $T_1^{'}, \ldots, T_{\underline{k}}^{'}$ has a path to each $T_j^{'} \in \mathcal{K}$, each $T_i^{'} \in \mathcal{K}$ locks A_k in a mode that conflicts with the mode in which T_c locks A_k, and for all distinct transactions $T_i^{'}, T_j^{'} \in \mathcal{K}$, there is no path from $T_i^{'}$ to $T_j^{'}$ in $D(S^{'})$.

(d) $S^{'}$ can be extended to a complete legal and proper schedule, that is, can avoid deadlock.

The difference between the above theorem and Theorem 1 is in condition 3c. The condition 3c in Theorem 1 requires $D(S^{'})$ to have a unique sink, in contrast to the present theorem, in which $D(S^{'})$ may have multiple sinks. This is a consequence of introducing shared locks in the model because of which we may have a situation in which each transaction in $\mathcal{K}$ locks A_k in shared mode and T_c locks A_k in exclusive mode. In such a scenario, when $S^{'}$ is extended with the ($\mathtt{L}$ A_k) step, all transactions in $\mathcal{K}$ participate in causing cycles in $D(S)$.

Before we proceed with the proof of Theorem 6, let us give some intuition into how the proof is structured. The *if* direction of the theorem follows easily by observing that the existence of schedule $S^{'}$ and the conditions 1 and 2 imply a cycle in the serializability graph giving the desired result. The *only if* direction is, however, more involved. We begin by arguing that the serializability graph of any arbitrary non-serializable schedule must have a shape similar to that of $D(S^{'})$ (Lemma 5). Then we transform the arbitrary non-serializable schedule to a serial schedule, but still retaining the shape of the serializability graph. This transformation is carried out using Lemma 4. Once the arbitrary non-serializable schedule has been transformed such that its prefix is serial, the desired result follows.

PROOF OF THEOREM 6.

If. Let S be a (complete) schedule of τ that has $S^{'}$ as a prefix (such a prefix exists because of 3d above). Construct the digraph $D(S)$ for this schedule. For each $\tilde{T}_i$ and $\tilde{T}_{i+1}$ (with arithmetic mod k) that are adjacent in the cycle in $G(\tau)$, there is an edge $(\tilde{T}_i, \tilde{T}_{i+1})$ (with arithmetic mod k) in $D(S)$ labeled $\tilde{A}_i$. Thus, there is a cycle in $D(S)$ corresponding to a cycle in $G(\tau)$, and therefore, the schedule S is not serializable.

Only If. Suppose τ is not safe, and let S be a legal and proper non-serializable schedule of τ. Let S^- be the longest prefix of S such that $D(S^-)$ is acyclic, S^+ be the shortest prefix for which $D(S^+)$ has a cycle. Thus, S^+ is S^- extended with one lock step; let T_c be the transaction that performs this step by locking A_k. Intuitively, T_c is the transaction that causes the earliest cycle in $D(S)$. We need the following lemma before we proceed further.

PROOF OF THEOREM 6 (TO BE CONTINUED).

Lemma 5 *τ contains a legal, proper, and non-serializable schedule S such that:*

(a) There exists a set of transactions $\mathcal{K} = \{T_{\underline{k}+1}, \ldots, T_k\}$, such that each of $T_1, \ldots, T_{\underline{k}}$ has a path to each $T_j \in \mathcal{K}$ in $D(S^-)$ and for all distinct transactions $T_i, T_j \in \mathcal{K}$ there is no path from T_i to T_j in $D(S^-)$.

(b) $D(S^+)$ contains an edge (T_j, T_c), for all $T_j \in \mathcal{K}$.

PROOF OF LEMMA 5.

Consider an arbitrary legal, proper and non-serializable schedule S of τ. To prove Lemma 5, we show that either S satisfies the properties (a) and (b) above, or it implies the existence of another schedule $\overline{S}$ which is legal, proper and non-serializable that satisfies the properties of Lemma 5.

Consider a set of transactions $\mathcal{K}$ such that for each transaction $T_i \in \mathcal{K}$, T_i locks A_k in a mode that conflicts with the mode in which T_c locks A_k, and no other transaction that locks A_k in a mode that conflicts with the mode in which T_c locks A_k, follows T_i in $D(S^-)$. Let $T_1, \ldots, T_l$ be the transactions whose steps appear in S. Without loss of generality, suppose transactions $T_1, \ldots, T_{\underline{k}}$ have a path of non-zero length, in $D(S^-)$, to each $T_i \in \mathcal{K}$, $T_{\underline{k}+1}, \ldots, T_k$ are in the set $\mathcal{K}$, and transactions $T_{k+1}, \ldots, T_l$ do not have a path, in $D(S^-)$, to any $T_i \in \mathcal{K}$. Furthermore, without loss of generality, let $T_k \in \mathcal{K}$ be the last transaction to lock A_k in S^-. Since T_c creates a cycle in $D(S^+)$ by adding the edge (T_k, T_c) in $D(S^-)$, there is a path from T_c to T_k in $D(S^-)$, that is, $c < k$.

If $k = l$, S trivially satisfies the properties of the lemma. We consider the case when $k \neq l$ and show that given S, it is possible to obtain a schedule $\overline{S}$ which satisfies the properties of Lemma 5.

We first define some terminology. In any schedule, we call a maximal contiguous subsequence of the operations of a transaction T a "sub-transaction" of T. Consider the sub-transactions $t_1, \ldots, t_s$, of $T_{k+1}, \ldots, T_l$ in S^+ such that for every t_i and t_j, if $i < j$, t_i appears after t_j in S^+. (Informally, if we scan S^+ left-to-right, we encounter these sub-transactions in the order $t_s, t_{s-1}, \ldots, t_1$.) We construct $\overline{S}$ from S as follows: Starting with S^+, we drop $t_i, 1 \leq i \leq s$, and then concatenate the resulting schedule by the sequence $t_s, \ldots, t_1$, and the operations of S that follow S^+. More formally, let $S^{+0} = S^+$ and for $1 \leq i \leq s$, S^{+i} is obtained from S^+ by dropping the operations of $t_1, \ldots, t_i$. Furthermore, let $S^0 = S$ and for $1 \leq i \leq s$, S^i is obtained by concatenating S^{+i} by $t_i, \ldots, t_1$ and $S - S^+$, where $S - S^+$ denotes the operations that follow S^+ in S. Thus, $S^s = \overline{S}$.

By induction on i, we show that for $0 \leq i \leq s$, S^i is legal and proper.

Base Case. S^0 is the same as S which is legal and proper by assumption.

Induction step. Suppose that for all $i < m$, S^i is legal and proper.

Now we show that S^m is legal and proper. Suppose that sub-transaction t_m belongs to a transaction $T_j, j > k$ (by definition of k). We must show that in $S^{(m-1)}$, if we commute t_m with any step t that follows it in $S^{+(m-1)}$, then the resulting schedule S^m is legal and proper. If the step t is performed by a transaction T_i then $i \leq k$ (by definitions of k and $S^{+(m-1)}$).

We first prove that there is no path from T_j to T_i in $D(S^+)$. For contradiction, suppose there is a path from T_j to T_i in $D(S^+)$. If this path also existed in $D(S^-)$, then there would also be a path in $D(S^-)$ from T_j to each transaction in $\mathcal{K}$, contradicting the choice of k. If this path did not exist in $D(S^-)$ then we must have a transaction $T_{j'}$ that locks A_k, $(T_{j'}, T_c) \notin D(S^-)$ and $(T_{j'}, T_c) \in D(S^+)$. Furthermore, there must be a path from T_j to $T_{j'}$, and from T_c to T_i in $D(S^-)$, so that the addition of edge $(T_{j'}, T_c)$ in $D(S^-)$ would explain the existence of a path from T_j to T_i in $D(S^+)$ that did not exist in $D(S^-)$. Since $T_{j'}$ locks A_k in a mode that conflicts with the mode in which T_c locks A_k, either $T_{j'} \in \mathcal{K}$ or some transaction $T_n \in \mathcal{K}, n \leq k$ follows $T_{j'}$ in $D(S^-)$. In either case, there is a path in $D(S^-)$ from $T_{j'}$ to each transaction in $\mathcal{K}$, and consequently, a path in $D(S^-)$ from T_j to each transaction in $\mathcal{K}$, thus, contradicting the choice of k. Hence there is no path from T_j to T_i in $D(S^+)$.

Since $S^{+(m-1)}$ is obtained from S^+ by dropping some operations, $D(S^{+(m-1)})$ is a subgraph of $D(S^+)$. Since there is no path from T_j to T_i in $D(S^+)$, we can claim that there is no path from T_j to T_i in $D(S^{+(m-1)})$. Therefore, $(T_j, T_i) \notin D(S^{+(m-1)})$. Hence by repeatedly applying Lemma 4 to the prefix $S^{+(m-1)}$ of the schedule S^{m-1}, we can commute t_m and t without violating legality and properness until t_m follows all sub-transactions that belong to $S^{+(m-1)}$.

Thus, by induction S^i is legal and proper for all i. Therefore $S^s = \overline{S}$ is legal and proper.

The schedule $\overline{S}^-$ contains the operations of only transactions $T_1, \ldots, T_k$. As $T_1, \ldots, T_{\underline{k}}$ have a path in $D(S^-)$ to each transaction in $\mathcal{K}$, and to obtain $\overline{S}$ from S we did not commute any operations of these transactions, there is a path in $D(\overline{S}^-)$ from each of $T_1, \ldots, T_{\underline{k}}$ to each transaction in $\mathcal{K}$, satisfying condition (a) of the lemma. Using a similar argument we can claim that $D(\overline{S}^+)$ has an edge (T_j, T_c) for all $T_j \in \mathcal{K}$ as $D(S^+)$ did, thus, satisfying condition (b) of the lemma. Thus, there is a cycle in $D(\overline{S})$, and therefore, $\overline{S}$ is non-serializable.

Finally, we observe that $\overline{S}$ is a schedule of τ because we obtain $\overline{S}$ from S just by permuting the steps of the latter. Hence the lemma is true.

END OF LEMMA 5. □

PROOF OF THEOREM 6 (CONTINUED).

From now on we assume that the non-serializable schedule S of τ has the properties of Lemma 5. Let $RX(T_i)$ denote the set of entities locked by T_i in exclusive mode.

Without loss of generality, renumber the transactions in their topological order in $D(S^-)$. For $1 \leq i \leq k$, let $F_i = \left(RX(T_i) \cap [\cup_{(T_i,T_j) \in D(S)} R(T_j)]\right) \cup \left(R(T_i) \cap [\cup_{(T_i,T_j) \in D(S)} RX(T_j)]\right)$. Intuitively, F_i is the set of entities locked by T_i that are subsequently locked by some other transaction in conflicting mode in S. Each F_i is non-empty because the node corresponding to each of $T_1, \ldots, T_k$ has a successor in $D(S)$ (due to Lemma 5). By definition of S^+, A_k is the first entity of F_k locked by T_c and for $\underline{k} < i \leq k$, let $A_i = A_k$. For $1 \leq i \leq \underline{k}$, let A_i be the last item of F_i unlocked by T_i in S^+. ($T_i, 1 \leq i \leq \underline{k}$, unlocks at least one item in F_i, because each one of them has a successor in $D(S^+)$. $T_j, \underline{k} < j \leq k$ has unlocked at least A_k, because T_c locks A_k in conflicting mode in S^+, and thus, causes the edge $(T_j, T_c) \in D(S^+)$ (Lemma 5(b).)

We claim that the chosen T_is and A_is satisfy the conditions of the theorem. Condition (1a) is satisfied because there is a cycle in $D(S)$ implying a cycle in $G(\tau)$. Condition (2) is satisfied by the choice of k, and the fact that $T_c^{'}$ unlocks A_c (as argued in the previous paragraph, $c \leq k$), and because T_c locks A_k just after the schedule S^- (causing an edge (T_k, T_c) in $D(S^+)$ — due to Lemma 5(b)).

We prove that conditions 3a and 3b are satisfied. That is, the schedule $S^{'}$ is legal and proper.

We define some notation: For $0 \leq i \leq k$, inductively define S_i as follows. Intuitively, in the sequence of schedules $S_0, S_1, \ldots, S_k$, we start with S and move the steps of $T_1^{'}, T_2^{'}, \ldots, T_k^{'}$ to the beginning of the schedule, in the order of their indexes. More formally, $S_0 = S$. For $0 < i \leq k$, let S'^i be S with steps of $T_1^{'}, \ldots, T_i^{'}$ removed. Then define S_i as the concatenation of $T_1^{'}, T_2^{'}, \ldots, T_i^{'}$ followed by S'^i. Thus, $S^{'}$ is a prefix of S_k.

For $1 \leq i \leq k, 1 \leq j \leq l_i$, let $T_{i,j}^{'}$ be the sub-transactions of $T_i^{'}$ in S^-. For $1 \leq i \leq k, 1 \leq j \leq l_i$, inductively define $S_{i,j}$ as follows. Intuitively, in the sequence of schedules $S_{i,0}, S_{i,1}, \ldots, S_{i,l_i}$, we start with S_i and move the sub-transactions of $T^{'}_{i+1}$ one by one towards the beginning so as to follow the steps of $T^{'}_1, \ldots, T^{'}_i$. More formally, $S_{i,0} = S_i$. For $j > 0$, $S_{i,j}$ is obtained inductively from $S_{i,j-1}$ by moving the steps of $T^{'}_{i+1,j}$ backwards so that they follow directly after all the steps of $T^{'}_1, T^{'}_2, \ldots, T^{'}_i, T^{'}_{i+1,1}, T^{'}_{i+1,2}, \ldots, T^{'}_{i+1,j-1}$. For all $1 \leq i < k$, $S_{i,l_i} = S_{i+1}$.

We show, by induction on $i, 0 \leq i \leq k$, that each S_i is legal and proper, and that $D(S^-_i) = D(S^-)$.

Base case. $i = 0$. $S_0 = S$ is legal and proper by assumption and $D(S^-_0) = D(S^-)$.

Induction step. Let $m \geq 0$ and for all $i \leq m$, suppose S_i is legal and proper, and $D(S^-_i) = D(S^-)$.

Now, we show that S_{m+1} is legal and proper, and that $D(S^-_{m+1}) = D(S^-)$. By definition, $S_{m+1} = S_{m,l_m}$. By induction on $j, 0 \leq j \leq l_m$, we show that $S_{m,j}$ is legal and proper for all j and that $D(S^-_{m,j}) = D(S^-)$.

Base case. $j = 0$. $S_{m,0} = S_m$ is legal and proper by induction hypothesis. Furthermore, $D(S^-_{m,0}) = D(S^-_m) = D(S^-)$.

Induction step. Let $n \geq 0$ and for all $j \leq n$, suppose $S_{m,j}$ is legal and proper and that $D(S^-_{m,j}) = D(S^-)$.

We show that $S_{m,n+1}$ is legal and proper as well. To prove this, we need to show that $T^{'}_{m+1,n+1}$ can be commuted backward with any $T^{'}_{p,q}$, $m+1 < p \leq k, 1 \leq q \leq l_p$ without violating legality and properness. Furthermore, we need to show that $D(S^-_{m,n+1}) = D(S^-)$.

Because the transaction indices are assigned in their topological order in $D(S^-)$, there is no path from T_p to T_{m+1} in $D(S^-)$ and consequently in $D(S^-_{m,n})$. Therefore, by Lemma 4, $T^{'}_{m+1,n+1}$ can be commuted backward with any $T^{'}_{p,q}$, $m + 1 < p \leq k, 1 \leq q \leq l_p$ without violating properness or legality. When this series of commutations has occurred, the resulting schedule is by definition $S_{m,n+1}$. Thus, $S_{m,n+1}$ is legal and proper. Furthermore, since there is no edge from T_p to T_{m+1} in $D(S_{m,n})$, $T^{'}_{m+1,n+1}$ and $T^{'}_{p,q}$ do not access any conflicting entities. Therefore, $D(S^-_{m,n+1})$=$D(S^-_{m,n})$=$D(S^-)$.

Hence by induction $S_{m,j}$ is legal and proper for all $j, 0 \leq j \leq l_m$, and $D(S^-_{m,j}) = D(S^-)$.

From the above we can conclude that, S_{m,l_m}, and therefore, S_{m+1} is legal and proper and that $D(S^-_{m+1}) = D(S^-)$. By induction it follows that for all $i, 0 \leq i \leq k$, S_i is legal and proper and that $D(S^-_i) = D(S^-)$. Therefore, $S^{'}$ which is a prefix of S_k is legal and proper. This proves conditions 3a and 3b.

Since $D(S^-_k) = D(S^{'}) = D(S^-)$, by Lemma 5(a), each of $T_1, \ldots, T_{\underline{k}}$ has a path in $D(S^{'})$ to each transaction in $\mathcal{K}$, and for all $T_i, T_j \in \mathcal{K}, i{\neq}j$, there is no path in $D(S^{'})$ from T_i to T_j. Also, the topological order of transactions in $D(S^{'})$ is the same as in $D(S^-)$. This proves condition 3c.

The partial schedule $S^{'}$ satisfies property 3d because it is a prefix of a complete, legal and proper schedule S_k.

END OF THEOREM 6. □

Observation 2 *Let us now see how Theorem 1 follows from Theorem 6. As pointed out earlier, the only difference between Theorems 1 and 6 is in condition 3c. Condition 3c in Theorem 6 defines a set of transactions $\mathcal{K}$ which satisfy the following properties: each transaction $T^{'}_i \in \mathcal{K}$ locks A_k, and for each $T^{'}_i, T^{'}_j \in \mathcal{K}$ there is no path between $T^{'}_i$ and $T^{'}_j$ in $D(S^{'})$. When all locks are exclusive, then any two transactions that lock A_k will have an edge between them in $D(S^{'})$ (because exclusive locks on the same entity conflict), and therefore, only one transaction can satisfy the property required of* *the set of transactions in $\mathcal{K}$. Hence, $\mathcal{K}$ is a singleton set containing $T_k^{'}$. This implies that $T_k^{'}$ is a unique sink as required in condition 3c of Theorem 1.*

Let us now use Theorem 6 to prove the correctness of the DDG-SX policy. We observe that the Lemma 1 holds for the case of the DDG-SX policy as well. The following observation is a consequence of Lemma 1(b).

Observation 3 *If a transaction T begins execution in a database state G, then in the absence of any other concurrent transaction, if T locks nodes A and B such that A is an ancestor of B in G, then T locks all ancestors of B that are descendants of A in G.*

Before proceeding with the proof of the DDG-SX policy, we need the following definitions and lemmas.

We define the *nearest common dominator* of two distinct nodes A and B in a rooted and connected graph G as a node C such that C does not lie on the same SCC as A and B, C dominates both A and B in G, and no proper descendant of C satisfies this property. Consider a set of nodes N that are locked by a transaction T in exclusive mode. We say that N *separates* nodes A and B in the structural database state G, if some ancestor of A or B in G that is also a descendant of their nearest common dominator in G belongs to N. (This definition of *separates* is different from the one used earlier in Section 2.4.4.)

Lemma 6 *Suppose a transaction T_1 holds exclusive locks on a set of nodes N before another transaction T_2 begins execution in the database state G. If T_2 locks nodes A and B in G while T_1 has not released any locks from N, then N does not separate A and B in G.*

PROOF OF LEMMA 6.

Let C be the nearest common dominator of A and B in G, and D be the first node locked by T_2. Due to Lemma 1(a), D dominates both A and B, and therefore, C is a descendant of D in G. Since C is also an ancestor of A (and B) in G, due to Lemma 1(b), it must be locked by T_2. Since T_2 locks A, T_2 also locks all ancestors of A that are descendants of C in G (due to Observation 3). Therefore, T_1 could not be holding an exclusive lock on any ancestor of A that is a descendant of C in G. Similarly, we can argue that T_1 could not be holding an exclusive lock on any ancestor of B that is a descendant of C in G. Therefore, no ancestor of A or B that is a descendant of C belongs to N. Hence, N does not separate A and B in G.

END OF LEMMA 6. $\square$

Lemma 7 *Let N be a set of nodes and G be a graph. If N does not separate A and B, but separates A and C then N also separates B and C.*

PROOF OF LEMMA 7.

All ancestor, descendant, dominator and separation relationships in this proof are with respect to graph G.

Suppose the lemma is not true. Then we must have nodes A, B and C such that N does not separate A and B, N separates A and C, but N does not separate B and C.

Let D_{AB}, D_{BC} and D_{AC} be the nearest common dominators for A and B, B and C, and A and C respectively. D_{AB} and D_{BC} cannot be incomparable in G, because if that were the case, then there is a path from root to node B that passes through D_{AB} (D_{BC}), but does not pass through D_{BC} (D_{AB}), and therefore, D_{BC} (D_{AB}) could not dominate B. Suppose, D_{AB} is an ancestor of D_{BC}. The proof for the case when D_{BC} is an ancestor of D_{AB} is analogous.

Since D_{AB} is an ancestor of D_{BC}, D_{AC} must be a descendant of D_{AB}. This is because, D_{AC} could not be a proper ancestor of D_{AB}, because D_{AB} being an ancestor of D_{BC}, dominates A, B and C. Furthermore, D_{AC} could not be incomparable to D_{AB}, because then there is a path from root to A that does not go through D_{AB}. Using similar arguments, we can show that D_{AC} could be either an ancestor or descendant of D_{BC}, but not incomparable to D_{BC}.

Since N does not separate A and B in G, of all ancestors of A or B that are descendants of D_{AB}, none belongs to N. Since D_{AC} is a descendant of D_{AB}, of all ancestors of A that are descendants of D_{AC}, none belongs to N.

Since N does not separate B and C, of all ancestors of B or C that are descendants of D_{BC}, none belongs to N. We partition the set of all ancestors of C that are descendants of D_{AC} into two sets. In the first set are the ancestors that are also descendants of D_{BC}, and as argued above, none of these belongs to N. In the second set are the ancestors that are not descendants of D_{BC}. If D_{AC} is a descendant of D_{BC}, then this set is empty. If D_{AC} is an ancestor of D_{BC}, then this set of ancestors is a subset of the ancestors of B that are descendants of D_{AB}, and as argued in the previous paragraph, none of these belongs to N. Thus, of all ancestors of C that are descendants of D_{AC} none belongs to N.

Based on the above arguments we can say that of all ancestors of A or C that are descendants of D_{AC} none belongs to N. Therefore, N could not separate A and C, which is a contradiction to the assumption that N separates A and C. Hence the lemma is true.

END OF LEMMA 7. □

Before giving the proof of correctness of the DDG-SX policy, let us give some intuition into how the proof is structured. The key to this proof is the notion of separation defined earlier in this section. We begin by constructing a canonical non-serializable schedule. Then we argue that when T_c locks A_k in the database state G_{k+1}, the set of entities N locked by it in exclusive mode separates A_k from any entity unlocked by it. The argument relies on Lemmas 8 and 9. We extend this argument to transactions $T_{c+1}, \ldots, T_k$ by repeated applications of Lemmas 6, 7 and 9. This eventually leads to a contradiction, because for a transaction $T_i^{'}$ that is adjacent to $T_c^{'}$ in $D(S^{'})$, $T_c^{'}$ must lock an entity common to $T_i^{'}$ and $T_c^{'}$ which must be and must not be separated from any entity unlocked by $T_c^{'}$. Let us now present this argument in more detail.

Theorem 7 *The DDG-SX policy is safe.*

PROOF OF THEOREM 7.

Suppose the DDG-SX policy is not safe. Then choose transactions $T_1, \ldots, T_k$, entities $A_1, \ldots, A_k$ and construct the corresponding schedule $S^{'}$ as in Theorem 6. The serialization graph $D(S^{'})$ is a directed acyclic graph over $T_1^{'}, \ldots, T_k^{'}$.

For a node A, and a structural database state G, define $F(A, G)$ as the set of entry points of the SCC in which A lies in G. (Thus, if A is an SCC by itself, $F(A, G)$ is just the set of predecessors of A in G.) For $1 \leq i \leq k$, let G_i be the state of the graph when $T_i^{'}$ begins its execution. Furthermore, let G_{k+1} be the structural database state when $T_k^{'}$ finishes its execution.

Since A_k is not the first entity locked by T_c (see property 2 of the canonical schedule in Theorem 6), and T_c locks A_k when the database is in structural state G_{k+1}, it must hold a lock on a node $f \in F(A_k, G_{k+1})$. This follows from the locking rule L5a and the fact that T_c could not be inserting A_k using rule L2, because if this were the case it would mean that T_k deleted A_k (because $S^{'}$ is proper), and an entity that has been deleted cannot be re-inserted. T_c has held this lock since it executed its prefix $T_c^{'}$. This lock is held throughout the execution of $T_{c+1}^{'}, \ldots, T_k^{'}$.

We observe that Lemma 2 holds for the DDG-SX policy as well, that is, A_k is a successor of f in all of $G_c, \ldots, G_{k+1}$. To proceed further, we need the following lemmas.

Recall that T_c is a transaction in the canonical schedule with some special properties as required in Theorem 6: In particular, if T_c is the next transaction to execute a step after the schedule $S^{'}$, then we will have a non-serializable execution.

Let N be the set of nodes on which T_c holds an exclusive lock just before (but not including) the step (L A_k).

PROOF OF THEOREM 7 TO BE CONTINUED.

Lemma 8 *If in some database state $G_i, c < i \leq k+1$, a node A is a descendant of some node $B \in N$ then for all $j, c < j \leq k+1$, if $A \in G_j$, A is a descendant of B.*

PROOF OF LEMMA 8.

Suppose the lemma is false. Then we must have nodes A and B and a database state $G_i, c < i \leq k+1$ such that $B \in N$, A is a descendant of B in G_i and A is not a descendant of B in G_j, where either $j = i+1$ (assuming $c < i \leq k$) or $j = i-1$ (assuming $c+1 < i \leq k+1$). We are able to assume that $\mid i - j \mid = 1$ due to the unique object identifier assumption, because otherwise, a node can be deleted from a database state G_i in which the lemma holds (does not hold) and may appear in a successive state $G_j, j - i > 1$, in which the lemma does not hold (holds).

First, consider the case when $j = i+1$ (assuming $c < i \leq k$). Therefore, A must be a descendant of B in G_i but not in G_{i+1}. Therefore, $T_i^{'}$ must delete an edge (D_1, D_2) such that D_1 is a descendant of B in G_i and D_2 is an ancestor of A in G_i, so that after deletion of this edge, A is no longer a descendant of B in G_{i+1}. Since the database is always rooted and connected (rules P1-P4, M1-M5), after deletion of the edge (D_1, D_2), there still must exist a path from root to A that does not pass through B. This means that the first node locked by $T_i^{'}$, C, must be an ancestor of B (due to Lemma 1(a)). Therefore, to lock D_1, $T_i^{'}$ must lock B (due to Lemma 1(b)). $T_i^{'}$ cannot lock B which is locked by T_c in exclusive mode. Thus, $T_i^{'}$ cannot delete this edge and hence A must be a descendant of B in G_{i+1}.

Next, consider the case when $j = i-1$ (assuming $c+1 < i \leq k+1$). Therefore, A is not a descendant of B in G_j but is a descendant of B in G_{j+1}. Therefore, $T_j^{'}$ must add an edge (D_1, D_2) such that D_1 is a descendant of B in G_j and D_2 is an ancestor of A in G_j, so that after the insertion of this edge, A is a descendant of B in G_{j+1}. Let C be the nearest common dominator of D_1 and D_2 in G_j. Since D_1 is a descendant of B in G_i, and D_2 is an ancestor of A which is not a descendant of B, C must be an ancestor of B in G_j. $T_j^{'}$ cannot lock both D_1 and D_2, because to do so, it must lock B (due to Lemma 1(b)) which belongs to N, and T_c holds an exclusive lock on B. Hence A must be a descendant of B in G_j. Hence the lemma is true.

END OF LEMMA 8. □

Lemma 9 *If N separates nodes A and B in some database state $G_i, c < i \leq k+1$, then N also separates nodes A and B in every database state G_i, $i, c < i \leq k+1$.*

PROOF OF LEMMA 9.

Suppose the lemma is false. Then we must have nodes A and B and a database state $G_i, c < i \leq k+1$ such that N separates A and B in G_i, $A, B \in G_j$, but N does not separate them in G_j, where, either $j = i+1$ (assuming $c < i \leq k$) or $j = i-1$ (assuming $c+1 < i \leq k+1$). We are able to assume that $\mid i - j \mid = 1$ due to the unique object identifier assumption, because otherwise, a node can be deleted from a database state G_i in which the lemma holds (does not hold) and may appear in a successive state $G_j, j - i > 1$, in which the lemma does not hold (holds).

First, consider the case when $j = i+1$ (assuming $c < i \leq k$). As $T_i^{'}$ inserts and deletes edges from G_i, let the database successively undergo the states, $G_i^1, \ldots, G_i^h = G_{i+1}$. When $h = 1$, the lemma is trivially true. We only consider the case when $h > 1$. Let G_i^{p+1} be the earliest database state in which the lemma is violated. Then we must have nodes A and B such that N separates A and B in G_i^p, but does not separate A and B in G_i^{p+1}. Let C_p and C_{p+1} be the nearest common dominators of A and B in G_i^p and G_i^{p+1} respectively.

This is possible only if there is a node $D \in N$ that is an ancestor of A or B in G_i^p and a descendant of C_p in G_i^p, but there is no node in G_i^{p+1} that is an ancestor of A or B and a descendant of C_{p+1} that belongs to N. Suppose, A is a descendant of D in G_i^p. (The proof when B is a descendant of D in G_i^p is analogous.) Due to Lemma 8, A is a descendant of D in G_i^{p+1} as well. Therefore, D must not be a descendant of C_{p+1} in G_i^{p+1}. Since D is an ancestor of A in G_i^{p+1}, and C_{p+1} dominates A in G_i^{p+1}, C_{p+1} must be a descendant of D in G_i^{p+1}. Therefore, there must exist a path from C_p to A (or B) in G_i^p that does not pass through C_{p+1} and T_i must have deleted an edge (D_1, D_2) on this path, so that after deletion of this edge, C_{p+1} is the nearest common dominator of A and B in G_i^{p+1}. Also, D_1 must be a descendant of C_p in G_i^p, and D_2 a proper descendant of C_{p+1} in G_i^p. But in order to delete the edge (D_1, D_2), $T_i^{'}$ must lock both D_1 and D_2, and also their nearest common dominator D_3 in G_i^p (due to locking rule L1 and Lemma 1(a)). D_3 could not be a descendant of D in G_i^p, because if that were the case, then D_3 also dominates A and B in G_i^p, and therefore, $C_p = D_3$, and D is not a descendant of C_p, which is a contrary to what we assumed. Therefore, D_3 must be an ancestor of D in G_i^p. (D_3 cannot be incomparable to D because it dominates a descendant D_2 of D.) Since $T_i^{'}$ locks D_2 and D_3, and D is an ancestor of D_2 and a descendant of D_3, it must lock D (due to Lemma 1(a)). But T_i cannot lock D because it belongs to N and is locked by $T_c^{'}$ in exclusive mode. Therefore, no such G_i^{p+1} exists.

Next, consider the case when $j = i-1$ (assuming $c+1 < i \leq k+1$). As $T_j^{'}$ inserts and deletes edges from G_j, let the database successively undergo the states, $G_j = G_j^1, \ldots, G_j^h = G_{j+1}$. When $h = 1$, the lemma is trivially true. We only consider the case when $h > 1$. Let G_j^{p+1} be the earliest database state in which the lemma is violated. Then we must have nodes A and B such that N separates A and B in G_j^{p+1}, but does not separate A and B in G_j^p. Let C_p and C_{p+1} be the nearest common dominators of A and B in G_j^p and G_j^{p+1} respectively.

This is possible only if there is a node $D \in N$ that is an ancestor of A or B in G_j^{p+1} and a descendant of C_{p+1} in G_j^{p+1}, but there is no node in G_j^p that is an ancestor of A or B and a descendant of C_p that belongs to N. Suppose, A is a descendant of D in G_j^{p+1}. (The proof when B is a descendant of D in G_j^{p+1} is analogous.) Due to Lemma 8, A is a descendant of D in G_j^p as well. Therefore, D must not be a descendant of C_p in G_j^p. Since D is an ancestor of

A in G_j^p, and C_p dominates A in G_j^p, C_p must be a descendant of D in G_j^p. Therefore, $T_j^{'}$ must create a path from C_{p+1} to A (or B), by adding an edge (D_1, D_2) in G_j^p, that does not pass through C_p, so that after insertion of this edge, C_{p+1} is the nearest common dominator of A and B in G_j^{p+1}. Also, D_1 must be a descendant of C_{p+1} in G_j^p, and D_2 a proper descendant of C_p in G_j^p. But in order to insert (D_1, D_2), $T_j^{'}$ must lock both D_1 and D_2, and also their nearest common dominator D_3 in G_j^p (due to locking rule L1 and Lemma 1(a)). D_3 could not be a descendant of D in G_j^p, because if that were the case, then D_3 also dominates A and B in G_j^p, and then $C_{p+1} = D_3$, and D is not a descendant of C_{p+1} in G_j^{p+1} contrary to what we assumed. Therefore, D_3 must be an ancestor of D in G_j^p. Since $T_j^{'}$ locks D_2 and D_3, and D is a descendant of D_3 and an ancestor of D_2, it must lock D (due to Lemma 1(a)). But $T_j^{'}$ cannot lock D because it belongs to N. Therefore, no such G_j^{p+1} exists. Hence the lemma must be correct.

END OF LEMMA 9. □

Lemma 10 *If A is an entity unlocked by $T_c^{'}$ and $A \in G_{c+1}$ then N separates A and A_k in G_{c+1}.*

PROOF OF LEMMA 10.

We first observe that $A_k \in G_{c+1}$, because due to Lemma 2, A_k is a successor of f in $G_c, \ldots, G_{k+1}$.

For contradiction, suppose N does not separate A and A_k in G_{c+1}. Let B be the nearest common dominator of A and A_k in G_{c+1}. Then none of the ancestors of A or A_k that is a descendant of B in G_{c+1} belongs to N.

Consider some path $\mathcal{P}$ between A and A_k in the underlying undirected graph of G_{c+1}, such that $\mathcal{P}$ consists of a directed path from B to A and a directed path from B to A_k. Thus, if we consider all such paths in G_{c+1}, they will include all ancestors of A or A_k in G_{c+1} that are descendants of B in G_{c+1}. We claim that after $T_c^{'}$ finishes execution, it holds a shared lock on all the nodes on $\mathcal{P}$ except A_k (to be proven below). Therefore, $T_c^{'}$ holds a shared lock on A. This is a contradiction, because we assumed that $T_c^{'}$ unlocked A and an item can be locked by a transaction only once (locking rule L3). Hence N must separate A and A_k in G_{c+1}.

It remains to be proven that after $T_c^{'}$ finishes execution, it holds a shared lock on all the nodes on $\mathcal{P}$ except A_k. The proof is by induction on the distance of a node from A_k along $\mathcal{P}$.

Base Case. Consider a node D such that its distance from A_k along $\mathcal{P}$ is 1. We claim that D must be an ancestor of A_k (to be proven below). Suppose $D = f$, which was locked by $T_c^{'}$. $T_c^{'}$ did not lock any node on $\mathcal{P}$ in exclusive mode (because N does not separate A from A_k in G_{c+1}), therefore it locks f in shared mode, and holds this lock after it finishes execution (by definition of f).

To complete the proof for the base case, it remains to be proven that B is a proper ancestor of A_k. This would follow if we can show that B is not equal to A_k.

For contradiction, suppose B is equal to A_k. Then A would be a descendant of A_k in G_{c+1}. Either A is a descendant of A_k in G_c also or $T_c^{'}$ adds an edge from a descendant of A_k in G_c to an ancestor of A in G_c so that A is a descendant of A_k in G_{c+1}. In the former case, $T_c^{'}$ must lock A_k in order to lock A, and in the latter case, it must lock A_k to lock a descendant of A_k in G_c (due to Lemma 1(b), and because T_c locks a predecessor f of A_k, and therefore, first node locked by T_c must be a proper ancestor of A_k in G_c). But T_c locks A_k after it has executed the prefix $T_c^{'}$ and a transaction can lock a node only once (rule L3). This is a contradiction. Therefore, A could not be a descendant of A_k in G_{c+1}. Hence B must be a proper ancestor of A_k in G_{c+1}, and therefore, D is also a proper ancestor of A_k in G_{c+1}. This proves the base case.

Induction Step. Suppose, after $T_c^{'}$ finishes its execution, it holds locks in shared mode on all nodes on $\mathcal{P}$ which are at a distance less than m, but greater than zero, from A_k along $\mathcal{P}$. Consider a node D that is at a distance m, along $\mathcal{P}$, from A_k. D can be either an ancestor of A_k in G_{c+1} or be incomparable to A_k in G_{c+1}.

First, consider the case when D is an ancestor of A_k in G_{c+1}. Let E be a node that is at a distance $m-1$ from A_k along $\mathcal{P}$ and $(D, E) \in G_{c+1}$. $T_c^{'}$ could not have added this edge because to do so would require locking E in exclusive mode (due to locking rule L1) and by induction hypothesis, E is locked in shared mode. The nearest common dominator of A and A_k in G_c, say C, could be either an ancestor or a descendant of E in G_c (it could not be incomparable to E because there is a path from E to A in G_{c+1} which also existed in G_c, because $T_c^{'}$ locks each node on this path in shared mode). If C is an ancestor of E in G_c, then since $T_c^{'}$ has locked E, it must have locked D (due to Observation 3). If C is a proper descendant of E in G_c, then $T_c^{'}$ must add an edge from an ancestor of E to a descendant of E so that the nearest common dominator of A and A_k in G_{c+1}, B, is an ancestor of E in G_{c+1}. In this case again, $T_c^{'}$ must have started execution by locking a node that is an ancestor of E in G_c, and therefore, D is a descendant of B_c. Therefore, to be able to lock a descendant of D, $T_c^{'}$ must lock D (Lemma 1(b)). Since N does not separate A and A_k in G_{c+1}, $T_c^{'}$ must lock D in shared mode. Thus, there is an undirected path from D to f, an entry point of the SCC to which A_k belongs, each node on which is locked by $T_c^{'}$ in shared mode. Therefore, this path must also exist in the database state G_{k+1}, and since T_c locks A_k in G_{k+1}, $T_c^{'}$ could not have released lock on D (locking rule L5b).

Next, consider the case when D is incomparable to A_k in G_{c+1}. Let E be a node that is at a distance $m-1$ from A_k along $\mathcal{P}$ and $(E, D) \in G_{c+1}$. D was a descendant of B in G_c as well, because its predecessor E is locked by $T_c^{'}$ in shared mode, and therefore, $T_c^{'}$ could not have inserted or deleted the edge (E, D). Furthermore, either D is an ancestor of A in G_c or $T_c^{'}$ adds an edge from a descendant of D in G_c to an ancestor of A in G_c. By induction hypothesis, $T_c^{'}$ locks E, and therefore, B_c is a proper ancestor of D (Lemma 1(a)). Thus, in either case, $T_c^{'}$ must lock D (due to Lemma 1(b)). Since N does not separate A and A_k in G_{c+1}, $T_c^{'}$ must have locked D in shared mode. Then using an argument similar to the one used in the previous paragraph we can show that $T_c^{'}$ could not have released lock on D.

Therefore, by induction, $T_c^{'}$ locks every node on $\mathcal{P}$ in shared mode and it has not released lock on it. Therefore, $T_c^{'}$ did not unlock A. This is a contradiction, because we assumed that $T_c^{'}$ unlocked A. Hence N must separate A and A_k in G_{c+1}.

END OF LEMMA 10. □

PROOF OF THEOREM 7 (CONTINUED).

Let B_i be the first entity locked by $T_i^{'}$. Let A be any entity unlocked by $T_c^{'}$. We prove that for each $i, c < k+1$, that if $A \in G_i$, N separates A from B_i in G_i. The proof is by induction on the length of shortest path in $D(S^{'})$ from $T_i^{'}, c < i \leq k$ to some $T_j^{'} \in \mathcal{K}$.

Base Case. The length of shortest path in $D(S^{'})$ from $T_i^{'}$ to some $T_j^{'} \in \mathcal{K}$ is zero. Consider one such transaction $T_i^{'} \in \mathcal{K}$. Due to Lemma 10, N separates A and A_k in the database state G_{c+1}. Then due to Lemma 9, N must also separate A and A_k in G_i. Since $T_i^{'}$ locks B_i and

A_k, due to Lemma 6, N does not separate B_i and A_k in G_i. Then, due to Lemma 7, N must separate A and B_i. This proves the base case.

Induction step. Suppose that N separates A and B_i in G_i for all $T_i^{'}, c < i \leq k$, such that the length of the shortest path in $D(S^{'})$ from $T_i^{'}$ to some $T_j^{'} \in \mathcal{K}$ is less than m. Next, we show that the same holds true for for all transactions $T_i^{'}$ such that the length of shortest path to some $T_j^{'} \in \mathcal{K}$ is m.

Consider a transaction $T_i^{'}, i > c$, such that the length of the shortest path in $D(S^{'})$ from $T_i^{'}$ to some $T_j^{'} \in \mathcal{K}$ is $m, m > 0$. Let $T_j^{'}$ be a transaction such that $(T_i^{'}, T_j^{'}) \in D(S^{'})$ (such a transaction must exist due to condition 3c of Theorem 6 since $T_i^{'} \notin \mathcal{K}$). Therefore, there is an entity locked by both $T_i^{'}$ and $T_j^{'}$. Let B be such an entity. By induction hypothesis, N separates B_j and A in G_j. Since N does not separate B and B_j in G_j (due to Lemma 6), it separates B and A in G_j (due to Lemma 7). Then, due to Lemma 9, N separates B and A in G_i as well. Since N does not separate B_i and B in G_i (due to Lemma 6), and separates B and A in G_i, it separates B_i and A (due to Lemma 7).

Hence, by induction, N separates A and B_i in G_i, for all $i, c < i \leq k$.

Let $T_i^{'}, c < i \leq k$ be a transaction that locks A_c. Such a transaction must exist due to condition 3c of Theorem 6. A_c is unlocked by $T_c^{'}$ (by condition 2 of Theorem 6). As proven above, N separates A_c and B_i in G_i. But $T_i^{'}$ also locks A_c, and therefore, N cannot separate A_c and B_i in G_i (Lemma 6). This is a contradiction. Hence such a schedule $S^{'}$ does not exist and the DDG-SX policy must be safe.

END OF THEOREM 7. □

2.6 Other Applications of Canonical Schedules

In this section, we consider two other locking policies, *altruistic* locking (Salem, Garcia-Molina and Shands 1994) and *dynamic tree* locking (Croker and Maier 1986) that are similar to the DDG policy in the sense that they can allow release of locks before a transaction reaches its locked point. We prove their correctness using Theorem 1 showing its power and generality. For both dynamic tree policy and altruistic locking, we assume that all transactions acquire only exclusive locks.

2.6.1 Altruistic Locking Policy

We first define the rules of the altruistic locking policy for a database that undergoes insertion and deletion of entities. We consider the basic version of altruistic locking in which all the locks are exclusive. A transaction T_i is said to be in the *wake* of another transaction T_j if T_i locks an item which has been unlocked by T_j, and T_j has not reached its own locked point. The rules of altruistic locking are as follows:

AL1. A transaction must acquire a lock on an item before performing any operation on it.
AL2. If a transaction T_i is in the wake of another active transaction T_j, then all items locked by T_i so far must have been unlocked by T_j in the past.
AL3. A transaction may lock an item only once.

As an example of the application of the altruistic locking consider the schedule shown in Figure 2.11. Once T_1 releases lock on node 1, T_2 is able to lock node 1, thus entering the

```
T1: (LX 1,2) (U 1)      (LX 3) (U 2)            (U *)
T2:             (LX 1)           (LX 2) (LX 5) (U *)
```

Figure 2.11: An example application of altruistic locking

wake of T_1. From now on, T_2 can lock only those nodes that have been unlocked by T_1. Once, T_1 reaches its own locked point, when it locks node 3, T_2 is no longer in the wake of T_1 and can lock any item it needs.

The advantage of altruistic locking relative to the DDG policy is that it does not impose any pre-specified order on the entities in the database. The disadvantage is that rule AL2 can be too restrictive, because to take advantage of the pre-release of locks a transaction has to be completely within the wake of another transaction. This may be relaxed, but only at the cost of several additional assumptions (Salem, Garcia-Molina and Shands 1994).

We now prove the correctness of altruistic locking using Theorem 1. Let us first give intuition into the structure of the proof. We begin by constructing a canonical non-serializable schedule involving transactions $T_1, \ldots, T_k$. By induction, we show that each of T_c+1$, \ldots, T_k$ must be in the wake of T_c. This leads to a contradiction (to rule AL2), because T_k has already locked an entity A_k which has not been locked by T_c in S'. The formal argument is as follows.

Theorem 8 *The altruistic locking policy is safe.*

PROOF OF THEOREM 8.

Suppose the altruistic locking policy is not safe. Then choose transactions $T_1, \ldots, T_k$, entities $A_1, \ldots, A_k$ and construct the corresponding schedule S' as in Theorem 1. The serialization graph $D(S')$ is a directed acyclic graph over $T_1', \ldots, T_k'$.

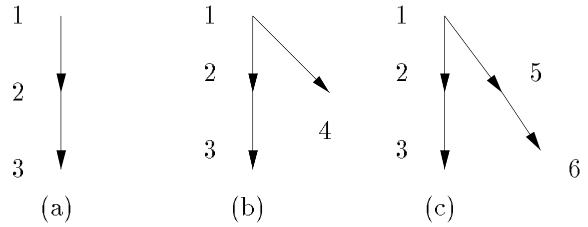
We prove that in the schedule S', T_i' must be in the wake of T_c. The proof is by induction on the length of the shortest path from T_c' to $T_i', c \leq i \leq k$ in $D(S')$.

Base case. The length of the shortest path from T_c' to T_i' in $D(S')$ is equal to 1. Since T_c' and T_i' are adjacent in $D(S')$, T_c' and T_i' must lock an entity in common. Let A be an entity that is unlocked by T_c' and is locked by T_i'. In the prefix T_c', T_c has not reached its lock point, because it locks A_k after it has executed the prefix T_c' (condition 2 of Theorem 1). Since T_i' locks A, which has been unlocked by T_c', and T_c has not reached its locked point in S', T_i' must be within the wake of T_c (locking rule AL2) in S'. This proves the claim for the base case.

Induction Step. Suppose the claim is true for all transactions T_i' such that the length of the shortest path from T_c' to T_i' in $D(S')$ is less than m.

Consider a transaction T_j' such that the length of the shortest path from T_c' to T_j' in $D(S')$ is m. From $D(S')$, pick a transaction T_i' such that the length of the shortest path from T_c to T_i' in $D(S')$ is $m-1$ and $(T_i', T_j') \in D(S')$. Such a transaction exists because of condition 3c of Theorem 1. Let A be an entity that is unlocked by T_i' which is later locked by T_j'. Such an entity exists because of the edge (T_i', T_j') in $D(S')$. By induction hypothesis, T_i' is in the wake of T_c in S'. Therefore, all entities locked by T_i' (including A) have been unlocked by T_c'. Since T_j' locks A, T_j' must be completely within the wake of T_c as well (locking rule AL2).

Hence by induction, for all transactions $T_j', c < j \leq k$ T_j' is in the wake of T_c.



T_1: `(LX 1,2) (U 1)        (LX 3) (U 2)                                     (U *)`
T_2: `          (LX 1,4) (U 4)        (LX 2) (U 1,2)`
T_3: `                                         (LX 1,5,6) (U 5,6) (LX 2) (U *)`

Figure 2.12: An example application of Dynamic Tree Policy

Thus, $T_k^{'}$ is in the wake of T_c. Since $T_k^{'}$ locks A_k, it must have been locked and then unlocked by T_c in its prefix before (L A_k). But we know that T_c also locks A_k in the step immediately following $T_c^{'}$ which is a contradiction (since each transaction may lock an entity only once, by locking rule AL3). Hence the assumed canonical non-serializable schedule does not exist and the altruistic locking policy is safe.

END OF THEOREM 8. □

2.6.2 Dynamic Tree Policy

The Dynamic Tree (DTR) Policy was proposed to allow a changing set of partial orders to be defined over the objects in a database (Croker and Maier 1986). Thus, instead of assuming that we are given a directed graph corresponding to the database, the DTR policy defines a forest for itself. To distinguish the forest defined by the DTR policy from the directed graph that might exist in the database itself, we call the former a *database forest*. We first give some definitions and then define the rules of the DTR policy for the case when all locks are exclusive.

We say that a well-formed transaction T is *tree-locked* with respect to a forest G if except for the first lock step, each (L A) step is preceded by a lock step (L B) and followed by an unlock step (U B) where B is the predecessor of A in G. Furthermore, a tree-locked transaction may lock an entity only once.

Recall that $R(T)$ is the set of entities referenced by a transaction. Let $A(T)$ be the set of entities for which there is an explicit `ACCESS, INSERT` or `DELETE` operation in T. These two sets can be different because a transaction locks some entities only to satisfy the rules of the locking policy. Let $G(E)$ be the forest obtained by deleting from G the entity E.

The rules of the DTR policy can be defined as follows:

DT0. Initially the database forest G is empty.
DT1. To join two database trees $G_1, G_2 \in G$, draw an edge from the root of G_1 to the root of G_2. To add a set of entities to a tree G_1, first connect them to form a tree G_2 and then join G_1 and G_2.
DT2. At the beginning of an active transaction T, join all the trees that contain

some entity $v \in A(T)$ to form a single tree G. Add to G all the entities in $A(T)$ that are not already in it. Tree-lock T with respect to G.
DT3. A node E may be deleted from the database forest if it is not currently locked by any active transaction and for each active transaction T, T is tree-locked with respect to some $G_i \in G(E)$.

As an example of the application of the DTR policy, consider the schedule shown in Figure 2.12. When T_1 begins execution, the database forest is as shown in Figure 2.12(a) (DT0, DT2). Since T_2 accesses node 4, it is added to the database forest as shown in Figure 2.12 (DT1, DT2). Once T_2 finishes execution, node 4 can be deleted from the database forest, since T_1 will still be tree-locked with respect to $G(4)$. The execution of T_3 can be explained in a similar fashion.

Let us highlight the differences between the DDG policy and the DTR policy. First, in the DDG policy, the database graph is assumed to be given and the transactions may insert or delete entities from it. In contrast, in the DTR policy, the database forest is created by the concurrency control algorithm, and insertions into and deletions from the graph are caused by the algorithm and not by the transaction. Second, in the DTR policy, it is necessary to pre-compute the locked transactions when a transaction begins which is not the case for the DDG policy. Finally, in the DDG policy, the database is an arbitrary rooted and connected graph, whereas in the DTR policy, the database is a forest.

We observe that Lemma 1 applies to the DTR policy as well. Thus, if B is the first node locked by a transaction T in a database forest G, then in the absence of any other concurrent transactions, all nodes locked by T are dominated (or are descendants) of B in G. Similarly, in the absence of any other concurrent transaction, for each A in $R(T)$, all nodes that are ancestors of A and descendants of B in G, have been referenced by T when it locks A.

Let us now prove the correctness of this locking policy using Theorem 1. Before we proceed, we give some intuition into the structure of the proof. We begin by constructing a canonical non-serializable schedule involving transactions $T_1, \ldots, T_k$. Then by induction, we show that each transaction $T_{c+1}, \ldots, T_k$ begins by locking a node that is a descendant of A_k in the database state in which it begins execution. The argument is based on Lemmas 11–12 and the properties of the tree policy as stated in Lemma 1. We eventually encounter a contradiction because then a transaction T_i that is adjacent to T_c in $D(S^{'})$ cannot lock an in entity common with T_c. The formal argument is as follows.

Theorem 9 *The Dynamic Tree policy is safe.*

PROOF OF THEOREM 9.

Suppose the dynamic tree locking policy is not safe. Then choose transactions $T_1, \ldots, T_k$, entities $A_1, \ldots, A_k$ and construct the corresponding schedule $S^{'}$ as in Theorem 1. The serialization graph $D(S^{'})$ is a directed acyclic graph over $T^{'}_1, \ldots, T^{'}_k$.

For $1 \leq i \leq k$, let B_i be the first entity locked by T_i and let G_i be the state of the database forest at that instant. Let G_{k+1} be the state of the database forest when T_c performs its (L A_k) step.

Since A_k is not the first node locked by T_c (condition 2 of Theorem 1), T_c must hold a lock on a predecessor f of A_k in G_{k+1}. This is because T_c is tree-locked with respect to G_c (due to rule DT2) and is also tree-locked with respect to G_{k+1} (due to rule DT3). Before we proceed further, we need the following lemmas.

PROOF OF THEOREM 9 (TO BE CONTINUED).

Lemma 11 *A_k is a successor of f in the database forests $G_{c+1}, \ldots, G_{k+1}$.*

PROOF OF LEMMA 11.

First, we show that A_k and f belong to database forests $G_{c+1}, \ldots, G_{k+1}$. Since T_c holds a lock on f since it executed its prefix $T_c^{'}$, f cannot be deleted from the database forests $G_{c+1}, \ldots, G_{k+1}$ (rule DT3). Similarly, A_k cannot be deleted from the database forests $G_{c+1}, \ldots, G_{k+1}$ because then T_c will not be tree-locked with respect to one database tree (rule DT2, DT3). Hence, A_k and f belong to database forests $G_{c+1}, \ldots, G_{k+1}$.

Next, we prove by backward induction on $i, c < i \leq k+1$, that A_k is a successor of f in the database forests $G_{c+1}, \ldots, G_{k+1}$.

Base Case. $i = k+1$. Follows from the definition of f.

Induction step. Suppose, A_k is a successor of f in database forests $G_{m+1}, \ldots, G_{k+1}$, where $m > c$.

Now let us prove that A_k is a successor of f in G_m. For contradiction, suppose that A_k is not a successor of f in G_m. This can only happen if A_k is first deleted from the database forest G_m and then re-inserted as a descendant of f in G_{m+1} using rule DT1 (assuming f is a root of G_{m+1}). But A_k cannot be deleted from the database forest, because then T_c, which is an active transaction, is not tree-locked with respect to a single database tree (rules DT2, DT3). Hence f must be a predecessor of A_k in G_m.

END OF LEMMA 11. □

Lemma 12 *If a node A is a proper descendant of A_k in some $G_i, c < i \leq k+1$, then for all $j, c \leq j < i$, if $A \in G_j$, A is a proper descendant of A_k in G_j.*

PROOF OF LEMMA 12.

For contradiction suppose the lemma is not true. Consider the smallest $j, c \leq j \leq k$, such that $A \in G_j$, A is not a descendant of A_k in G_j, but A is a descendant of A_k in G_{j+1}. The only way to change the position of A in G_j is to first delete A from G_j and then re-insert it as a descendant of the root (rule DT1). Still A cannot be a descendant of A_k in G_{j+1}, because A_k being a descendant of f in G_{j+1} (by Lemma 11) is not a root in G_{j+1}. Hence the claim of the lemma must be true.

END OF LEMMA 12. □

PROOF OF THEOREM 9 (CONTINUED).

For $1 \leq i \leq k$, let B_i be the first entity locked by T_i. We prove that B_i is a descendant of A_k in G_i, for $c < i \leq k$. The proof is by backward induction on $i, c < i \leq k$.

Base case. $i = k$. Since f is a predecessor of A_k in G_k (due to Lemma 11), and T_k could not have locked f (because it is locked by T_c), and it locks A_k, A_k must be the first entity locked by T_k (because T_k is tree-locked). This proves the claim for the base case.

Induction step. Suppose B_i is a descendant of A_k for all $i, c < m < i \leq k$.

Consider the transaction T_m. Suppose, for contradiction, that B_m is not a descendant of A_k in G_m.

We first prove that $T_m^{'}$ did not lock any descendants of A_k in G_m. For contradiction, suppose $T_m^{'}$ locked a descendant A of A_k in G_m. Then $T_m^{'}$ must also lock f because of Lemma 1(b). But $T_m^{'}$ cannot lock f because it is already locked by T_c. Hence $T_m^{'}$ cannot lock any descendant of A_k in G_m.

Next, from $D(S')$, pick a transaction T_j' such that $(T_m', T_j') \in D(S')$ (such a transaction must exist due to condition 3c of Theorem 1 and because $m \neq k$). By induction hypothesis, B_j is a descendant of A_k in G_j. Therefore T_j' locked only descendants of B_j, and hence only descendants of A_k in G_j (by Lemma 1(a)). The descendants of A_k in G_j were also descendants of A_k in G_m if they existed in G_m (due to Lemma 12). Thus, T_j' locked only descendants of A_k in G_m.

Thus, we have shown that T_m' did not lock any descendant of A_k in G_m and T_j' locked only descendants of A_k in G_m, implying that $R(T_m') \cap R(T_j') = \emptyset$. This is a contradiction because $(T_m', T_j') \in D(S')$. Hence B_m must be a descendant of A_k in G_m.

Hence, by induction, B_i is a descendant of A_k in G_i for all $c < i \leq k$.

Let $T_i', c < i \leq k$ be a transaction that locks A_c (such a transaction must exist because of condition 3c of Theorem 1). As proven above, B_i is a descendant of A_k in G_i. Therefore, A_c is also a descendant of A_k in G_i and also in G_c (due to Lemma 12). Since T_c' locks a parent of A_k and also locks A_c which is a descendant of A_k, it must also lock A_k (due to Lemma 1(b)). But we know that T_c locks A_k in the step immediately following T_c'. This is a contradiction, because a tree-locked transaction may lock an entity only once. Hence the DTR policy must be correct.

END OF THEOREM 9. □

The proofs of this section and the previous section demonstrate the usefulness of the canonical schedules theorem as a general tool for proving the correctness of locking policies.

2.7 Related Work

In this section, we review some work related to the results presented in this chapter.

2.7.1 Other Algorithms on Graph-Structured Data

The result of Theorems 1 and 6 are an extension of a previous result that did not deal with insertions and deletions in the database (Yannakakis 1982a). The DDG policy with exclusive locks is an extension of tree policy that dealt with only static trees (Silberschatz and Kedem 1980). The DDG-SX policy extends a similar policy that dealt with only static graphs (Kedem and Silberschatz 1983).

There are several other papers studying concurrency control algorithms for graph-structured data (Kedem and Silberschatz 1981; Fussell, Kedem and Silberschatz 1981; Silberschatz and Kedem 1982; Mohan, Fussell and Silberschatz 1982; Kedem, Mohan and Silberschatz 1982; Silberschatz 1983; Buckley and Silberschatz 1984; Mohan et al. 1985; Silberschatz and Buckley 1985). As pointed out earlier, these papers deal with graphs that cannot undergo insertions and deletions of nodes and edges. Furthermore, they deal only with acyclic graphs except in one case (Silberschatz and Buckley 1985), which assumes knowledge of starting points — that is, the data items from which a transaction may begin execution. The starting points of a transaction are known in advance and are fixed. To allow release of locks before the locked point, the set of starting points has to be a proper subset of the database. If the set of starting points is equal to the database, that is a transaction may begin anywhere, no release of locks before the locked point is possible. In our model, there is no such assumption. Another major difference in our work from the work cited above is that in the DDG policy, a transaction may determine its access set at run-time. This is not possible in any of the above approaches.

There have also been some attempts to study the dynamic versions of tree policy. For example, the Dynamic Tree policy considered in Section 2.6.2 is one example of such an approach. Another approach is to restrict the set of operations on the tree such that the tree locking policy still produces serializable schedules (Lanin and Shasha 1990). Our approach is more general than these, because we deal with general graphs and do not impose any restrictions on the operations that may be performed on the database.

The multiple-granularity DAG algorithm also views the database as a directed graph (Gray et al. 1976). The database graph represents different levels of granularities present in the database, for example, record, file, segment, etc, and is used to support two-phase locking on these granules. In contrast, in our approach, the database graph represents the semantic relationships between the entities in the database, and supports non-two-phase locking at a single level of granularity. Work on adaptation of multiple granularity DAG for the KBMS environment is in progress elsewhere (de Ferreira Rezende and Härder 1994).

2.7.2 Concurrency Control Algorithms for B-Trees

We first describe a basic concurrency control algorithm for B-Trees and then discuss its relationship to our work.

A B-Tree is a commonly used index structure in database systems. To take advantage of its graph-like structure, several special-purpose algorithms have been developed to support concurrent access. In a B-Tree there are three types of transactions: *search*, *insert* and *delete*. Unlike the transactions in a knowledge base, all transactions in a B-Tree always begin from the root.

One of the common themes in the concurrency control algorithms for B-Trees is *lock-coupling*. A transaction is said to lock-couple when it requests a lock on a node A, such that A is not a root, while holding a lock on its predecessor, and releases the lock on the predecessor as soon as A has been locked. An example B-Tree algorithm works as follows (Bayer and Schkolnick 1977). A search transaction begins by getting a shared lock on the root and lock-couples to a leaf using shared locks. An insert transaction begins by getting an exclusive lock on the root and lock-couples to a leaf using exclusive locks. Lock-coupling using exclusive locks may cause contention, and there are several variations of this algorithm to correct this problem.

There are three differences between our work and the above algorithm. First, the graph corresponding to a database may not be always a tree. Second, transactions in a database may not always start at the root and may not always progress in the root to leaf order, for example, in depth-first search. In such situations it is not possible to use lock-coupling as an implementation technique. Finally, the transactions on a B-Tree consist of a single operation (insert/delete/search), whereas the transactions in a knowledge base may consist of an arbitrary sequence of operations.

2.8 Chapter Summary

In this chapter, we started by presenting a model of a dynamic database, that is, a database that may undergo insertions and deletions of entities. This required us to distinguish between the *structural* and *value* states of the database. We defined the notion of *proper* schedules which formalizes all sensible interleavings of transactions. To take into account the dynamism in the database, we defined a *locking policy as a function*, in contrast to its definition in a static database, where it was defined as a relation.

Inspired by the rich semantic structure of knowledge bases, we viewed the knowledge base as a directed graph. We developed the Dynamic Directed Graph (DDG) policy that can deal with general graphs that may undergo insertions and deletions of nodes and edges (Section 2.3). We considered several variants of this policy (Section 2.5), for example, the DDG-SX policy, that deals with both shared and exclusive locks. We analyzed the correctness (Theorems 2 and 7), deadlock-freedom (Theorem 3) and well-structured-ness (Theorem 4) of the DDG policy.

Another important result presented in this chapter was the generalization of the canonical schedules theorem for dynamic databases (Theorems 1 and 6). The canonical schedules theorem made it convenient to analyze the correctness of locking policies for dynamic database. To demonstrate the utility of such a technique, we used it several times in this chapter: to prove the correctness of the DDG and the DDG-SX policies (Theorems 2 and 7) and of two other locking policies (Theorems 8 and 9) that were developed by other researchers.

Chapter 3

Implementation of the DDG Policy

In this chapter we describe the implementation of the DDG policy. We consider the version of the DDG policy that supports both shared and exclusive locks (ie., DDG-SX policy). For the rest of the thesis, we use “DDG” to denote “DDG-SX” policy in the interest of brevity. Consequently, wherever we refer to the locking rules (for example, L1, L2, etc.), we assume that they correspond to the locking rules of the DDG-SX policy.

The implementation is done in a simulation environment. The system component that implements the DDG policy is called the Concurrency Control Manager (CCM). In the next chapter we will discuss the simulation model in more detail and describe how CCM interrelates with the other system components. For the purposes of this chapter, the implementation ideas are presented in a way that is independent of any specific system.

The role of the present chapter in the overall results of this thesis is quite an important one. This is because even though the DAG policy has been known for quite some time and there has been considerable theoretical analysis of related algorithms (see Section 2.7.1), no work has been done to understand the implementation issues. Implementing the DDG policy posed several technical problems, some of which could be addressed with little effort, whereas, others required research effort on our part. For example, to compute dominator relationships we were able to use existing algorithms. But there was no algorithm available for incrementally computing the strongly connected components and an algorithm for this purpose had to be developed.

Another contribution of this chapter is that we show how a non-two-phase locking policy can be integrated into a conventional lock manager. We specify the data structures that need to be maintained and a mechanism that can be used to release locks before the locked point of a transaction. The scheme that we propose is more general than lock-coupling (see Section 2.7.2 for definition), which is often used in the implementation of non-two-phase locking policies. We also make explicit the assumptions that we need to make about the transactions. For example, previous work in this area assumed that before a transaction begins, we need complete knowledge of the data items that a transaction is going to access. Since in knowledge bases, the access patterns are restricted, (for example, depth-first traversal), it is possible to make a weaker assumption: we need to know the first node to be accessed by a transaction, and the number of *levels* in the graph (see Section 3.4.1 for definition) it is going to access.

We begin with an overview of the implementation and define terminology and notation. We consider the processing of the knowledge base and the transactions. We give an alternative implementation to reduce the preprocessing cost and conclude with a summary.

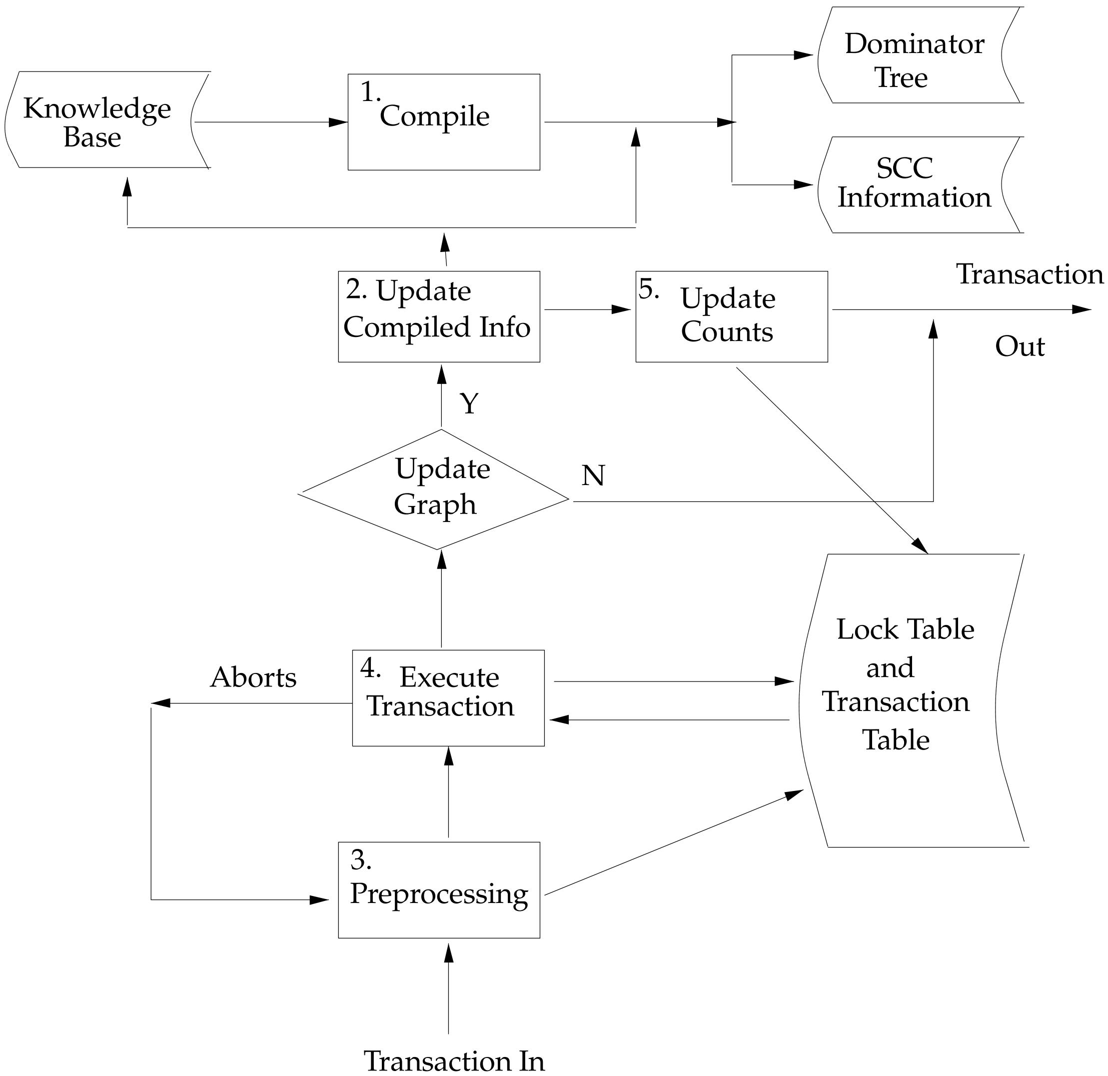


Figure 3.1: Overview of implementation

3.1 Overview of the Implementation

In implementing the DDG policy, we had to address two kinds of issues. First, we had to compute properties of the graph, such as dominator relationships and strongly connected components, that are used by the locking policy. Second, we had to devise a mechanism for releasing locks before the locked point of a transaction. We can divide the operations in our implementation along these two lines: processing of the knowledge base and processing of transactions. Figure 3.1 shows an overview of the implementation.

When the knowledge base is compiled, we generate information on strongly connected components and dominator relationships (Box 1 in Figure 3.1). (The utility of this information will be explained shortly.) Every time there is a transaction that performs insertions and deletions to the knowledge base, the information on strongly connected components and dominators is (Box 2 in Figure 3.1).

The processing of transactions has three phases. First, when a transaction begins, we perform preprocessing and create entries in the transaction table (see Box 3 in Figure 3.1).

Second, we have run-time checks that are used as the transaction executes (Box 4 in Figure 3.1). The mechanism for releasing the locks before commit time is implemented as part of these run-time checks. Finally, when the transaction commits, if it has updated the graph it triggers an update of the compiled information about the knowledge base (Box 2 in Figure 3.1) and of the information about active transactions (Box 5 in Figure 3.1).

3.2 Terminology and Notation

While discussing the computational complexity of an algorithm for a graph G, we use n to denote the number of nodes and m to denote the number of edges of G.

We describe our algorithms using high level pseudo-code. We use two standard loop constructs, *foreach* and *while*, usual *if-then-else* conditionals and procedure calls. In addition, we use the constant symbols, **true**, and **false**, which have their usual semantics. The statement **exit** signals an unconditional termination of a loop. (When **exit** appears in a loop which is nested inside one or more other loops, then it implies exit from the innermost loop.)

3.3 Processing of Knowledge Base

In the implementation of the DDG policy we need information about the following properties of the knowledge base:

- parent-child relationships
- strongly connected components
- dominator relationships

Information on parent-child relationships is used to enforce the rule L5 of the DDG policy. We need to know the strongly connected components to implement locking rules L4 and L5. Information about dominator relationships is necessary, because to be able to lock all the items that a transaction needs, it must begin by locking a node that dominates all the nodes that it is going to access.

If the knowledge base is stored as an adjacency list, it is straightforward to determine the parent-child relationships. The computation of strongly connected components and dominator information, however, is a more involved process. In the next two subsections, we explain the compile-time and run-time algorithms adopted for this purpose.

3.3.1 Compiling the Knowledge Base

The algorithms described in this section correspond to Box 1 in Figure 3.1. More details about the compile-time processing of the knowledge base are shown in Figure 3.2. We first compute strongly connected components and then the dominator information.

Compile-time Computation of Strongly Connected Components

In the first step (Line 3, Figure 3.2), we compute the strongly connected components (SCCs) of the graph. The procedure *DepthFirstSearch* implements this using a depth-first search

```
1 funct PreProcessKB(inputKB) ≡
2       Using Depth-first search, compute SCCs of inputKB and condense each
3       non-trivial SCC into one node to get condensedKB
4       Assign a priority index to each node in the condensedKB
5       Construct the dominator tree of the condensedKB
6 end
```

Figure 3.2: Processing of knowledge base at compile time

algorithm which has a complexity of $O(m)$ (Aho, Hopcroft and Ullman 1987). The procedure *DepthFirstSearch* takes a directed graph *inputKB* as input, also called the *original graph*, and produces another graph, called the *condensed graph*, in which each SCC is condensed into a single node and is assigned a new non-zero node identifier. The original graph and the condensed graph co-exist in the system at all times. Whenever there is a change in the original graph, the condensed graph is updated to reflect the changes. In the subsequent discussion, whenever we refer to the underlying graph, we assume that it refers to the condensed graph.

In the procedure *DepthFirstSearch*, we also associate an *SCC index* with each node. If a node is not an SCC by itself, the SCC index is the node identifier of the condensed node corresponding to this SCC. If a node is an SCC by itself, its SCC index is 0.

From now on, if a node A is not an SCC by itself, we implicitly assume that its identifier is mapped to the identifier of the condensed node corresponding to the SCC to which it belongs. This transformation is performed every time a node is accessed, but we do not show it in any of the algorithms in the interest of brevity.

The next step in the compilation of the knowledge base is to compute a *priority index* for each node in the condensed graph (Line 4). The priority index of each node is drawn from a totally ordered set R, so that if there is an edge (A, B) in the condensed graph, then the priority index of A is less than the priority index of B. We implement this in the procedure *AssignPriority*, based on the algorithm presented elsewhere (Dietz and Sleator 1987). We later use this index in the incremental computation of strongly connected components.

Compile-time Computation of Dominators

The last step in the compilation of the knowledge base is to compute the *dominator tree* of the condensed graph (Line 5). The dominator tree is defined as follows:

Definition 1 *Dominator Tree: The dominator tree of a directed acyclic graph* $G = (V, E)$ *is a tree* $T = (V, E^{'})$ *such that* $(A, B) \in E^{'}$ *if and only if* A *dominates* B *in* G *and no proper descendant of* A *in* G *satisfies this property.*

In our implementation, we have used a bit vector algorithm to compute the dominator tree (Aho, Sethi and Ullman 1986). We implement this in the procedure *ConstructDomTree* (see Figure 3.2). The bit vector algorithm has a running time complexity of $O(n^2)$. It is possible to use a more efficient algorithm with a running time of $O(m\alpha(m, n))$, where $\alpha(m, n)$ is a functional inverse of Ackermann's function (Lengauer and Tarjan 1979). We chose the bit vector algorithm, because for the size of the problem we were trying to solve, the advantage of the simplicity of the bit-vector approach far outweighed the disadvantage of its greater computational cost.

3.3.2 Run-time Processing of the Knowledge Base

The run-time processing corresponds to Box 2 in Figure 3.1. Every time there is an insert/delete in the knowledge base, we update the dominator tree and the condensed graph using *incremental* algorithms. In this section, we explain the algorithms used for the run-time processing of the knowledge base.

Incremental Computation of the Dominator Tree

To incrementally compute the dominator tree, one can use an already known algorithm (Caroll 1988). In our implementation, however, we took a shortcut approach and implemented a variation of the bit vector algorithm that we used for compile-time computation. Our algorithm incrementally computes the dominator tree when the insertions and deletions are at the leaf level but it recomputes the dominator tree from scratch for all other updates. Since most updates occur at leaf nodes, and our knowledge base was not very big, this approach sufficed for our purposes.

Incremental Computation of Strongly Connected Components

There is no known algorithm to incrementally compute the strongly connected components, and therefore, we developed an algorithm to perform this task. This algorithm was, however, not implemented. The algorithm, *UpdateSCC*, is shown in Figure 3.3. This algorithm takes the original graph (*originalKB*), condensed graph (*condensedKB*) and a list of updates containing insertions and deletions (*updateList*) as input, and produces an updated condensed graph.

Given the original knowledge base and a list of updates we compute the updated original knowledge base. We first process each deleted edge by iterating over the list *updateList* (Lines 2-9). If the two end points of an edge belong to different SCCs, the deletion of the edge cannot have any effect on the SCC information. On the other hand, if the end points of an edge are from the same SCC, G_i, we perform a depth first search on a subgraph of the updated original graph. If the resulting condensed graph is the same as its condensed graph before the update (which was a single node), there has been no change in the SCCs. If the resulting condensed graph contains more than one node, this means that the SCC has been split into smaller components. We treat the nodes and edges in the new SCC as insertions into the condensed graph and process them like any other insert operation. Thus, we use a compute-from-scratch approach inside the strongly connected components. The efficiency of this approach depends on the size of the SCCs being searched. For the applications that we considered, the maximum size of a strongly connected component was 22, and thus, we did not expect that such an approach would cause a serious problem.

Next, we process the edge insertions using an incremental approach (Lines 10-20) (Alpern et al. 1990). Using a procedure called *discovery*, we first compute a subgraph of the condensed graph, called *cover*, which has the property that all cycles in the graph are in the cover (After edge insertions, the condensed graph may no longer be acyclic.) Then we perform a depth-first search on this cover to compute a condensed cover. Finally, we merge this cover with the old condensed graph and reassign the priorities for the new condensed graph graph using a priority re-assignment algorithm. The incremental complexity of this algorithm was not computed and is left as a problem for future research.

```
funct UpdateSCC(originalKB, condensedKB, updateList)
      foreach edge that was deleted by the transaction do
         if its ends points are in the same SCC then
            Search the old SCC to check if there is any change in it
            if there has been a change in the SCC then
               Treat the new SCC as an insertion to the knowledge base
            fi
         fi
      od
      foreach inserted edge in the updateList do
         if its end points are not in the same SCC then
               Save it in edgeList that contains all the edges
               that have potential to change the SCCs
         fi
      od
      Compute the cover of edgeList using a discovery algorithm where
      cover contains the maximal set of nodes affected by the change
      Using depth-first search compute the condensed cover
      Merge the condensed cover with the knowledge base and
      recompute the priorities
end
```

Figure 3.3: Incremental computation of strongly connected components

3.4 Processing of Transactions

We begin this section by explaining our assumptions about transactions. We then describe the data structures maintained for processing of transactions. In the next three subsections, we describe three phases in the processing of a transaction: preprocessing, execution and commitment (corresponding to Boxes 2, 3 and 4 respectively in Figure 3.1). Finally, we describe an alternative implementation which reduces the preprocessing of transactions.

3.4.1 Assumptions About the Transactions

In our implementation, we deal with two classes of transactions: Class 1 and Class 2. Class 1 transactions are usually long and involve traversal along some semantic relationship of the knowledge base, for example, `isA` or `PartOf`. Class 2 transactions are short and perform lookup or update of attribute values. Occasionally they may also change some semantic relationship in the knowledge base.

For Class 1 transactions, we assume that when a transaction begins, it specifies the node from which it begins the traversal and how many *levels* in the graph it is going to traverse, where number of levels is defined as follows. If A is the first node locked by a transaction T, and B is some other node locked by T, then the level number of A is the length of the shortest path from A to B plus one. For Class 2 transactions, we assume that we have complete knowledge of the entities that a transaction will access. In Appendix C, we show the validity of these assumptions by considering sample transactions from a real application. More details of the application are given in Section 5.1.1.

TxnInfo:
- TID
- requestList
  - source
  - target
  - requestType
- class
- dominator
- locks
- currrentRequest
- pendingRequestList
- UnlockTable
- waitsFor

Table 3.1: Transaction table record

3.4.2 Data Structures Maintained by the Concurrency Control Manager

As shown in Figure 3.1, in our implementation, we maintain a transaction table and a lock table. In this section, we describe the information stored in these two tables.

Transaction Table

For each active transaction, an entry called *TxnInfo* is created in the transaction table. When the transaction commits, this entry is deleted. Table 3.1 shows the fields of an entry in the transaction table. The first field *TID* is a unique identifier assigned to a transaction when it begins execution and also serves as the key for the transaction table. The next field *requestList* stores the list of all items to be accessed by a transaction. A transaction may add or delete entries from this list as it is executing. The requestList consists of a sequence of access records. Each access record has three fields: *source, target* and *requestType*. For an access on an edge (A, B), the source field is set to A and the target field is set to B. If the access is on a node B then the source is set to B and the target to 0. The *requestType* denotes the type of the request and its possible values are `READ, WRITE, INSERT` and `DELETE`.

The next two fields, *class* and *dominator* store the class and the dominator information for the transaction. The *class* information is provided by the transaction when it begins. The *dominator* is computed by the concurrency control manager at the transaction initiation time and is the dominator of the nodes to be accessed by the transaction. The next field *locks* is a list of lock records and stores the list of locks owned by a transaction. The details of a lock record will be described later in this subsection. The field *currrentRequest* stores the current request of the transaction to the CCM. The structure of this field is the same as the structure of an access request except that the *requestType* can take some additional values — `INITTRANS, COMMIT` and `ABORT` — indicating initiation, commit and abort requests respectively.

The list *pendingRequestList* stores the items that need to be locked in the current call to the concurrency control manager. This list is necessary, because even though a transaction may have requested lock on just one item, it may require locking several items in order to enforce the rules of the locking policy. The last field in the transaction record is *UnlockTable*,

LockRec:	UnlockRec:
nodeNo	TID
lockType	count
lockers	neededByTransaction
waiters	lockType
onePredecessor	locksSuccessors
UnlockTable	unlockable

Table 3.2: Data structures maintained in the lock table

and we describe this later in this subsection. The last field *waitsFor* stores a pointer to the lock which that transaction is waiting to acquire. This field is used in deadlock detection. If the transaction is not waiting, then the value of this field is **nil**.

Lock Table

The lock table contains a collection of lock records. The fields of a lock record are shown as *LockRec* in the left side of the Table 3.2. The first field in LockRec is *nodeNo* indicating the identifier of the node on which this lock is placed. The field nodeNo serves as the key for the lock table. The second field, *lockType*, indicates the type of lock and can take two values: `SHARED` or `EXCLUSIVE`. The field *lockers* maintains the list of transactions that currently hold this lock and *waiters* maintains the list of transactions that are waiting for this lock. Each entry in lockers and waiters points to the corresponding transaction record in the transaction table.

In order to satisfy the locking conditions for a node A, a transaction should currently be holding a lock on at least one of the entry points of A (unless this is the first node to be locked by the transaction). The field *onePredecessor* stores such an entry point. For example, for the graph shown in Figure 3.4, onePredecessor for node 5 may be node 2.

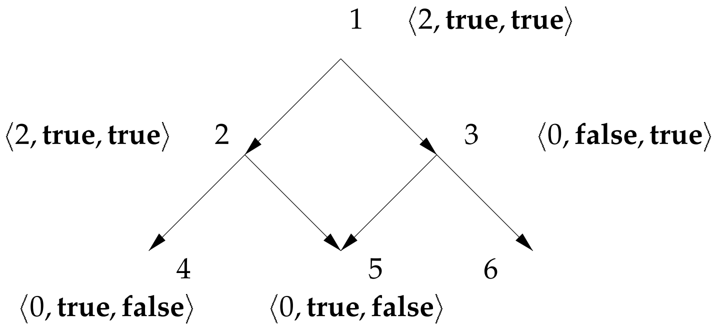
The last field in LockRec is another table consisting of a collection of *UnlockRec*s. The CCM uses the information in an UnlockRec for making unlocking decisions, hence the name, *UnlockTable*. There is an UnlockTable corresponding to each LockRec, thus, implicitly associating each UnlockRec to the nodeNo of the LockRec. This data structure is an extension to the conventional concurrency control manager (such as the one for two-phase locking) which does not need to maintain this information.

The structure of an UnlockRec is shown in the right side of Table 3.2. The first field of the UnlockRec is *TID* and is the key for unlock table. To specify an UnlockRec we need to specify a node identifier and the transaction identifier. To locate the appropriate UnlockRec, we first locate the LockRec with the given node identifier, and then in the UnlockTable of that LockRec, we locate the UnlockRec with the given transaction identifier. From now on, we do not explicitly specify the node identifier and the transaction identifier for an UnlockRec as long as their values are clear from the context.

The second field *count* is defined as follows:

Definition 2 *For a node A and a transaction T, $count(T, A)$ is the number of successors of A that will be locked by T and will have A as their onePredecessor.*

As an example, consider transaction T_1 on the knowledge base shown in Figure 3.4. The *count* for node 2 is 2 because both nodes 4 and 5 have node 2 as their onePredecessor. The



T_1: (R 1) (W 2) (R 5) (R 4)
Each node is labeled by the triple ⟨count, neededByTransaction, locksSuccessors⟩.

Figure 3.4: An example transaction and its UnlockRec

count for node 3 is 0, because T does not need to lock one of its successors, 6, and the other successor, 5, has node 2 as its onePredecessor. It is possible to come up with different values of counts for this transaction. For example, node 5 could have node 3 as its onePredecessor and then the count of both node 2 and 3 will be 1. Our current rule for making this decision is to choose the predecessor that appears first in the list of predecessors of a node.

The next field, *neededByTransaction*, is defined as follows:

Definition 3 *For a transaction T and node A, neededByTransaction(T,A) is true if A belongs to requestList(T), otherwise it is false.*

For example, in Figure 3.4, T_1 must lock node 3 even though it does not need it. Therefore, neededByTransaction(T_1, 3) = **false**, but neededByTransaction(T_1, 2) = **true**.

The field *lockType* denotes the type of lock that will be requested by the transaction TID for the nodeNo of the LockRec of this UnlockRec. The value of lockType in the UnlockRec is not necessarily the same as the current value of the *lockType* stored in the LockRec.

The next field *locksSuccessors* can be defined as:

Definition 4 *For each transaction T, and each node A accessed by it, locksSuccessors(T,A)=**false** if A is a leaf node or none of the successors of A will ever be locked by T. Otherwise, locksSuccessors(T,A)=**true**.*

For example, in Figure 3.4, for transaction T_1, the value of *locksSuccessors* for node 3 is **true**and for node 5 is false.

The last field of UnlockRec, *unlockable*, is defined as follows:

Definition 5 *A node A that is locked by a transaction T is unlockable, or unlockable(A,T)=***true***, iff one of the following holds:*

- *T locks A in exclusive mode, or*
- *T locks A in shared mode, and for all successors B, unlockable(B)=***true***, or*
- *T locks A in shared mode, and locksSuccessors(A)=***false***.*

This flag supports efficient checking of the condition L3b of the DDG policy. (We consider this in more detail in Section 3.4.4 when we consider the flow of control in unlocking procedure.)

3.4.3 Preprocessing of a Transaction

The processing described in this section corresponds to Box 3 in Figure 3.1. As soon as a transaction becomes active, it sends a message to the concurrency control manager (CCM) requesting it to initiate the transaction. In response to this, the CCM executes a procedure, called *ComputeCounts*, which preprocesses the transaction. In this section, we describe this procedure, which is shown in Figure 3.5.

As noted earlier, when a Class 1 transaction begins execution, it specifies the node from which it will start the traversal and the number of levels of the graph it plans to traverse. Based on this information, we estimate a set of nodes, setOfNodes, that it is ever going to access. In general, the set of nodes that a transaction will access cannot be predicted with certainty. For a traversal transaction, we assume that the set of descendants of the starting node contains all the nodes that will be accessed by the transaction. The transaction may actually access a subset of the descendants or the set of descendants may change. For example, in Figure 3.4, if the edge $(2, 5)$ is deleted, the transaction T_1 will never access node 5. These situations are taken into account by the incremental version of the *ComputeCounts* procedure which updates the setOfNodes every time there is an insertion or deletion in the graph.

When a Class 2 transaction begins execution, it specifies a *tentative* sequence of operations with the freedom to change this set at a later time. In fact, some of the operations that it specifies at start may not be valid in the state of the knowledge base in which they are executed. For example, in the knowledge base of Figure 3.4, suppose, a transaction wants to insert an edge $(5, 6)$. By the time it completes the insertion some other transaction may delete node 6 making the edge insertion invalid. Such operations are ignored as *stale requests*. On the other hand, if a transaction decides to add a new operation, then its successful completion will depend on what nodes it currently locks and whether they are enough for it to be able to satisfy the locking rules for locking the rest of the nodes that it needs. If the transaction is not able to finish, then it must abort and restart with a new value of the dominator.

The first step in the *ComputeCounts* procedure is to determine the dominator of the set setOfNodes (Line 2). Using the dominator tree that was generated when the knowledge base was compiled, and maintained using the incremental algorithms discussed in Section 3.3.2, the dominator of the set of nodes in the transaction can be computed in time linear in the cardinality of the setOfNodes. This is achieved by computing the nearest common ancestor of pairs of nodes (Schieber and Vishkin 1988) until we find a node that is the nearest common ancestor in the dominator tree of the nodes in setOfNodes. In our implementation, we used a simpler algorithm: start with the first node A in setOfNodes as the dominator, and check to see if it dominates every other node in setOfNodes; If yes, we are done, and if not, we repeat the same process with the predecessor of A. This approach requires time bounded by the product of the cardinality of setOfNodes and the maximum number of levels in the knowledge base.

The second step is to process all the nodes that are descendants of the dominator D and ancestors of some node $A \in$ *setOfNodes* (Lines 3-13). These are all the nodes that will be locked by the transaction provided the changes to the graph do not force a

```
1  funct ComputeCounts(txnInfo, setOfNodes) ≡
2        Compute the dominator D of setOfNodes
3        foreach node in setOfNodes do
4           Create its LockRec and UnlockRec
5           foreach ancestor nextNode of node that is a descendant of D do
6              foreach predecessor of nextNode do
7                 Create its LockRec and UnlockRec
8                 Set locksSuccessors to true
9                 if it is the first predecessor then
10                   Increment its count
11                fi
12             od;
13          od;
14       od;
15       Insert D into the pendingRequestList
16       Invoke RequestLock
17 end
```

Figure 3.5: Preprocessing of a transaction

change. Recall that this follows from Lemma 1(b). The inner **foreach** loop (Lines 5-13 in Figure 3.5) accomplishes this task. As we process each node, we create its LockRec and a corresponding UnlockRec if none already exists. While creating the UnlockRec, we also set the neededByTransaction flag.

Next, we compute the values of two fields in the UnlockRec. This is done in the innermost **foreach** loop (Lines 6-12). We increment the value of *count* for the first predecessor of each node and set the *locksSuccessors* flag to **true** (see Definition 4). Finally, we request a lock on the dominator (Lines 14-19). The lock mode is determined by the procedure *RequestLock* that is described in the next section.

As an example, consider the transaction T_1 as shown in Figure 3.4. When this transaction begins, we compute the dominator as node 1. Then we process the nodes 1,2,3,4 and 5 and create the corresponding LockRec and UnlockRecs. In Figure 3.4, we label each node that will be locked by T_1 by a triple that shows the count, neededByTransaction and locksSuccessors flags for T_1. After this processing, T_1 requests a lock on node 1 by a call to the procedure *RequestLock*, which is described in the next section.

3.4.4 Execution of a Transaction

The procedures described in this section correspond to Box 4 as shown in Figure 3.1. As a transaction executes, it requests the CCM for a lock on every item before accessing it (locking rule L1). The CCM processes such requests to enforce the rules of the DDG policy.

There are two main procedures in the CCM — locking procedures and unlocking procedures. Each call to the CCM may lead to the acquisition and release of several locks. It might be necessary to lock more than one data item to enforce the rules of the DDG policy. Such nodes are automatically identified by the CCM and locks set locks on them on behalf of a transaction. In every call to the CCM, a transaction also specifies any data items that it has accessed but which it no longer needs. This may lead to release of some locks. Some locks may be released on nodes which were locked to satisfy the rules of the policy and are no longer needed. If a lock cannot be immediately granted, a transaction may be made to wait. Since waiting can lead to deadlocks, we check for deadlocks every time a transaction blocks. In this section, we explain the procedure for locking, for checking deadlocks and for unlocking.

Flow of Control in the Locking Procedures

We first describe the locking procedure, *RequestLock*, which is shown in Figure 3.6. We accumulate the set of nodes that will be locked in the current call to the CCM in the set *pendingRequestList* which is maintained in the transaction table. Initially, this set contains only the node for which the transaction requests a lock. The control in the locking procedure loops over all the nodes in this list. For each node, the loop begins by checking if the transaction has already locked that node (Line 3). In the next step, we check if the node is currently deleted (Line 5). If the node is deleted and the current request is not for an insertion then CCM treats the request as stale. In our current implementation, the transaction continues execution from the next pending request.

Next, we check the locking rules (Lines 10-13). If the node is a dominator for the transaction or the transaction inserts this node, no pre-conditions are required. Otherwise, we need to make sure that the transaction has locked all of the node's predecessors in the past and is currently holding a lock on at least one predecessor (locking rule L5a). This also applies to the nodes which are not SCCs by themselves, because in such cases, we consider corresponding condensed node. If the transaction has not locked all the predecessors, then we insert the predecessors that have not been locked into the pendingRequestList. If all parents have been locked in the past, we check to see if at least one of them is currently locked. If a transaction does not hold a lock on at least one predecessor, then it must abort, and an abort signal is issued (it is not shown in Figure 3.6).

The rule L5b which requires that "for every path $A_1, \ldots, A_p, A$ in the present state of the underlying undirected graph of $\overline{G}$, such that A_1 is locked (in any mode), and $A_2, \ldots, A_p$ are locked in shared mode, T has not unlocked any of $A_1, \ldots, A_p$" is enforced while releasing locks, and will be explained later in this section. After verifying that the locking conditions are met, a lock is requested (Lines 14-19). If the transaction will be reading the node or the lock is being requested due to the rules of the locking policy, the locking mode is shared; otherwise, it is exclusive. If the transaction successfully acquires the lock, we adjust the counts in the UnlockRec of this node and see if any lock release is possible (Lines 20-24). Otherwise, the transaction is blocked, and we check for deadlocks (Line 28-30). This process is repeated until the pendingRequestList becomes empty.

As an example, consider transaction T_1 shown in Figure 3.4. The first call of T_1 to the CCM will be to lock node 1 in shared mode. Since node 1 is also a dominator, T_1 has already locked it during the preprocessing. In this case, lock is granted without any further checks. Upon receiving this message, the transaction proceeds with the processing of node 1, and if no further processing of this node is required, a signal indicating this is sent to the CCM, by piggy-backing it with the next call. In its second call to the CCM, T_1 requests an exclusive lock on node 2. Since this call also indicates that node 1 is no longer required, the CCM calls the *Unlock* procedure which sets neededByTransaction(T_1, 1) to **false**. The lock on node 1 cannot be released at this moment, because its count is greater than 0 (We consider this in more detail in the following paragraph). Next, we check the locking conditions for node 2, and since they are satisfied, we request a lock. If the transaction successfully acquires the lock, we decrement the count of node 1. Since the count of node 1 is now 1, we

```
funct RequestLock(txnInfo) ≡
    foreach nodeNo in the pendingRequestList do
      if the nodeNo is already locked by the transaction then
        Remove nodeNo from the pendingRequestList
      elsif nodeNo does not exist in the knowledge base and
            the transaction is not inserting it then
            if the node is locked on transaction's request then
              Send a message indicating that the request is stale
            fi
      elsif (nodeNo is a dominator for the transaction) or
            (transaction is inserting nodeNo) or
            (all parents of nodeNo have been previously locked
            and one parent of nodeNo is presently locked) then
             Determine the lockType for current request
            if lockType = READ then
              Request a read lock
            else
                Request a write lock
            fi
            if lock is acquired then
              Decrement the count of the onePredecessor of nodeNo
              if (lockType is WRITE) or
                 (the transaction does not lock any successors of nodeNo)
                 then Check for release of lock on nodeNo
              fi
              Remove nodeNo from the pendingRequestList
            else
                Put the transaction in the blocked queue
                Check for deadlocks and resolve, if any
                exit
            fi
      else
           exit
      fi
    od
end
```

Figure 3.6: Procedure to acquire locks

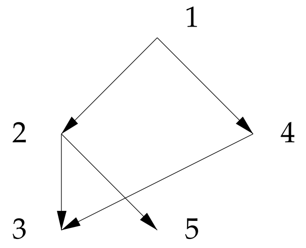


Figure 3.7: A deadlock situation for the DDG-SX policy

still cannot release its lock. The CCM repeats this process of checking the conditions and requesting locks in the next call when T_1 requests a lock on node 5. T_1 has not yet locked node 3, and therefore, the locking conditions for node 5 are not satisfied. The procedure *AllParentsLocked* inserts node 3 in the *pendingRequestList* of T_1. Once T_1 acquires a shared lock on node 3, the locking conditions for node 5 are checked and a similar process is repeated.

Checking for Deadlocks

Every time a transaction blocks, we check for deadlocks. This is done by checking for cycles in the waits-for graph (Gray and Reuter 1993) which is maintained using the fields waitsFor and lockers in TxnInfo and LockRec respectively. When a transaction T is unable to acquire a lock on node A, T is the first transaction in a potential cycle in the waits-for graph. For each transaction T_i in the field lockers of the LockRec for node A, we check the waitsFor field of T_i to see if it is waiting to acquire some lock. If T_i is waiting for lock on some node B, we repeat the same process for all the transaction that hold lock on B. This process continues until a cycle is found, or we are able to conclude that there are no cycles. The deadlock is resolved by aborting the youngest transaction.

In our experimentation we found that the DDG-SX policy was more susceptible to deadlocks as compared to 2PL. This is because when transactions access the data items in the same order (say breadth-first or depth-first), 2PL gets the deadlock avoidance for free. With the DDG-SX policy, however, some times a transaction acquires more locks than it actually needs, due to which all transactions may not acquire locks in the same order, thereby possibly leading to deadlocks. As an example, consider the knowledge base shown in Figure 3.7. Suppose a transaction T_1 wants to access nodes 1, 2, 5 and 4 and another transaction T_2 wants to access nodes 1, 2, 3, 4 and 5. Furthermore, let us assume that T_1 and T_2 wish to access these nodes in a depth-first fashion. Then running under the DDG-SX policy, T_1 will acquire locks in the order 1, 2, 5 and 4. T_2 will acquire locks in the order 1, 2, 4, 3 and 5, because to lock node 3, it must first lock node 4 (rule L5a). Thus, we can see that the order of access is violated for nodes 4 and 5 giving a possibility of deadlock. In contrast, if the transactions run under 2PL, T_1 acquires locks in the order 1, 2, 5, 4 and T_2 acquires locks in the order 1, 2, 3, 5 and 4, thus maintaining the same order on the entities that are common to the two.

Flow of Control in the Unlocking Procedure

In order for T to release lock on a node A before reaching locked point, CCM has to check the following conditions:

U1 For every path $A, A_1, \ldots, A_p, B$ in the present state of the underlying undirected graph, such that A is locked (in any mode), $A_1, \ldots, A_p$ are locked in shared mode, T intends to lock B in future, T must not unlock any of $A, A_1, \ldots, A_p$" (by locking rule L5b).

U2 The node A is no longer needed by T (by rule L3).

U3 Releasing the lock on node A does not prevent the locking of any of its successors at a later stage in the execution of the transaction (as required by rule L5a).

The CCM releases locks before locked point of a transaction using a procedure called *CheckRelease* which is shown in Figure 3.8 and checks for conditions U1-U3.

Condition U1 requires checking of paths in the underlying undirected graph which can be potentially expensive. We try to minimize this checking by using the locksSuccessors and unlockable flags defined earlier. For convenience, we say that for a transaction T, there is an S-path from node A to B, if T locks A (in any mode), locks B or expects to lock B (in any mode), and T currently has a shared lock on each entity (other than B and A) on a path between B and A in the underlying undirected graph. Thus, an alternative way to state U1 is that T can release lock on A only if there is no S-path from A to B such that B is a node that T plans to lock in future. To make this checking efficient, we make use of the following observations.

The first observation is that the CCM needs to check for S-paths only after T acquires an exclusive lock on A or it acquires a shared lock on A, but does not intend to lock any successor of A (that is, locksSuccessors=**false**). This is because, if a transaction locks A in shared mode but intends to locks some successor of A in future, then the condition U1 is trivially false.

The second observation is that we can use the unlockable flag and avoid checking some of the paths for which U1 is false. This is because the unlockable flag of a node A remains false until U1 is not satisfied for at least one directed path from A to some descendant of A. Thus, if unlockable flag of a node is false, without further checks we can conclude that U1 is not satisfied for at least one S-path passing through A.

Finally, a transaction T can release an exclusive lock on a node A only if all its *relevant neighbors* are in the same S-path. The relevant neighbors of a node A include all its predecessors (provided A is not a dominator for T) and successors (provided they are/or will be locked). This is because if a node is locked in exclusive mode, it can be a part of more than one S-paths.

Let us consider in detail how this is implemented in the procedure *CheckRelease*. *CheckRelease* is invoked either after granting a lock or upon receiving a message from the transaction that a transaction no longer needs a locked node. We accumulate the nodes on which the locks could possibly be released in a set called *searchQ*. If *CheckRelease* is called after a lock on a node A is acquired, searchQ initially contains all the predecessors of A. If it is called in response to a transaction's declaration that it no longer needs a node A, then this set only contains the node A (Line 2).

The inner **while** loop (7–24) checks the S-paths passing through each node A in searchQ. If there is an S-path from A to some node B, T intends to lock B, and B has not been locked by the transaction so far, then lock on A cannot be released and the loop exits (Line 9). If node B has been locked in the past, then we proceed to check its unlockable flag.

If a node A is locked in an exclusive mode, then its unlockable flag is **true**. But we have to make sure that it is not part of any other S-path (Recall the observation 3 from above). We check this using the procedure *CheckPathToWLock*. To do this we check all its relevant

```
funct CheckRelease(txnInfo, nodeNo) ≡
    Initialize the searchQ to contain the nodes on which
        lock could be released
    foreach nodeNo in searchQ do
      Add nodeNo to nodeList that contains the nodes that
          may potentially lie on an S-Path passing through nodeNo
      while there is an unprocessed node in nodeList and
            unlocking condition U1 is not violated do
            if the node is not locked by the transaction then
              exit
            else
                if node is locked in write mode then
                  CheckPathToWLock(node, txnInfo, sPath)
                elsif node is locked in read mode then
                    CheckPathToRLock(node, txnInfo, sPath)
                fi
                if locksSuccessors is true then
                  Add all successors of node to nodeList
                fi
                if (node is not the dominator for the transaction) then
                  Add all predecessors of node to nodeList
                fi
            fi
      od
      if (the unlocking conditions U1 is not violated) then
        foreach node in sPath do
          if the value of the count is 0 and
            the node is not needed by the transaction then
            Invoke ReleaseLock(node)
          fi
        od
      fi
    od
end
```

Figure 3.8: Procedure to release the locks before commit time

```
funct CheckPathToWLock(node, txnInfo, sPath) ≡
      onPath = true
      if node is not the dominator for the transaction then
         Check each of its predecessors to see if
              it is locked in exclusive mode or
              if it is locked in the shared mode, it also lies on sPath
         otherwise set the onPath to false
      fi
      if (onPath) and
         (the transaction locks some successors of the node) then
         Check all successors as above
      fi
      if onPath then
         add the node to the sPath
      fi
end
```

Figure 3.9: Procedure to check nodes locked in exclusive mode

```
funct CheckPathToRLock(node, txnInfo, sPath) ≡
      if unlockable flag for the node is true then
         Insert the node to the sPath
      elsif the unlockable of all the successors is true then
             Set the unlockable flag of node to true
             Add the node to sPath
      else
            U1 is not satisfied
      fi
end
```

Figure 3.10: Procedure to check nodes locked in shared mode

neighbors (Lines 2-15 in Figure 3.9). The node A is a part of the S-path that is currently being checked, if the following conditions are satisfied: (1) All the relevant neighbors have been locked in the past, and (2) The relevant neighbors that are currently locked are locked in the exclusive mode or are locked in shared mode and are part of the same S-path that is being currently checked. The outcome of this check does not influence the unlocking of other nodes on this S-path.

If the node A has been locked in shared mode, we check its unlockable flag. We look at the unlockable flag of the successors of A (Lines 2-4 in Figure 3.10) to enforce Definition 5. If the unlockable flag of all the successors of A is found to be true, the unlockable flag of A is set to be true and is added to the S-path. If the unlockable flag of even one of the successors is false, the search stops, and we stop looking for an S-path passing through A.

If the unlockable flag for a node A is found to be **true**, we need to find other nodes that might lie on the S-path. Such nodes are all the relevant neighbors of A as defined earlier (Lines 17-22). We continue this process until we find at least one node that cannot be unlocked or we determine that no such node exists.

If unlockable flags of all the nodes on an S-path are found to be **true**, we proceed to check the conditions U2 and U3 (Line 27). Condition U2 is checked by looking at the neededByTransaction flag. To satisfy U3, the count of that node must be zero. If these conditions are true, the lock on that node is released.

As an example, let us continue to follow the execution of transaction T_1 of Figure 3.4. As T_1 acquires a lock on node 3, we decrement the *count* of node 1, which has now a value of 0. In the subsequent call to *CheckRelease*, we check all S-paths passing through node 1. The only S-path that passes through node 1 contains nodes 2, 1, 3 and 5. Since node 5, which is one of the successors of node 3, has not been locked yet, unlockable($T_1, 3$) = **false**, and therefore, we cannot release locks on any of these nodes. Once T_1 locks node 5, unlockable($T_1, 5$) is **true**; which makes unlockable($T_1, 3$) = **true** and also unlockable($T_1, 1$) = **true**. Furthermore, since node 2 is locked in exclusive mode, unlockable($T_1, 2$) = **true**. Thus, we can call *ReleaseNodes* for all nodes on this S-path. In this call to *ReleaseNodes*, we can release locks only on nodes 1 and 3 because at this moment neededByTransaction($T_1, 5$)=**true** and $count(T_1, 2) = 1$.

3.4.5 Commit Time Processing

This processing corresponds to Box 5 in Figure 3.1 and needs to be executed only when the committing transaction has inserted or deleted any entities. This processing is implemented in the procedure *UpdateCounts* shown in Figure 3.11. One of the inputs to this procedure is the set *affectedNodes* which contains the nodes affected by an edge insertion or deletion. Specifically, for an edge insertion (A, B), this set contains the following:

- node B, if the edge insertion does not cause any change in the SCCs; otherwise,
- the node in the condensed graph that corresponds to the newly created SCC G_{ij} and all the nodes that enter G_{ij} as a result of the edge insertion.

For an edge deletion (A, B), this set contains the following:

- if the edge deletion does not cause any change in the strongly connected components, then affectedNodes is empty; otherwise,

```
1  funct UpdateCounts(txnInfo, affectedNodes) ≡
2        foreach node inserted by the transaction do
3           Increment the count of its first predecessor
4        od
5        foreach node affected by an edge insertion do
6           foreach transaction that intends to access the node and
7              node has not been locked by the transaction do
8              Recompute the counts of all ancestors of node that are descendants
9              of a node which was not processed in the preprocessing
10          od
11       od
12       foreach edge (source, target) deleted by the transaction do
13          if the source was the first parent of the target then
14             Assign another parent as the first predecessor and
15             increment its count
16          fi
17       od
18 end
```

Figure 3.11: Procedure to incrementally maintain the counts

- it contains the nodes that leave the SCC and any new condensed nodes created as a result of it.

The first step in this procedure is to create a onePredecessor field in the LockRec for each inserted node A (Lines 2-4). The Class 1 (traversal) transactions that access the parent B of A may now also access A. Since this information was not available at the time of preprocessing the transactions, we locate all such transactions T_i and increment the $count(T_i, B)$. For example, consider a situation in which a transaction adds a node 7 and an edge (4,7) in the knowledge base of Figure 3.4. We will first define onePredecessor(7)=4. Then, by looking at the UnlockTable of node 4, we find all the Class 1 transactions and in their UnlockRecs, we increment the count of node 4 by 1.

The second step in this procedure is to take care of the changes caused by an edge insertion (Lines 5-11). The new SCCs are treated as insertions into the graph. An edge insertion may create some new paths in the knowledge base that will now be accessed. For all nodes on such paths, we generate the UnlockRecs. Furthermore, it is possible that for some transactions the dominator has changed in such a way that they will now be unable to complete execution under the rules of the policy. For such transactions, we issue an abort signal. For example, in Figure 3.4, suppose, we add an edge (6,4) after a transaction whose dominator is node 2 (with access set containing nodes 2,4 and 5) has finished its preprocessing. Such a transaction aborts because the dominator is no longer node 2, and the transaction will never be able to satisfy the locking rules for node 4. On the other hand, consider a transaction that accesses nodes 1, 2 and 4. In its preprocessing phase, it processes nodes 1, 2 and 4. But after the edge (6,4) is added, it will also need to lock nodes 6 and 3. Therefore, we process nodes 6 and 3 and create the Unlock records for them.

The last step is to update the counts to reflect the effect of edge deletions (Lines 12-17). The deletion of edge (A, B) may cause some paths to disappear from the graph and may affect the onePredecessor of some of the nodes. For the sake of simplicity, we do not deal with the situation when some of the paths in the graph may disappear. Due to this, some of the nodes will be held locked longer than they need to be. Next, we consider the nodes for which the onePredecessor is deleted. If B was the onePredecessor for the node A, we assign the predecessor C, which is the first in the current list of predecessors of A, as the new onePredecessor. We then adjust the counts of all transactions that will access some successor of B. To illustrate this process, suppose in Figure 3.4 we delete the edge (3,5). Now, T_1 will no longer have to lock node 3. We ignore such changes. The count of node 1 will never be decremented to 0, and therefore, if it has already been locked, it will be unlocked before commit time. If we delete the edge (2,5), then node 5 will lose its onePredecessor. In this case, we decrement the count of node 2, assign node 3 as the onePredecessor of node 5 and increment the count of node 3.

3.5 An Alternative Implementation to Minimize the Preprocessing of a Transaction

In the implementation presented so far, we preprocess a transaction and compute the values of the fields in the UnlockRecs for all the entities to be accessed by it at run time. Such an implementation can be considered as a *dynamic* implementation. Since Class 1 transactions can be very long, this preprocessing may lead to several disk accesses causing a performance degradation. But on the other hand, if these items are retained in the buffer until the transaction has accessed them, the disk accesses at the time of preprocessing are not an extra overhead. In this section, we present an alternative approach which eliminates the need for preprocessing but may require locking more items than necessary or for a longer time than necessary. The key difference is that we compute the necessary information when the knowledge base is compiled by making some assumptions about the transactions. We call this implementation a *static* implementation.

As seen in Section 3.4.3, in the preprocessing phase of a transaction, we compute the following information: dominator, count, neededByTransaction and locksSuccessors. Instead of computing these values when a transaction begins execution, we compute them while compiling the knowledge base.

We compute the dominator of a node A with the assumption that if a transaction begins its traversal from A, then all its descendants will be accessed by it. In other words, with each node A, we associate a node B, such that B dominates A and also dominates all the descendants of A. The disadvantage of this approach is that if the transaction does not access all the descendants, we may end up locking more nodes than are necessary.

Similarly, we compute the value of the count with the assumption that the transaction will access all the descendants. When a transaction creates its UnlockRec, we copy this value of the count into its UnlockRec and update this private copy as the transaction progresses. The disadvantage of this approach is that if none of the successors of a node is accessed by the transaction, it will still have a non-zero value of the count, and thus, it will be held locked longer than necessary. We can overcome this problem if we require the transaction to send a message to the CCM whenever it decides that it does not plan to access the descendants of a node. On receiving this message, the CCM will set the count to zero, and also set the locksSuccessors flag to **false**. The locksSuccessors flag for other nodes will always be initialized to **true**.

We compute the neededByTransaction flag based on the observation that a transaction accesses only those nodes that are dominated by the first node on which requests a lock. (Recall that dominator of a transaction and the first node on which requests a lock are not necessarily the same.) For such nodes, we set the neededByTransaction to **true**. Rest of the nodes are locked to satisfy the rules of the locking policy, and therefore, for them, we set the neededByTransaction flag to **false**.

Needless to say, the dominator and the count values generated above will need to be maintained as the graph is updated. The incremental methods described earlier will still be applicable to these situations. Specifically, for computing the dominator nodes, the incremental dominator tree algorithm in conjunction with incremental nearest common ancestor algorithm can be used. To compute the new values of counts, we can use a modified version of *UpdateCounts* in which, in addition to updating the values for all active transactions, we also update the values of the counts that were computed at compile time.

Thus, using the above approach, we can eliminate the preprocessing phase of a transaction and avoid the costly disk IO operations incurred in it. This, of course, is achieved at the cost of inefficiency in the locking process. The choice between the static and dynamic implementations depends on this tradeoff. We will analyze this tradeoff quantitatively in Chapter 5.

3.6 Chapter Summary

In this chapter, we described the implementation of the DDG policy. To the best of our knowledge, the problems that arise while implementing the DAG policy and its variants have not been documented elsewhere. To implement the DDG policy, two types of problems had to be addressed. First, we needed to keep track of the properties of the underlying graph, and second, we needed a mechanism to release the locks of a transaction before its locked point.

The properties of the graph were computed at compile-time and were updated at run-time using incremental algorithms. The graph properties that were necessary included information on strongly connected components and dominators. There was no existing algorithm to incrementally compute the strongly connected components, and therefore, we sketched an algorithm for this purpose.

A scheme to release locks before the locked point was presented which roughly worked as follows. At the start of the execution of a transaction, we performed preprocessing to generate information about the entities it may potentially access. The locks necessary to enforce the rules of the DDG policy were automatically identified by the CCM. We required a transaction to send a message to the CCM once it had finished processing an entity. The CCM also checked which locks could be released and released them when possible. The checking for release of locks utilized three pieces of information: *count* – which indicated whether the transaction has locked all the successors that it will ever need, *neededByTransaction* – which indicated whether the transaction has finished processing that item, and finally, *locksSuccessors* and *unlockable* flags which helped in enforcing the locking rule L5b. This scheme is more general than lock-coupling which is often used in the implementation of non-two-phase locking policies.

If a committing transaction inserted or deleted items from the knowledge base, it triggered an update of the knowledge base properties that were generated at compile time. It also updated the preprocessed information of any active transactions that may be affected due to this change.

The results presented in this chapter are a good starting point for someone who is interested in building a system using the DDG policy. In the next chapter we describe the simulation model in which we implemented the algorithms described in this chapter, and later, report the experimental results obtained using the simulation.

Chapter 4

Experimental Setup

In this chapter, we describe the experimental set up that we use in evaluating the DDG policy. In particular, we describe the simulation model, experimental methodology, validation of the model, measurement results, the choice of load control strategy and alternative approaches for the evaluation of concurrency control algorithms.

4.1 Simulation Model

A concurrency control algorithm for a multi-user knowledge base is a sub-component of a knowledge base management system (KBMS) (Mylopoulos et al. 1992a). We model the concurrency control algorithm in detail while the rest of the system is abstracted into a single component, called the transaction manager. In total, we assume that the KBMS has four components: a *source*, which generates transactions, a *transaction manager*, which models the execution behavior of transactions, a *concurrency control manager*, which implements the details of a particular algorithm, and a *resource manager*, which models the CPU and I/O resources of the system. Figure 4.1 illustrates these components and their interconnections. Our model is based on an earlier model which is often used in performance studies of concurrency control algorithms (Agrawal, Carey and Livny 1987). The parameters of the model are shown in Table 4.1. In this section, we describe components of the model in detail and then point out the differences from previous ones (Agrawal, Carey and Livny 1987).

4.1.1 Source

The *source* is the component responsible for generating the workload for the knowledge base management system. Our model has the flexibility to represent the workload by either a closed or an open model (Lazowska et al. 1984). In a closed model, the user requests are generated from a fixed number of terminals. Each user cycles through a period of thinking at the terminal and a period of getting service. Thus, arrival rate depends on the number of users who are currently being served. We parameterize a closed model with *NumTerminals* terminals each having think times selected from an exponential distribution with mean *ThinkTime*. In an open model, the user requests are generated from an infinite population. Once a user completes service, it permanently leaves the system. Thus, the arrival rate does not depend on the number of users already in the system. We parameterize such a model by the parameter *ArrivalRate*, which specifies the average number of transactions arriving

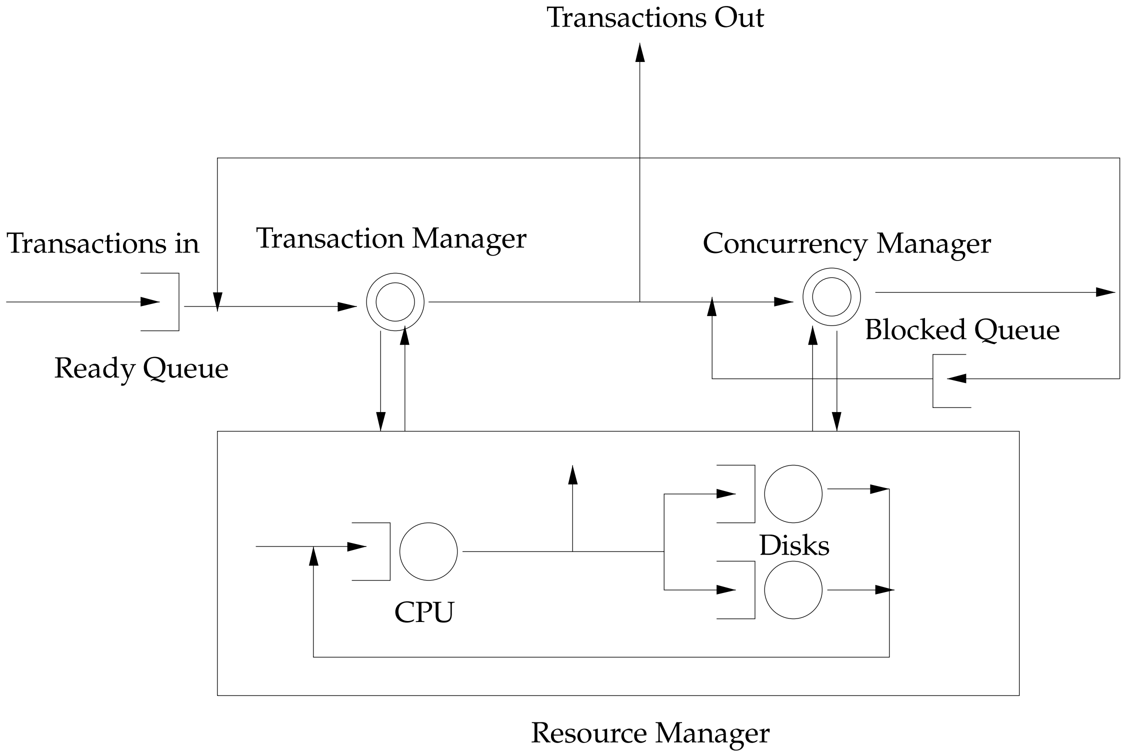


Figure 4.1: Performance model of the KBMS

per second. A Poisson arrival process is assumed. The parameter *AccessPattern* defines the traversal pattern of Class 1 transactions, which can be either breadth-first or depth-first. *RateUpdateStr* is the percentage of Class 2 transactions that perform insertions/deletions in the graph. The parameter *UnitCPU*, specifies the average amount of CPU time required for processing an entity when reading or writing it. The processing times are exponentially distributed.

4.1.2 Transaction Manager

The *transaction manager* (TM) is responsible for accepting transactions from the source and for modeling their execution starting from transaction initiation to transaction commit or abort. An important function of the transaction manager is to control the multiprogramming level (MPL) of active transactions. To do this, it maintains a ready queue. When a transaction arrives, the transaction manager checks the current multiprogramming level in the system. If it is already at its limit, the incoming transaction is made to wait in the ready queue. Once some transaction finishes execution, the transaction at the beginning of the queue is admitted into the system and begins execution.

4.1.3 Resource Manager

The *resource manager* can be viewed as a model of the operating system and its resources. The resource manager provides CPU and I/O services. We assume that the resources consist of one CPU and two disks. The CPU service discipline is processor sharing (Lazowska et al. 1984). Each of the disks has its own queue, which it serves in a FIFO manner. The disk required to serve a new request is chosen randomly, with all disks being equally probable. Thus, our I/O model assumes that files are placed on the disks in a way that access requests across the disks are balanced. Disk accesses require an average time equal to *diskIOTime*. The parameter *HitRate* specifies the probability of a buffer pool hit. If *Hitrate* = 0, a read or write request always leads to disk I/O. However, if *HitRate* > 0, then the disk I/O associated with a read or write request is skipped with probability *HitRate*.

4.1.4 Concurrency Control Manager

The *concurrency control manager* (CCM) implements a given concurrency control algorithm, and in our experiments it is the only module that will be changed to simulate the performance of different concurrency control algorithms. As illustrated in Figure 4.1, it is responsible for handling concurrency control requests made by the transaction manager, including read and write access requests, and requests to commit or abort a transaction. The algorithm maintains a blocked queue for the transactions for which the requested lock cannot be immediately granted. It incorporates a deadlock detection algorithm based on wait-for graphs. Even though it has a provision for lock upgrades[1], in all our experiments, we assume that there are no lock upgrades.

4.1.5 Differences from Previous Models

There are three important differences between our model and the one used in previous studies (Agrawal, Carey and Livny 1987).

First, our workload consists of two classes of transactions. Our model has the flexibility to represent each class by either an open model or a closed model.

Second, we have used a detailed model for capturing the cost of performing various operations in a concurrency control algorithm. Our cost model has four parameters (see last four entries in Table 4.1): *InitCost*, cost of initiating a transaction. *LockCost* cost of a call to the lock manager, *CommitCost* cost of committing a transaction, and finally, *AuxCost* which is the run-time cost of maintaining the properties of the knowledge base that are used in the implementation of the DDG policy. We describe these in more detail in a subsequent section.

Finally, we have defined the length of Class 1 transactions as the number of levels in the graph that a transaction traverse, where level number is defined as follows. If A is the first node locked by a transaction T, and B is some other node locked by T, then the level number of B is the length of the shortest path from A to B plus one. This is in contrast to a more traditional definition of transaction length where it is defined as the number of data items that a transaction accesses. Our choice is motivated by the characteristics of the knowledge base applications in which Class 1 transactions involve a traversal of the knowledge base.

4.2 Experimental Methodology

We implemented 2PL and the DDG policies in the concurrency control manager of our simulation model. (The implementation of the DDG policy was described in detail in the

[1]In a lock upgrade scheme, all locks are initially acquired in shared mode. At the time of write, the locks on items to be written are upgraded to exclusive mode (Gray and Reuter 1993).

Parameter	Meaning
Model Parameters:	
NumTerminals	Number of Terminals – for Class 1
ThinkTime	Think Time for the terminals – for Class 1
ArrivalRate	Arrival Rate of transactions – for Class 2
UnitCPU	Average CPU time for processing an entity
diskIOTime	Time required to perform one disk IO
HitRate	Buffer pool hit probability
TxnLength	Transaction Length
WriteProb	Update probability of a transaction
AccessPattern	Traversal pattern of Class 1 transactions
RateUpdateStr	Rate of insertions and deletions into the graph
Cost Parameters:	
InitCost	Cost of initiating a transaction
LockCost	Cost of acquiring a lock
CommitCost	Cost of committing a transaction
AuxCost	Cost of maintaining auxiliary information

Table 4.1: Simulation model parameters

previous chapter.) The implementation was done in the DeNet simulation environment (Livny 1990) on a DECStation 5000 (model 132).

The choice of DeNet was motivated by the following reasons. DeNet provides a modular programming environment which reduces the development effort. It has excellent facilities for parameterizing a simulation model, collecting statistics, generating random numbers and for creating reports which delegate these mundane tasks to the underlying programming system. It also has a graphical interface to display the measurement data which proved to be a useful debugging tool. The main drawback of DeNet is that the development has to be done in Modula-2 which is not as portable as the programming language C.

Our primary performance metrics are the response times of Class 1 and Class 2 transactions. In many cases, for the sake of clarity and ease of comparison, we report the *percentage improvement* between the mean class response times given by the two policies. If $R(j)_{2PL}$ and $R(j)_{DDG}$ are the mean response times of transactions in class j, then the percentage improvement of the DDG policy over 2PL is computed as $100 \times (R(j)_{2PL} - R(j)_{DDG})/R(j)_{2PL}$.

We employed a batch means method for the statistical data analysis of our results, and each simulation was run long enough to obtain a sufficiently tight confidence intervals (Law and Kelton 1991). The reported mean values are accurate within 5% at the 90% confidence level. A typical simulation run required at least 90 minutes of computing time in which we had about ten thousand completions of Class 1 and about a million completions of Class 2. More details about the analysis of the simulation output are given in Appendix A.

4.3 Validation of the Model

To verify the credibility of our implementation of the two-phase locking, we compared our results with a previous study (Agrawal, Carey and Livny 1987). We ran an experiment with the parameter values used in that study which are shown in Table 4.2. In addition,

Parameter	Meaning
Model Parameters:	
NumTerminals	200
ThinkTime	1s
ArrivalRate	Not Applicable
UnitCPU	0.015s
diskIOTime	0.035s
HitRate	0.0
TxnLength	4-12
WriteProb	0.2
AccessPattern	Random
RateUpdateStr	Not Applicable
Cost Parameters:	Not Applicable

Table 4.2: Parameters used for validation

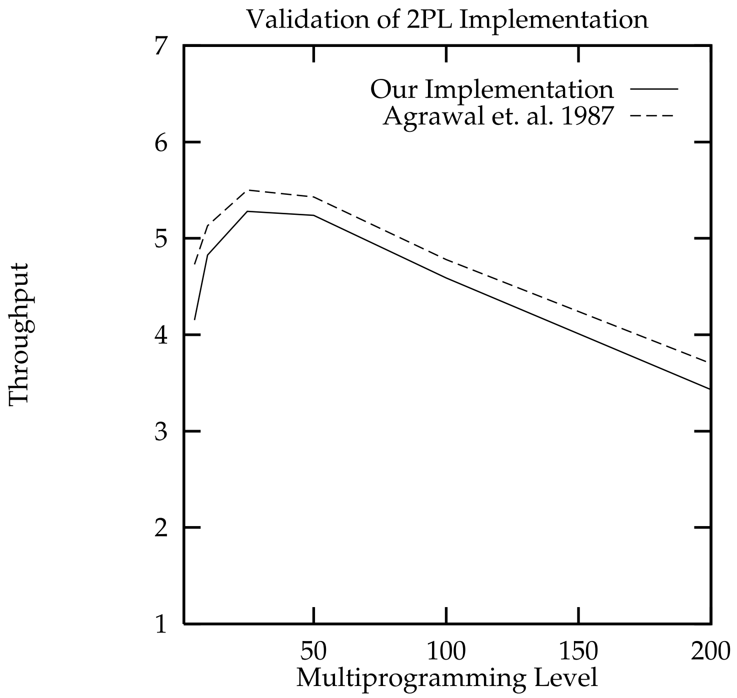


Figure 4.2: Validation of the model

we assumed a database size of 1000 pages and that no lock upgrades are allowed. Random access patterns indicate that any item in the database can be accessed with equal probability. The comparison of throughput for this case is shown in Figure 4.2. The overall shapes of the throughput curves are nearly identical in the two cases. Furthermore, our results were within 10% of the previous results (Agrawal, Carey and Livny 1987).

4.4 Measurement of Concurrency Control Overheads

To reflect the relative running costs of the DDG policy and 2PL in the simulation, we measured the cost of performing basic operations in concurrency control for both the policies. The measurements were done on the implementation embedded in the simulation model. We measured four parameters, the cost of starting a transaction (InitCost), the cost of acquiring a lock (LockCost), the cost of committing/aborting a transaction (CommitCost) and the cost of updating the structure (AuxCost).

For the DDG policy, the InitCost includes the cost of preprocessing the transactions and generating information that will be used for releasing locks before the locked point of a transaction. It also includes the cost of creating entries in the lock table and the transaction table. For 2PL, no preprocessing of the transactions is required, but entries in the transaction table and the lock table are created when the transaction begins.[2]

The LockCost includes the CPU time used between the time a lock request is made and the time it is finally granted. If the transaction gets blocked in between, information on the processing time consumed so far is maintained, and when the transaction gets unblocked later, this time is added to any further processing time. For the DDG policy, the cost of locking also includes the costs of checking for release of locks before the locked point, checking for deadlocks, releasing locks and unblocking of transactions that might happen in each call to the lock manager.

The CommitCost includes the processing required to release all the locks and to finally commit the transaction. In general, the cost of commit for 2PL is higher as compared to the cost of commit for the DDG policy. This is because, under the DDG policy, a transaction would have already released several of its locks prior to commit, whereas for 2PL policy, all the locks are released at commit time.

The AuxCost is the cost of incrementally computing the information on dominators and strongly connected components. Since 2PL does not need this information, this cost is incurred only by the DDG policy.

The measurements were done on a DECStation 5000 (model 132) running Ultrix 4.2 using the *getrusage* system call. We found that the values of concurrency control overheads were sensitive to the simulation parameters such as transaction length, write probability, etc. To take this into account, we established *baseline* values of the overheads by doing a preliminary simulation in which we did not run the simulation long enough to obtain sufficiently tight confidence interval on the transaction response time. We re-ran the simulation using these baseline values, this time to obtain the desired confidence intervals on response time, and at the end of the simulation, compared the actual values with the baseline values. If the difference between the actual values and the baseline values turned out to be large, we repeated the simulation with the new values. This approach had the

[2] It is possible to create these entries when they are actually needed. In that case, this cost will be included in the cost of locking. This distinction does not make any difference to the relative performance of the two algorithms.

Costs (μ seconds)	Baseline Mean	Baseline S.D.[3]	Actual Mean	Actual S.D.	% Error Mean	% Error S.D.
Two Phase Locking						
Class 1:						
InitCost	13000	32000	12925	30593	-0.59	-4.60
LockCost	330	1100	365	1144	9.57	3.83
CommitCost	12700	30000	12470	28899	-1.84	-3.81
Class 2:						
InitCost	300	1000	328	1084	8.48	7.74
LockCost	500	1400	494	1341	-1.19	-4.43
CommitCost	250	900	223	908	-11.96	0.83
DDG policy						
Class 1:						
InitCost	32000	80000	32643	79631	1.97	-0.46
LockCost	1800	2700	1762	2664	-2.17	-1.37
CommitCost	8000	19000	8010	19125	0.12	0.65
Class 2:						
InitCost	430	1200	476	1282	9.60	6.36
LockCost	400	1200	366	1150	-9.24	-4.33
CommitCost	250	1000	231.57	923.8	-7.96	-8.24
AuxCost	200	1000	155	765	-28.61	-30.78

Table 4.3: Concurrency control overheads

advantage that we were able to minimize the data collection efforts and the results were repeatable across different runs.

In Table 4.3, we show the baseline values used in a typical simulation run and also the actual values and the corresponding errors. These numbers should be interpreted in view of the following considerations. The resolution of the *getrusage* system call is 3906 micro seconds. As a result, several of our measurements were zero. The numbers shown in Table 4.3 are the values which were averaged over all zero and non-zero observations. Furthermore, we expect that these values can be substantially reduced if significant effort is invested in optimizing the implementation. Nonetheless, these numbers are useful to get a relative measure of the costs of the algorithms.

Since the concurrency control overheads have a high variance, we modeled the costs using a hyper-exponential distribution (Law and Kelton 1991). Let μ be the mean and σ be standard deviation of a cost parameter. We used the following expression to compute the parameters of the corresponding hyper-exponential distribution (Sauer and Chandy 1975):

$$f(t) = are^{-rt} + (1-a)se^{-st}$$

where: $a = (1/2)\left(1 - \sqrt{(\sigma^2-1)/(\sigma^2+1)}\right)$, $r = 2a/\mu$ and $s = 2(1-a)/\mu$. These parameters were then given as an input to the simulation model.

[3]Standard Deviation

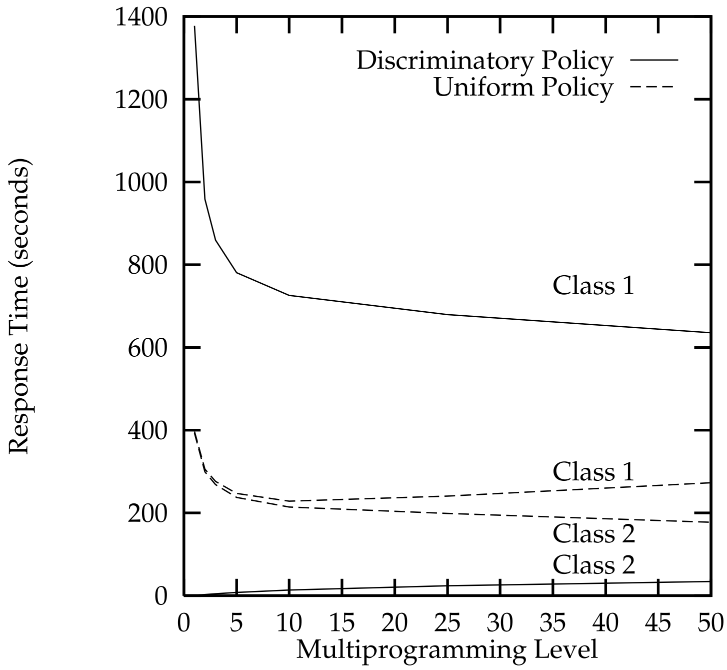


Figure 4.3: Effect of load control policy

4.5 Load Control Strategy

Using the *load control* strategy, the transaction manager controls the number of active transactions in the system at any given time. Permitting too many active transactions in the system may lead to thrashing due to data contention, while executing too few transactions does not exploit the full benefits of concurrency. In a single class workload, load control can be done by simply controlling the number of active transactions (Tay 1987; Carey, Krishnamurthi and Livny 1990). In a two class model, however, we can consider two options for load control: *uniform policy* and *discriminatory* policy.

In a *uniform policy*, the total number of transactions active in the system (regardless of their class) is controlled. In contrast, in a *discriminatory* policy, only the number of transactions in Class 1 is controlled. More precisely, let N_1 be the number of active transactions of Class 1 and N_2 be the number of active transactions of Class 2. In a uniform policy, for a multiprogramming level M, the transaction manager makes sure that at any given time, $N_1 + N_2$ is less than M. In a discriminatory policy, N_1 is restricted to M, and there is no limit on N_2, which equals the number of Class 2 transactions active in the system. We conducted a preliminary experiment to study the relative behavior of the two policies.

The parameters used in this experiment correspond to the first case study discussed in Chapter 5. The results, however, have general applicability to the systems that have a two class workload, with Class 1 consisting of long transactions and Class 2 transactions consisting of short transactions. The results are shown in Figure 4.3. For the uniform policy, the multiprogramming level denotes $N_1 + N_2$ and for a discriminatory policy it represents N_1. We found that the uniform policy gives much lower response time for Class

1 transactions and much higher response time for Class 2 transactions as compared to the corresponding values given by the discriminatory policy. This is natural because in the discriminatory policy, Class 2 transactions never have to wait in the ready queue and are processed immediately. Since it is desired for the Class 2 transactions to be really quick while a higher response time is typically acceptable for longer Class 1 transactions, we selected the discriminatory load control policy for all our subsequent experiments.

4.6 Other Approaches to Quantitative Evaluation

Extensive surveys of the work on concurrency control performance studies can be found elsewhere (Sevcik 1983; Tay 1987; Thomasian 1991). In this section, we briefly mention the difference between our work and the previous work.

Most performance studies of two-phase locking do not model the database as a graph and adopt a random access pattern of the database workload (Agrawal, Carey and Livny 1987; Tay 1987; Ryu and Thomasian 1990; Yu, Dias and Lavenberg 1993). The performance studies of concurrency control algorithms for B-Trees restrict the database to a tree (Johnson and Shasha 1993; Srinivasan and Carey 1993), and transactions always begin from the root. Instead, in our work, we have considered a general graph as a model of the database and access patterns such as breadth-first and depth-first traversal which are frequently encountered in the workloads of knowledge based systems (Genesereth and Nilsson 1987).

Performance of a multi-level implementation of 2PL, with a detailed model of recovery, has been studied for complex objects (Weikum and Hasse 1993). In this study, the database was modeled as a random graph. Only one class of transactions was considered, and the transactions accessed an object and some of its sub-objects. Using our terminology, one can see that these transactions are limited to traversal of up to two levels. On the other hand, this study had a model of recovery which we have not considered in our work so far.

4.7 Chapter Summary

In this chapter we described the simulation model that we use in evaluating the DDG policy. Our model was an adaptation of the one that was previously used in the evaluation of concurrency control algorithms. We also validated our simulation of two-phase locking against existing results. We described our cost model for various operations in concurrency control and our approach to measuring and incorporating them in the simulation model. We considered a load control strategy for a two class work load and settled on an approach in which we control the number of long transactions (Class 1) active at any give time but process the short transactions (Class 2) as soon as they arrive. We also discussed various approaches for the evaluation of concurrency control algorithms.

Chapter 5

Quantitative Evaluation

In this chapter, we describe the quantitative evaluation of the Dynamic Directed Graph (DDG) policy. The issues addressed in the evaluation are two-fold: influence of the knowledge base characteristics on the performance of the DDG policy and its comparison to two-phase locking (2PL). In the comparison of the two algorithms, our goal is to identify the parameter values for which one can obtain substantial improvements in response time by using the DDG policy as compared to 2PL. Our analysis is accomplished with the help of a simulation model and knowledge bases derived from real applications.

We consider three case studies for which we have studied the performance of the DDG policy. We made significant efforts in obtaining access to real knowledge bases and workloads. We decided not to use random graphs in our analysis, as they do not capture the knowledge base structure in a realistic fashion.

The first case study that we consider is based on a knowledge base used in the process control of a nuclear power plant. The second case study uses a knowledge base that was developed at AT&T Bell Labs for demonstration purposes. The third case study uses an enterprise modeling knowledge base being developed at University of Toronto. The source listings for the knowledge bases can be found elsewhere (Chaudhri 1994).

5.1 Case Study 1: APACS

The Advanced Process Analysis and Control System (APACS) is a knowledge based system that is being designed in collaboration with Ontario Hydro and CAE Electronics (Mylopoulos et al. 1992b). We first describe the APACS project, give the parameter settings and then give our experimental results.

5.1.1 Overview of the APACS Project

Ontario Hydro is a major supplier of electric power in the province of Ontario. A substantial portion of its electric power is produced in its nuclear power plants.

The Advanced Process Analysis and Control System (APACS) is a prototype of a process control system intended to aid a nuclear power plant operator during an emergency. The prototype handles the feedwater system of a power plant that supplies hot, pressurized, de-mineralized water to the boilers.

APACS is a distributed system that includes a number of functional components for data acquisition, alarm filtering, plant analysis, situation assessment, and diagnosis. It

Semantic Relationship	Number of Nodes/ Roots/Leaves	Fan In Max/ Average	Fan Out Max/ Average	Depth Max/ Average	Size of SCCs
`isA`	298/1/229	1/1	21/4.3	10/6.24	0
`instanceOf`	2707/184/2523	1/1	266/13.71	2/2	0
`linkedTo`	204/2/19	4/1.16	6/1.27	27/18	22
`partOf`	417/29/337	1/1	11/4.85	6/3.83	0
`equipment`	1027/755/272	17/2.77	1/1	2/2	0

Table 5.1: Structural properties of semantic relationships in the APACS KB

consists of several expert systems and conventional software systems. The knowledge base that we have taken as a test case for our study is used for alarm filtering and stores information about the components in the feedwater system, for example, boilers, valves, preheaters, alarms, etc.

The alarm filtering system handles a total of approximately 2000 types of alarms and provides displays designed to assist the operators. Each alarm indicates the status of a feedwater system component – for example, whether the boiler temperature is too high or normal. During plant transients or equipment failures, hundreds of alarms may be activated during a short period. The alarm filtering system preprocesses all the alarm messages, selects the most relevant alarms and presents them to the operator for further actions. There are always several processes active that monitor the current state of the plant and perform fault diagnosis, if necessary.

The objects represented in the knowledge base (boilers, valves, preheaters, alarms, etc.) are organized into a collection of classes, each with its own subclasses, instances and semantic relationships to other classes. For our experiments, we view this knowledge base as a directed graph. Each class and each instance is represented by a node. There are 2821 nodes in this graph. There is an edge between two nodes if they have some semantic relationship. For example, there is an edge from node A to node B, if the object represented by A is a part of the object represented by node B.

There are several semantic relationships in this knowledge base. We did an analysis of the frequency of occurrence of all the semantic relationships and identified the five most frequently occurring relationships. These relationships and some of the structural properties of the graphs defined by them are listed in Table 5.1. Not all of these graphs are connected. The `isA` relationship captures class-subclass relationship, the `instanceOf` relationship represents the instances of a class; the `linkedTo` relationship stores how the components are linked to each other in the power plant; the `partOf` relationship indicates the part-subpart relationship; and finally, the `Equipment` relationship associates pieces of equipment with each alarm.

From Table 5.1, we can see that the `isA` and the `partOf` graphs are trees. The `instanceOf` and `Equipment` graphs are shallow in the sense that the depth of these graphs is only two. The `linkedTo` graph is the only graph that has cycles. The depth of such a graph is the length of the longest path in the depth-first spanning tree. The `linkedTo` graph has the largest depth of all these graphs.

Transaction Type	Proportion (%)
Class 1	25 Terminals
traversal along	
`isA+instanceOf`	62
`linkedTo`	25
`partOf`	13
Class 2	25 alarms/second
Update	49.95
Look up	49.95
Change Semantic Relationship	0.1

Table 5.2: APACS workload

5.1.2 The APACS Workload

The APACS workload presented in this section, and shown in Table 5.2, was obtained by interviewing the scientists who developed the knowledge base (Wang 1993). They had observed the operation of this system over an extended period of time. We asked them to list the operations performed on the knowledge base and their relative frequencies of occurrence.

The knowledge base receives two classes of transactions. Transactions in the first class are generated because of the diagnostic and monitoring processes which perform a (partial) traversal along some semantic relationship. The distribution of traversals along different semantic relationships is shown in Table 5.2. The symbol "+" represents union of the graphs. Thus, `isA+instanceOf` represents the union of the graphs defined by the `isA` and `instanceOf` relationships. The choice of dividing the transactions into two classes was made because the traversal and lookup/update transactions naturally form distinct types of transactions. We could have further sub-divided the traversal transactions into three classes depending on the semantic relationship they traversed, but we felt that this extra detail would not offer any additional insight.

When the system is fully developed, it is expected that there will be 25 terminals active at any time issuing such transactions. Most of the time, these transactions perform read accesses to the knowledge base, but some times they may also perform updates, with write probability[1] varying between 0.0 and 0.5. The transactions in the second class are generated due to the alarms and the data collection components of APACS. These transactions are short, and half the time involve lookup of an attribute value, and half the time an update, accessing either one or two data items. Occasionally, they may also change a semantic relationship between two objects. Whenever there is a fault in the plant, up to 25 alarms may be triggered per second. It is required that the Class 2 transactions should have an extremely quick response time. For the rest of the thesis, we call the workload shown in Table 5.2 the *baseline* APACS workload.

The values of the other parameters are shown in Table 5.3. For Class 1 transactions, we fixed the NumTerminals to 25 and *ThinkTime* to 0. For Class 2 transactions, we fixed the ArrivalRate to 25 transactions per second. The values of the parameters UnitCPU and DiskIOTime were established based on the characteristics of the APACS and were fixed at 5 msec and 10 msec respectively. Initially, *HitRate* was set to 0.0, and then was varied

[1] Write probability is the proportion of accesses performed by a transaction that are updates.

Parameter	Value
NumTerminals	25
ThinkTime	1s
ArrivalRate	25 transactions/sec
UnitCPU	5 msec
diskIOTime	10 msec
HitRate	0.0
TxnLength	uniformly distributed between 3-6 levels
WriteProb	0.0,0.2,0.4
AccessPattern	breadth-first, depth-first
RateUpdateStr	0.1%

Table 5.3: Parameters for APACS

between 0.0 and 1.0 to study its influence on the results.

The graph corresponding to the APACS knowledge base was given as an input to the simulation program. The transactions were hand-coded to represent different access patterns, attribute accesses, and insert/delete operations. The Class 1 transactions traverse from 3 to 6 levels. The starting nodes for Class 1 transactions was selected in a way that the number of levels accessed by a transaction was always in the range 3-6. This means that the Class 1 transactions access, on average, 130 data items, with the longest transaction accessing more than 1000 data items.

5.1.3 Range of Experiments

Our experiments can be classified into two general themes: influence of knowledge base characteristics on implementation choices for concurrency control and quantitative comparison of the DDG policy with 2PL.

To study the interaction of knowledge base characteristics with concurrency control, we consider the choice of semantic relationship that should be used for concurrency control and the choice of traversal strategy. Interestingly, in some cases, these choices can make a big difference in the response time. These issues are of relevance, because we chose to model the knowledge base as a directed graph.

To compare the DDG policy with 2PL, we varied workload parameters and system parameters. Workload parameters are defined by the transactions and the system parameters depend on the hardware configuration that is used to implement the knowledge base. These parameters are listed below:

- Workload Parameters: Write Probability, distribution of traversals along different semantic relationships, transaction length, think time of Class 1 transactions, arrival rate of Class 2 transactions, rate of insertions and deletions into the graph.

- System Parameters: Buffer hit ratio, resource requirement, concurrency control overheads.

The baseline experiment for the comparison of the two policies is described in the first part of Section 5.1.5. We consider this experiment in significant detail. The rest of experiments in Section 5.1.5 represent perturbations of one or more parameters from the baseline experiment. Let us now describe our experiments.

5.1.4 Influence of Knowledge Base Characteristics on Concurrency Control

In this section, we study the interaction of knowledge base features with the performance of the DDG policy.[2] We accomplish this by studying the choice of semantic relationship to be used for concurrency control and the choice of traversal strategy. These choices can also be thought of as tuning decisions or implementation alternatives on part of the system administrator who can set these values to optimize the performance of the DDG policy.

More Semantic Information is not Always Better

As discussed in Section 5.1.1, there are five different semantic relationships in the APACS knowledge base: `isA`, `instanceOf`, `linkedTo`, `partOf` and `Equipment`. One can form $2^5 - 1(= 31)$ different graphs by taking the union of one or more of the graphs defined by each of these semantic relationships. We need to decide which of these should be used for the purposes of concurrency control — this choice defines the sub-graph with respect to which the locking rules of the DDG policy are applied.

The tradeoff in choosing the graph is that using a combination of graphs gives more information to the concurrency control algorithm that can be used to determine from where a transaction should begin execution. On the other hand, this may require some transactions to lock more entities than they actually need. Let us explain this in more detail.

The graph that is made known to the CCM is used by it to get information about the entities that may be accessed by a traversal transaction, and CCM uses that information to compute the dominator of the transaction. To understand this more clearly, let $G_{\texttt{isA}}$, $G_{\texttt{linkedTo}}$ and $G_{\texttt{partOf}}$ denote the graphs defined by the `isA`, `linkedTo` and `partOf` relationships respectively. If the graph seen by the CCM is $G_{\texttt{isA}}$, it can determine the entities that might be accessed by the transactions that traverse the `isA` relationships. If the graph seen by the CCM is a union of $G_{\texttt{isA}}$ and $G_{\texttt{linkedTo}}$, it can determine the entities that might be accessed by transactions that traverse `isA` and transactions that traverse `linkedTo` relationships. Thus, using a graph that is union of more semantic relationships helps the CCM to determine the entities that might be accessed by transactions that traverse those relationships. On the other hand, a less desirable consequence of using a graph that is union of more semantic relationships is that it may require a transaction to lock more items. In this section, we analyze and evaluate this tradeoff.

Since transactions request locks in the order defined by the graph used for concurrency control, it makes sense to consider only those graphs that are often traversed by transactions, thus making the transaction access patterns similar to lock acquisition patterns. For example, in the APACS workload, the graph defined by the union of `isA` and `instanceOf` relationships is traversed most often, and therefore, it should be a candidate for the graph to be used for concurrency control.

Once we have identified a graph that is traversed most often, the next question is whether we should consider its union with some other semantic relationships. The most important thing to know about the transaction access patterns is the dominator of the set of nodes that a transaction is going to access. (Recall that a transaction begins by locking

[2] We do not consider 2PL in these experiments, because it does not take into account the knowledge base characteristics.

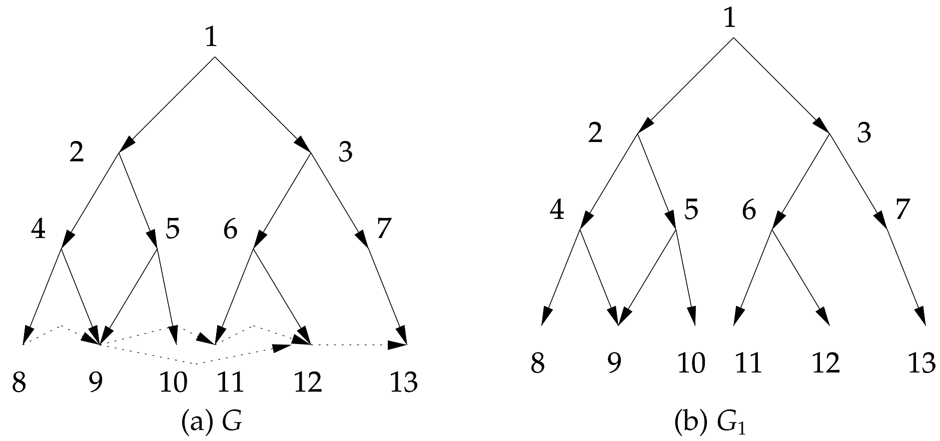


Figure 5.1: An example graph

the dominator of all the nodes that it expects to access.) Let us now understand how this computation is affected by the choice of graph.

To compute the dominator, we need some information about a transaction's *access set*, that is the set of nodes it is going to access. Since Class 1 transactions do not explicitly specify their access set, we designed two possible implementations for obtaining this information: *static* and *dynamic*. These implementations were described in Chapter 3. To recap, in the *static implementation*, when the knowledge base is compiled, we associate with each node A, a node B, that dominates all descendants of B such that no proper descendant of B satisfies this property. Node B is called the dominator node of A. A transaction begins by locking the dominator node of the node for which it makes the first lock request, thus, implicitly assuming that it may access any of the descendants of the first node. In the *dynamic implementation*, we obtain an estimate of the access set of a transaction by preprocessing the graph it is going to traverse. The access set includes the descendants of the first node up to the number of levels the transaction expects to traverse. The transaction begins by locking the dominator of this set. The static implementation minimizes the run-time cost, because it does not require any preprocessing. The dynamic implementation obtains more accurate estimates of the access set, because it uses the run-time information about a transaction.

To understand the influence of the choice of graph, consider the graph G shown in Figure 5.1(a). This graph has two kinds of edges – type 1 and type 2, which are shown by solid and dotted lines respectively. Let us call the graph defined by type 1 edges as G_1 (shown in Figure 5.1(b)), the graph defined by type 2 edges as G_2 (not shown separately). Consider two transactions T_1 and T_2. Suppose T_1 begins from node 6 and traverses G_1 up to two levels, and T_2 begins from node 8 and traverses G_2 up to 3 levels. If the graphs do not change in the meantime, T_1 will access nodes 6, 11 and 12, and T_2 will access nodes 8, 9, 11 and 12.

First, consider the static implementation. If we use G_1 for concurrency control, T_1 begins from node 6 and locks nodes 11 and 12. Since, in G_1, there are no descendants of node 8, its dominator node is node 8 itself. T_2 begins by locking node 8, realizes it cannot lock 9, restarts from node 4, realizes that it cannot lock node 11, and then restarts from node 1 to finish execution. On the other hand, if we use G for concurrency control, the dominator node for 6 is node 1. T_1 begins by locking node 1, and before it can lock 6, 11 and 12, it also has to lock nodes 1, 2, 4, 5, 8, 9 and 3 to satisfy the locking rule L5a. Thus using G,

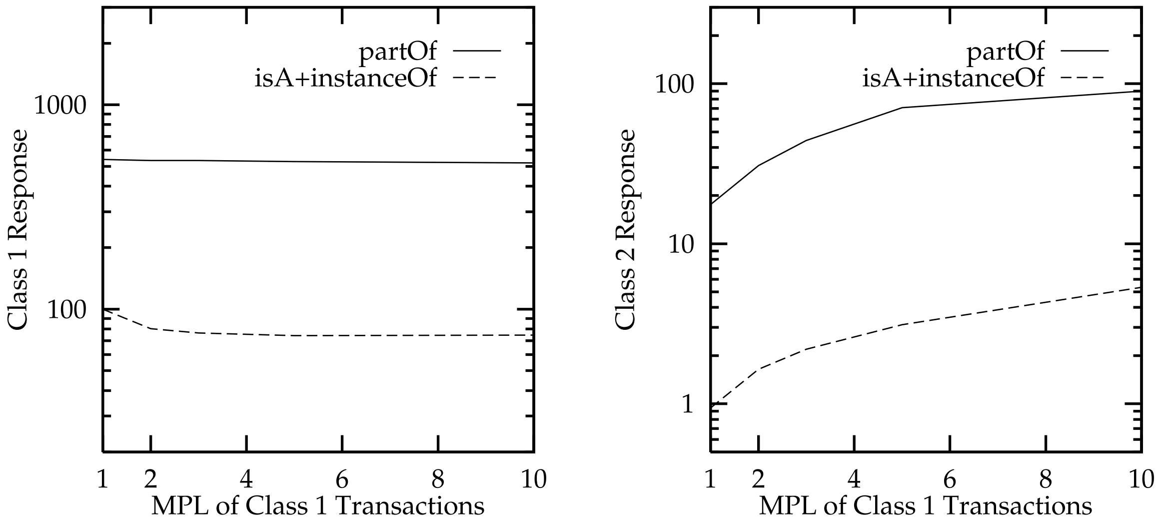
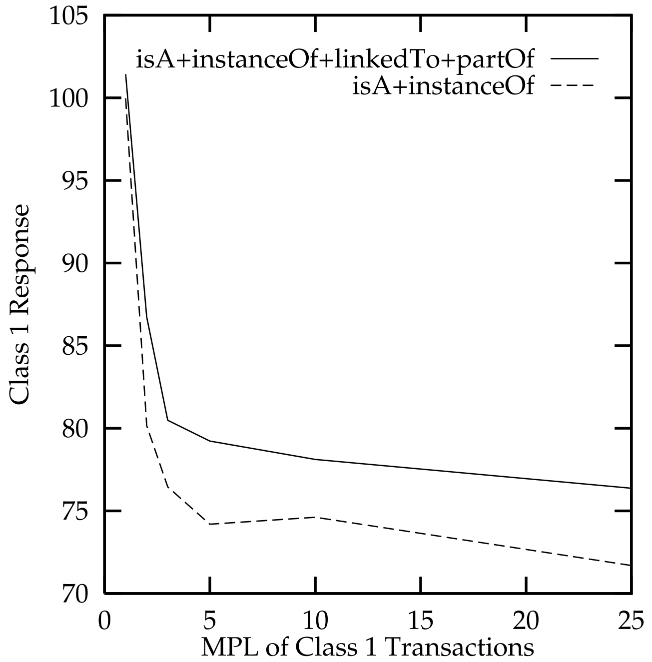


Figure 5.2: Choice of graph: Use a graph that is traversed with high frequency

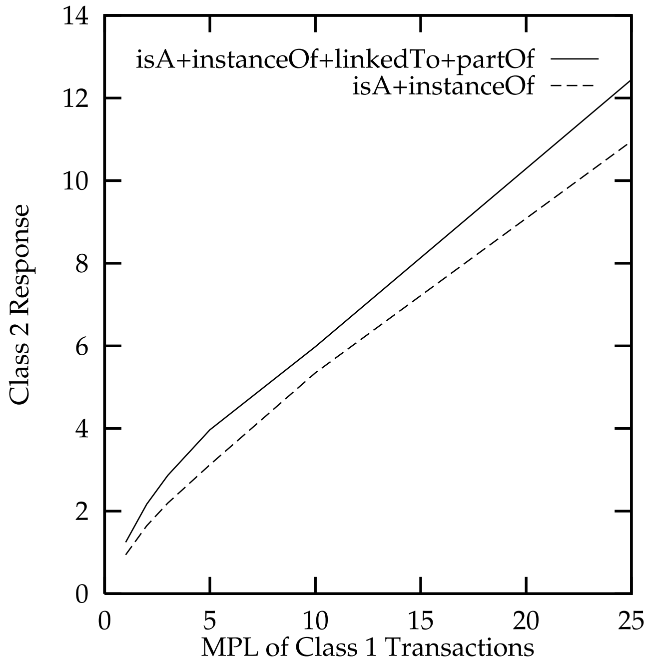
that has more edges than G_1, requires T_1 to lock more nodes as compared to the number of locks necessary when the CCM uses G_1. Since in G, node 8 has nodes 9, 11, 12 and 13 as its descendants, node 1 is its dominator node. T_2 begins by locking node 1 and successfully locks all the nodes that it needs in the first attempt. Thus, if we use G_1 for concurrency control, the transactions that traverse this graph need not lock more nodes than they need, but the transactions that do not follow it, may need to restart. If we use G, the node from which T_1 and T_2 should begin execution is accurately known, thus avoiding restarts, but for some transactions the number of nodes that they need to lock increases. This shows an inherent tradeoff in the choice of the graph that is used for concurrency control: using G_1 may lead to some aborts, whereas using G may require some transactions to lock more nodes.

Now, consider the dynamic implementation. In this case, if we use G_1 for concurrency control, T_1 begins from node 6, as before, and locks nodes 11 and 12. For T_2, we perform a pre-traversal of G_2 for three levels starting at node 8, and estimate that T_2 might access nodes 8, 9, 11 and 12. We compute the dominator of these nodes with respect to G_1, which is node 1, and therefore, T_2 begins by locking node 1 and avoids restarts. If we use G as the graph for concurrency control, T_1 and T_2 both begin from node 1, completing successfully, but T_1 locks more nodes than it actually needs. In this case, using G_1 has the advantage that the number of locks needed by T_1 does not increase. Using G does not offer any advantage, and in fact, is detrimental for T_1 as it now has to lock more nodes.

To test the above arguments, we conducted several experiments. First, we wanted to confirm that it is indeed better to use semantic relationships that are traversed more often. We conducted an experiment with the dynamic implementation. In one setting, we used the graph defined by the union of `isA` and `instanceOf`, and in the other, the graph defined by the `partOf` relationships. We compare the response times in Figure 5.2. We can see that using the graph defined by the union of `isA` and `instanceOf` leads to clearly better response times for both classes of transactions.



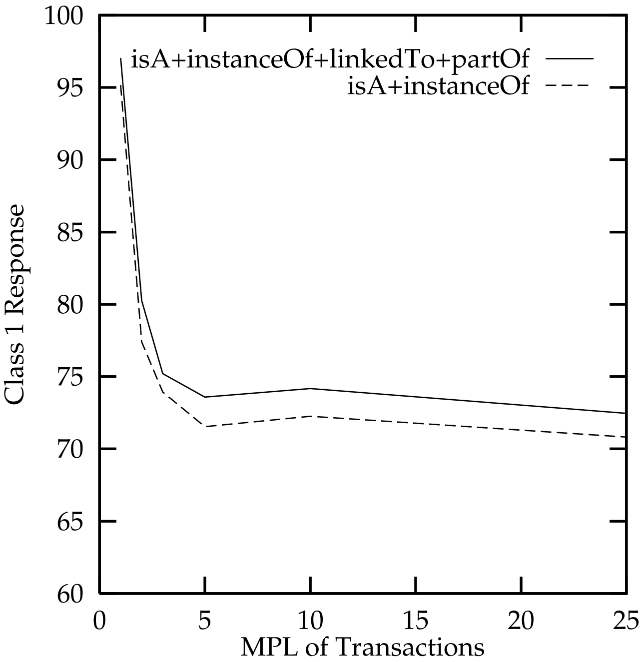
I=isA+instanceOf+ linkedTo+partOf II=isA+instanceOf	
MPL	Probability(I $\geq$ II)
1	0.64
2	0.97
3	0.88
5	0.94
10	0.87
25	0.77



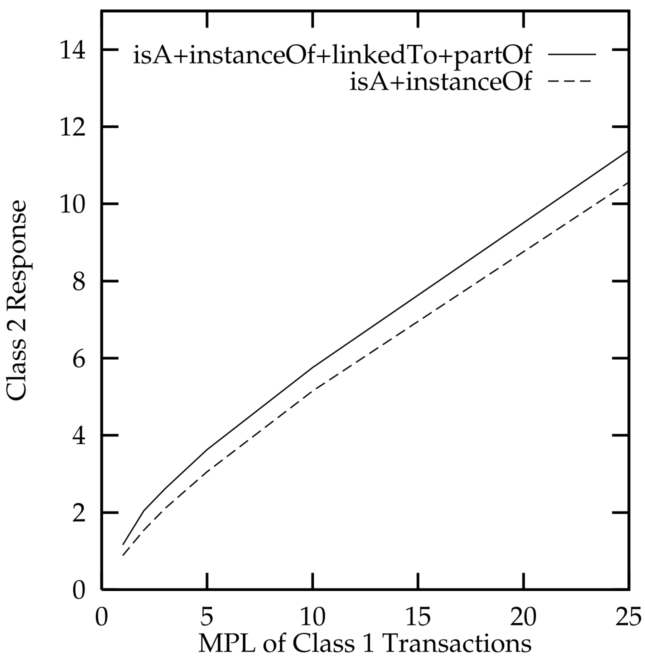
I=isA+instanceOf+ linkedTo+partOf II=isA+instanceOf	
MPL	Probability(I $\geq$ II)
1	1.00
2	1.00
3	1.00
5	1.00
10	0.99
25	0.94

The detailed computation of Probability(I $\geq$ II) can be found in Table B.1 on page 138.

Figure 5.3: Choice of graph: Dynamic implementation



I=isA+instanceOf+ linkedTo+partOf II=isA+instanceOf	
MPL	Probability(I $\geq$ II)
1	0.68
2	0.81
3	0.66
5	0.75
10	0.78
25	0.64



I=isA+instanceOf+ linkedTo+partOf II=isA+instanceOf	
MPL	Probability(I $\geq$ II)
1	1.00
2	1.00
3	1.00
5	1.00
10	1.00
25	0.88

The detailed computation of Probability(I $\geq$ II) can be found in Table B.2 on page 139.

Figure 5.4: Choice of graph: Static implementation

Next, using the dynamic implementation, we compared the response times for the cases when we use `isA` + `instanceOf` graph and when we use the `isA` + `instanceOf` + `linkedTo` + `partOf` graph. As expected, we obtained lower response times for the case when we used the `isA` + `instanceOf` graph. The results are shown in Figure 5.3. The confidence levels for each multiprogramming level are also shown and were computed using the approach described in Section A.4. We decided to choose the union of `isA` and `instanceOf` as the graph for concurrency control for the rest of our experiments.

Finally, we considered a static implementation and ran experiments for the same two cases as considered in the previous paragraph. In this case, surprisingly, the `isA` + `instanceOf` graph led to lower response times. The response times are shown in Figure 5.4. On analyzing the response times, we found that the cost of increased locking by using the `isA` + `instanceOf` + `linkedTo` + `partOf` far outweighed the cost of aborts caused when we use `isA` + `instanceOf` graph. This shows that using more semantic information may not always be useful, and one has to analyze the tradeoffs before deciding how much (or which) semantic information to use.

In summary, we can say that the choice of semantic relationship to be used for concurrency control is an important design decision for a graph-oriented locking policy. The graphs defined by the semantic relationships that are traversed by transactions more frequently are the prime candidates for this purpose. The comparison of response time obtained using the `partOf` graph as compared to `isA` + `instanceOf` graph showed that this choice can lead to an orders of magnitude difference in the response time. For a static implementation, the choice depends on the tradeoff between the cost of restarts and the time spent in the blocked queue. For a dynamic implementation, since the dominator is obtained by preprocessing, it is better to have a graph that minimizes the number of locks needed by a large percentage of transactions. In general, more semantic information may or may not be useful — on one hand, it provides information about the access patterns of transactions that traverse along semantic relationships, and on the other, it may require locking more items than are actually necessary. The choice of semantic relationship to be used for concurrency control should be made based on this tradeoff.

Depth-first Traversal is Better Than Breadth-first Traversal

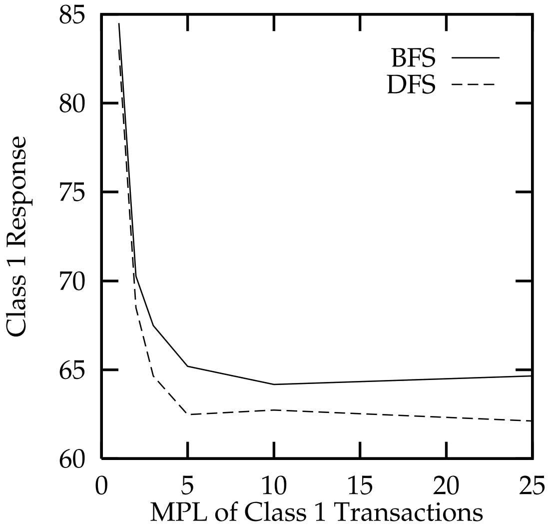
We consider two types of *transaction access patterns*: breadth-first traversal (BFT) and depth-first traversal (DFT). In some applications, the traversal pattern is specified as part of the transaction, and it is not possible to make a choice. In others, the transaction specifies that a certain portion of the knowledge base should be traversed and leaves the choice of traversal pattern to the implementor. In the APACS application programs, the traversal strategy is not specified, and is therefore, an important design decision.

The choice of traversal strategy is highly dependent on the workload. In general, the DFT strategy facilitates an earlier release of locks and on average holds a smaller number of locks as compared to BFT. For example, consider the graph shown in Figure 5.1(b), and a traversal transaction, T_3, that wishes to traverse the whole graph starting at node 1, and updates the values of nodes 2 and 3. If T_3 does this in a breadth-first fashion, and the graph does not change in the meantime, it can acquire and release locks in the following order:

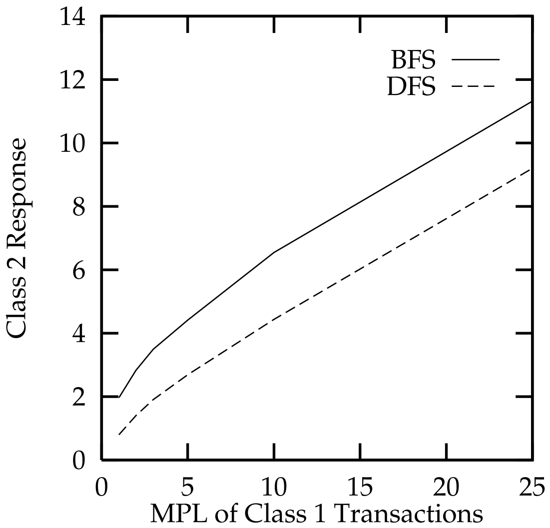
(`LS` 1) (`LX` 2) (`LX` 3) (`U` 1) (`LS` 4,5) (`LS` 6,7) (`LS` 8,9,10) (`U` 4,5,8,9,10,2) (`LS` 11,12,13) (`U` *)

If T_3 performs the same operations in a depth-first fashion, its execution can be as follows:

(`LS` 1) (`LX` 2) (`LS` 4,8,9,5,10) (`U` 4,8,9,5,10) (`LX` 3) (`U` 1,2) (`LS` 6,11,12,7,13) (`U` *)



I=BFT II=DFT	
MPL	Probability(I $\geq$ II)
1	0.68
2	0.73
3	0.84
5	0.84
10	0.67
25	0.81



I=BFT II=DFT	
MPL	Probability(I $\geq$ II)
1	1.00
2	1.00
3	1.00
5	1.00
10	1.00
25	1.00

The detailed computation of Probability(I $\geq$ II) can be found in Table B.3 on page 140.

Figure 5.5: Choice of traversal strategy: absolute response time

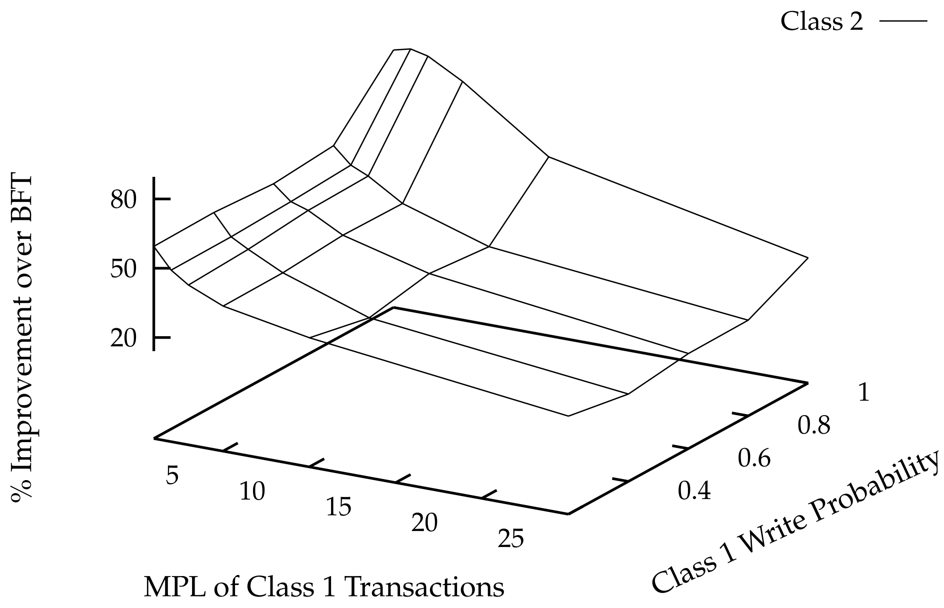


Figure 5.6: Choice of traversal strategy: relative response time

By comparing the above two schemes, we can see that by using the DFT, we keep node 1 locked longer, but we are able to release locks on nodes 4, 5, 8, 9 and 10 sooner. We need to keep node 1 locked longer, because it takes longer to lock all its successors: for each of its successors, we first process all its descendants, and then lock the next successor. We are able to release some locks sooner, because once we have finished processing all descendants of node 2, which is locked in exclusive mode, locks on descendants are not necessary for enforcing the locking rules of the DDG policy (in specific, locking rule L5b as defined early in Section 2.5.3) for any nodes that T_1 locks in the future. Thus, there is an inherent tradeoff in the choice of the traversal strategy. The BFT strategy tends to release locks on the nodes that are closer to the root of the graph sooner, whereas the DFT strategy tends to release locks on the nodes closer to the leaves sooner. If in a given workload, the locks requested on nodes near the root are in the shared mode, the DFT strategy would be a better choice. On the other hand, if the locks requested on the nodes near the root are in an exclusive mode, the DFT strategy has a greater potential of causing a bottleneck.

The above is true only if a transaction requests some exclusive locks. Had T_1 requested only shared locks, it could not have released any locks before it locks the last item. This is because if it releases any lock before locking the last item, it would not be able to satisfy condition L5b for any of the nodes that it may lock later on.[3] If all accesses are in shared mode, the choice of traversal strategy is irrelevant.

In Figure 5.5, we show the response time of both Class 1 and Class 2 transactions as Class 1 transactions use different traversal strategies. We also show the probability that the response time given by the BFT strategy is higher than the DFT strategy. These results are for the APACS baseline workload when the write probability for Class 1 transactions is set to 0.4. We can see that DFT leads to much better response time for Class 2 transactions whereas Class 1 response times are almost unaffected. To cross check the results, we

[3]This rule cannot be relaxed. Doing so leads to non-serializable schedules (Yannakakis 1982a).

compared the time spent by a transaction in different stages of execution. We found that under BFT, the transaction spends more time in the blocked queue as compared to the time spent in the blocked queue under the DFT strategy.

Next, to see if DFT would cause a bottleneck at high write probabilities, we conducted experiments by changing the write probability of Class 1 transactions between 0.0 and 1.0. We found that, even at high write probabilities, by using DFT, Class 2 response times showed consistent improvement. In Figure 5.6, we show the percentage improvement in the Class 2 response times as obtained by using DFT as compared to BFT. We do not show the improvement in Class 1 response time as it was found to be insignificant. If $R(j)_{BFT}$ and $R(j)_{DFT}$ are the mean response times of transactions in the Class j then the percentage improvement of DFT over BFT is computed as $(R(j)_{BFT} - R(j)_{DFT})/R(j)_{BFT} * 100$. We can see that, even at high write probabilities, by using DFT, Class 2 response times showed consistent improvement. The improvements in response times for Class 1 are not significant.

These results suggest that, for the APACS workload, holding fewer locks at a given time has more influence on response time than holding locks on nodes near the root for a longer time. This shows that in a multi-user environment, with a workload similar to APACS, DFT is a more desirable traversal strategy than the BFT.

5.1.5 Comparison of the DDG and 2PL Policies

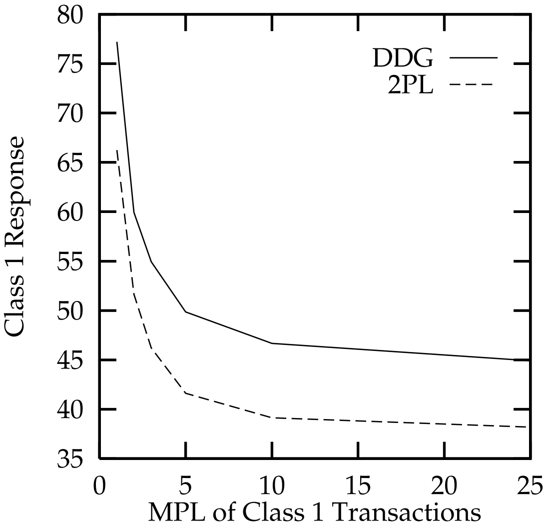
For all experiments in this section, we use the union of `isA` and `instanceOf` as the graph for concurrency control, and the DFT strategy. We vary other parameters one at a time and compare the performance of the DDG and 2PL policies.

Baseline Experiment: DDG Speeds up Class 2 at Heavy Data Contention

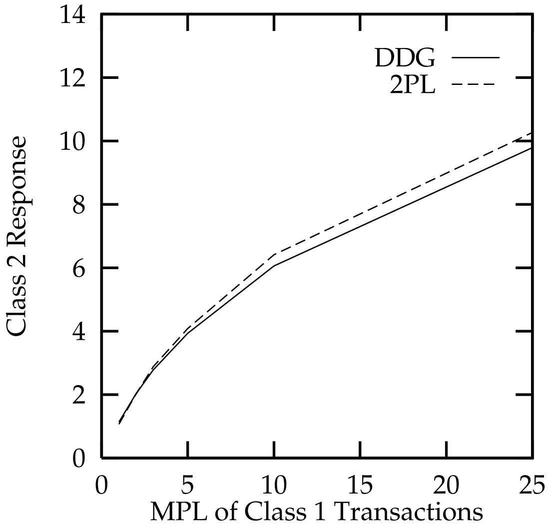
The write probability of transactions directly affects the data contention. In case of the DDG policy, a higher percentage of writes also means that a transaction is able to release more locks before the locked point compared to a situation in which it does not perform any updates. For example, consider transaction T_3 of the previous section. This transaction wishes to traverse the graph shown in Figure 5.1(b), starting at node 1. If T_3 locks nodes 2 and 3 in exclusive mode, it can release some of the locks before it commits – as shown in Section 5.1.4. On the other hand, if it does not update nodes 2 and 3, and thus locks them in shared mode, it cannot release any locks until it commits (a direct consequence of the locking rule L5b). Thus, the DDG policy can reduce data contention when there are updates.

To quantify this effect, we conducted experiments for various values of the Class 1 write probability: 0.0, 0.1, 0.2, 0.3 and 0.4. In Figures 5.7 and 5.8, we show the absolute values of the response time for the two classes for write probability 0.0 and 0.4 along with the probability of indicated difference. More details about the computation of the probability of the indicated difference may be found in Section A.5. In Figure 5.9, we show the surface plots of the percentage improvements in response time over a range of write probabilities between 0.0 and 0.4. The percentage improvement was defined in Section 4.2.

From these results, we see that the DDG policy improves the Class 2 response time at high write probabilities, while there is some degradation in Class 1 response time. To understand this behavior better, let us look at the break up of Class 1 and Class 2 response time at write probability 0.4. In Table 5.4, we show time spent by a Class 2 transaction in



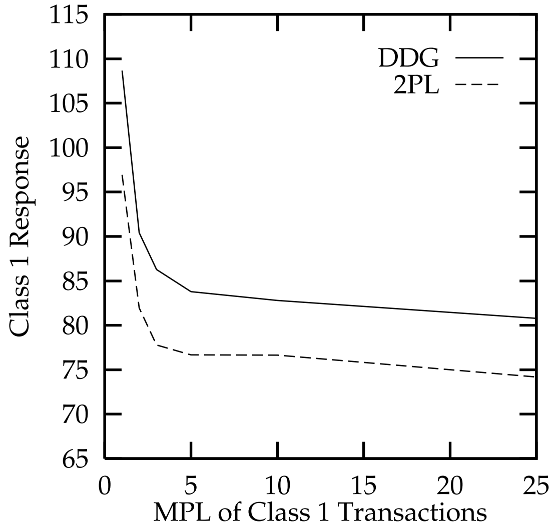
MPL	Probability($R_{2PL} \geq R_{DDG}$)
1	0.00
2	0.01
3	0.00
5	0.00
10	0.00
25	0.04



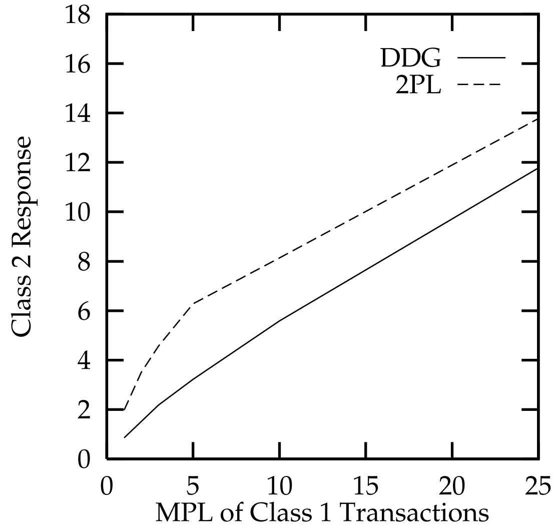
MPL	Probability($R_{2PL} \geq R_{DDG}$)
1	0.10
2	0.46
3	0.79
5	0.81
10	0.93
25	0.88

The detailed computation of Probability($R_{2PL} \geq R_{DDG}$) can be found in Table B.4 on page 141.

Figure 5.7: Comparison of DDG and 2PL at write probability 0.0 – absolute values



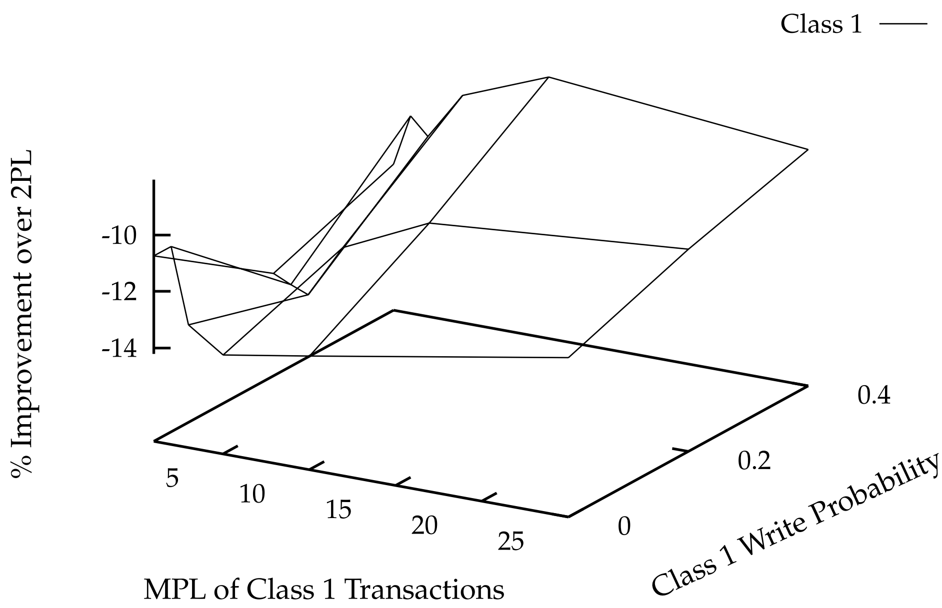
MPL	Probability($R_{2PL} \geq R_{DDG}$)
1	0.00
2	0.08
3	0.01
5	0.02
10	0.03
25	0.15

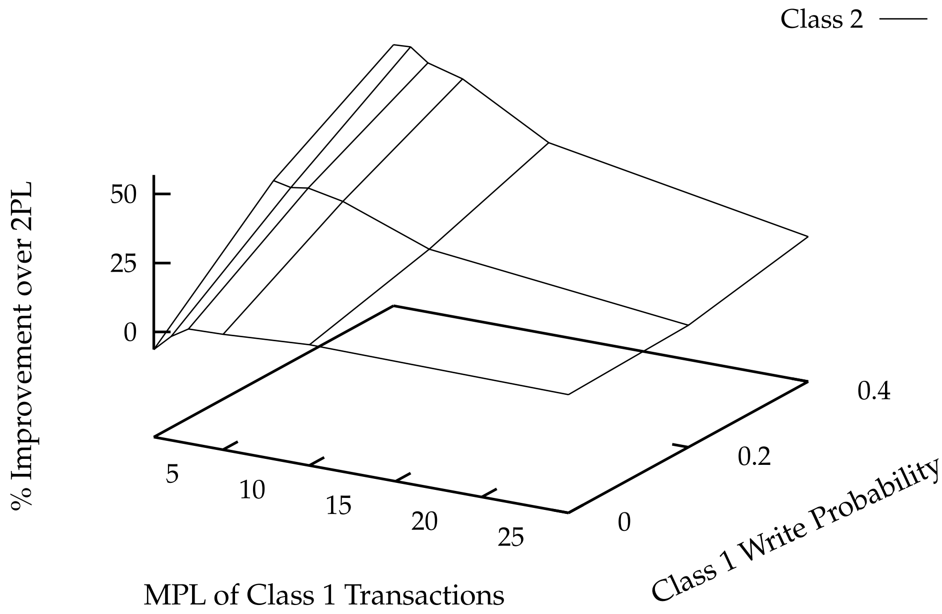


MPL	Probability($R_{2PL} \geq R_{DDG}$)
1	1.00
2	1.00
3	1.00
5	1.00
10	1.00
25	0.98

The detailed computation of Probability($R_{2PL} \geq R_{DDG}$) can be found in Table B.5 on page 142.

Figure 5.8: Comparison of DDG and 2PL at write probability 0.4 – absolute values





(Positive improvement indicates that DDG performs better than 2PL.)

Figure 5.9: Comparison of DDG and 2PL – percentage improvements

MPL	1	2	3	5	10	25
Ready Queue	0.00	0.00	0.00	0.00	0.00	0.00
CPU Requirement	0.01	0.01	0.01	0.01	0.01	0.01
Disk Requirement	0.02	0.02	0.02	0.02	0.02	0.02
CPU Wait	0.07	0.10	0.11	0.11	0.12	0.12
Disk Wait	0.09	0.13	0.14	0.15	0.15	0.15
Overhead	0.00	0.00	0.00	0.00	0.00	0.00
Blocked Queue	1.78	3.25	4.29	5.99	7.84	13.48
Aborts	0.00	0.00	0.00	0.00	0.00	0.00
Response Time	1.97	3.50	4.57	6.28	8.14	13.78

2PL at Class 1 Write Probability=0.4

MPL	1	2	3	5	10	25
Ready Queue	0.00	0.00	0.00	0.00	0.00	0.00
CPU Requirement	0.01	0.01	0.01	0.01	0.01	0.01
Disk Requirement	0.02	0.02	0.02	0.02	0.02	0.02
CPU Wait	0.03	0.04	0.04	0.04	0.05	0.05
Disk Wait	0.03	0.04	0.04	0.04	0.04	0.05
Overhead	0.002	0.002	0.002	0.002	0.002	0.002
Blocked Queue	0.77	1.40	2.06	3.09	5.45	11.65
Aborts	0.00	0.00	0.00	0.00	0.00	0.00
Response Time	0.86	1.51	2.18	3.22	5.58	11.78

DDG at Class 1 Write Probability=0.4

Table 5.4: Break up of Class 2 response time

MPL	1	2	3	5	10	25
Ready Queue	92.99	75.32	68.32	61.15	45.66	0.00
CPU Requirement	0.95	0.95	0.94	0.95	0.95	0.95
Disk Requirement	1.89	1.90	1.88	1.89	1.90	1.90
CPU Wait	0.39	0.67	0.79	0.88	0.90	0.89
Disk Wait	0.66	1.07	1.23	1.35	1.38	1.37
Overhead	0.00	0.00	0.00	0.00	0.00	0.00
Blocked Queue	0.05	2.04	4.53	10.36	25.76	68.91
Aborts	0.00	0.04	0.08	0.11	0.10	0.14
Response Time	96.92	81.97	77.78	76.69	76.64	74.18

2PL at Class 1 Write Probability=0.4

MPL	1	2	3	5	10	25
Ready Queue	104.12	83.10	75.81	66.75	49.42	0.00
CPU Requirement	0.95	0.94	0.95	0.95	0.95	0.95
Disk Requirement	1.89	1.89	1.90	1.89	1.90	1.89
CPU Wait	0.44	0.78	0.97	1.12	1.17	1.16
Disk Wait	0.66	1.01	1.18	1.30	1.34	1.34
Overhead	0.27	0.27	0.27	0.27	0.27	0.27
Blocked Queue	0.03	2.15	4.84	10.89	27.12	74.43
Aborts	0.00	0.0275	0.0513	0.14	0.18	0.30
Response Time	108.67	90.41	86.27	83.77	82.79	80.79

DDG at Class 1 Write Probability=0.4

Table 5.5: Break up of Class 1 response time

different stages of execution. We can see that even though a transaction running under the DDG policy has a higher running overhead, it spends less time in the blocked queue as compared to a transaction running under 2PL. The net effect is that the DDG policy leads to better response time. In Table 5.5, we show the corresponding numbers for the Class 1 transactions. Here, we can see that the Class 1 transactions have a higher running overhead and spend higher time in the blocked queue as compared to the transactions running under the 2PL. The reason for greater time spent in the blocked queue is because Class 1 transactions need to lock more items than they actually need. As a result of these factors, there is a slight degradation in the Class 1 response time.

These results confirm that the advantage of releasing locks before commit time can be better realized if the Class 1 transactions perform some updates in which case the DDG policy reduces the Class 2 response time without adding an unacceptable overhead for Class 1. Since, in the APACS application, it is crucial to improve the response time of Class 2 transactions, this behavior gives the DDG policy an advantage over 2PL.

Comparison to 2PL with 2° Isolation

Many commercial systems achieve greater concurrency by relaxing the serializability requirement. Serializability is termed as 3° isolation. In a 2° isolation system, read locks are released as soon as the entity has been read, whereas, write locks are held until the

MPL	Write Probability = 0.0		Write Probability = 0.2		Write Probability = 0.4	
	2PL	DDG	2PL	DDG	2PL	DDG
	Class 1 Response Time					
1	70.53	77.21	85.00	94.55	100.71	108.67
2	48.03	59.94	70.57	79.32	84.37	90.42
3	40.62	54.94	64.45	74.73	80.97	86.27
5	32.28	49.85	62.41	72.01	75.46	83.78
10	27.41	46.66	62.13	71.30	73.39	82.80
25	24.95	44.91	59.96	70.49	76.09	80.80
	Class 2 Response Time					
1	0.06	1.13	0.59	1.01	1.28	0.86
2	0.08	2.03	1.09	1.88	2.20	1.51
3	0.09	2.78	1.49	2.51	3.05	2.18
5	0.12	3.94	2.27	3.65	4.18	3.21
10	0.20	6.06	3.99	5.84	6.59	5.58
25	0.43	9.79	8.33	11.24	12.47	11.78

The write probability in the above table corresponds to Class 1 transactions.

Table 5.6: Comparison to 2PL with 2° isolation

transaction commits (Gray and Reuter 1993). This means that some of the schedules in such a system will be incorrect. Thus, even though the comparison to the DDG policy which always produces correct schedules is not fair, we consider this case to satisfy our curiosity. In Figure 5.6, we compare the Class 1 and Class 2 response times obtained by using 2PL with 2° isolation and the DDG policy with 3° isolation.

We can see that at write probability 0.0, 2PL with 2° isolation leads to considerably lower response time for both Class 1 and Class 2 transactions. This is because, Class 1 locks are held only for a brief period of time. At write probability 0.2, the response times given by the two algorithms are closer to each other as compared to the corresponding values at write probability 0.0. At write probability 0.4, the Class 2 response times given by the DDG policy are clearly better than the corresponding value given by 2PL with 2° isolation. Thus, we can conclude that in high contention environments, the DDG policy will outperform 2PL and also give a higher degree of isolation in the database.

All Transactions do not have to Follow the Same Graph

The DDG policy will perform the best when all the transactions traverse the knowledge base along the graph that is used for concurrency control. But as seen in the APACS workload, in a typical application, different transaction types may traverse the graph in different ways. The objective of this experiment is to study the sensitivity of the relative behavior of the two policies to this effect.

The *transaction offset* is defined as the percentage of Class 1 transactions that do not traverse along the graph that is used for concurrency control. For example, in the APACS knowledge base, if we choose the union of `isA` and `instanceOf` as the graph that will be used for concurrency control, then 62% (see Table 5.2) of the transactions traverse along this graph and the remaining 38% are along other semantic relationships. In this case, the transaction offset is 38%.

When the transaction offset has a non-zero value, a transaction using the DDG policy may lock many more items than it actually needs, thus causing an increase in data contention. For example, in the graph of Figure 5.1(b), if we use G_1 for concurrency control, then a transaction that accesses nodes 9 and 12 by following the type 2 edges, will need to begin from node 1 and lock nodes 1,2,3,5 and 6 even though it does not need them. The percentage of such transactions in a workload is one way to measure this effect. Another advantage of studying the relative behavior as a function of transaction offset is that the offset can be computed based on the workload information, thus giving a quick way of judging the benefits of the DDG policy.

To analyze the effect described above, we ran experiments for several values of transaction offsets between 0 and 100%. In Figure 5.10, we show the relative improvement in Class 1 and 2 response times for transaction offsets between 0 and 100%. We can see that even for high values of offsets such as 50%, the DDG policy is able to give better response time for Class 2 transactions without seriously slowing down Class 1 transactions. But when the transaction offset approaches 100%, the performance of the DDG policy degrades. The degradation is due to excessive blocking caused by the skewed access patterns. The DDG policy is able to deal with high values of transaction offsets, because the extra nodes that need to be locked to enforce the rules of the locking policy are locked in shared mode. Since accesses on such nodes are read accesses, concurrency does not suffer too much.

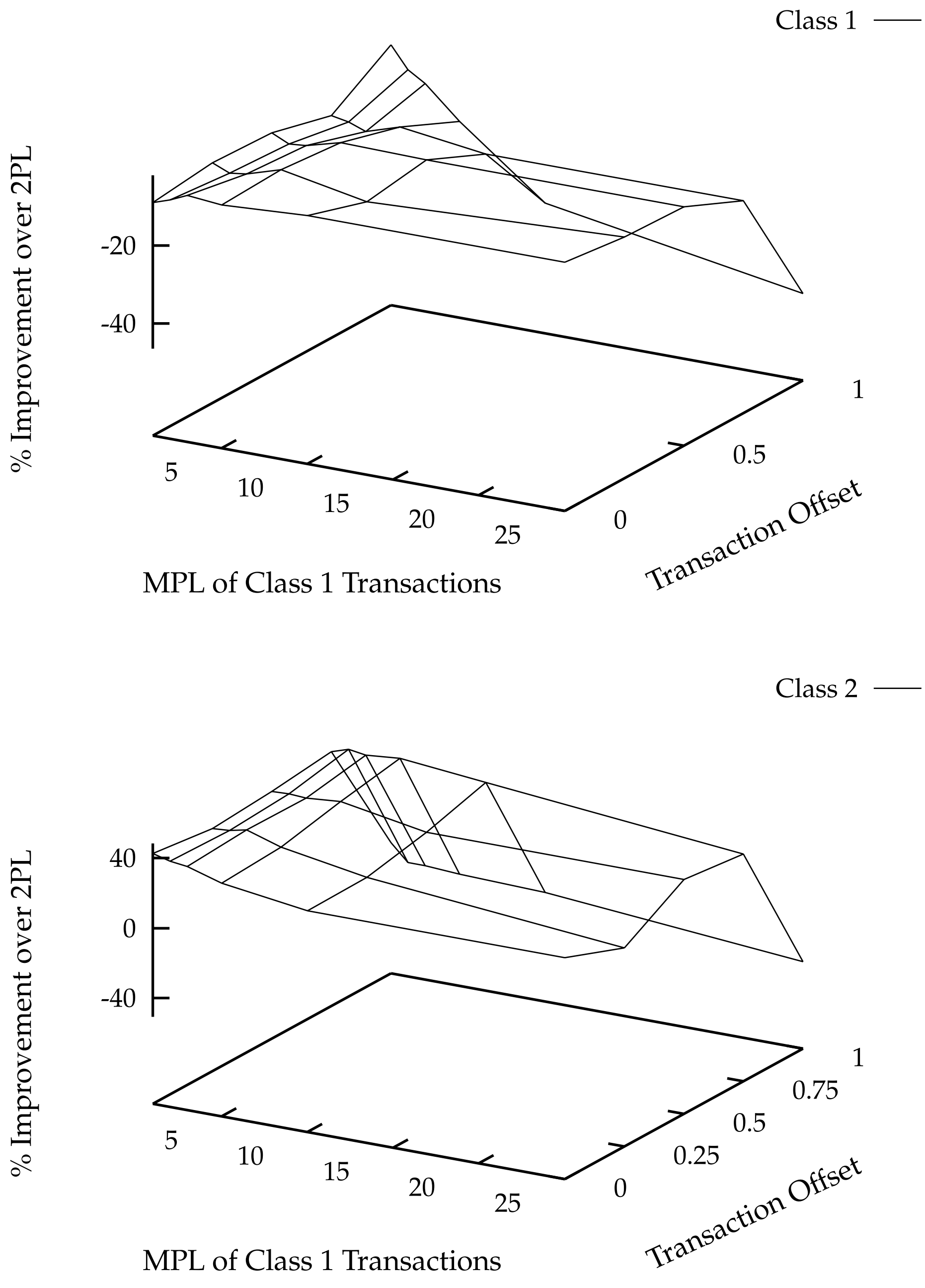
Hence, the DDG policy works better than the 2PL for those situations when most of the transactions follow the graph that is used for concurrency control. This is a desirable property as, in practice, the traversals are distributed along different semantic relationships.

Longer Transactions Lead to Better Improvements

In the baseline APACS workload, the length of transactions in Class 1 varied between three and six levels. Recall that we specify the length of a transaction in terms of the number of levels it accesses. In this section, we study the relative behavior of 2PL and the DDG policy as a function of transaction length. To understand, how the relative behavior is affected by transaction length, consider for example, the graph shown in Figure 5.1(b), and a transaction T_4 that begins at node 2 and accesses nodes 4 and 5 on the next level. Assume that it updates all the nodes that it accesses. A possible execution of such a transaction can be:

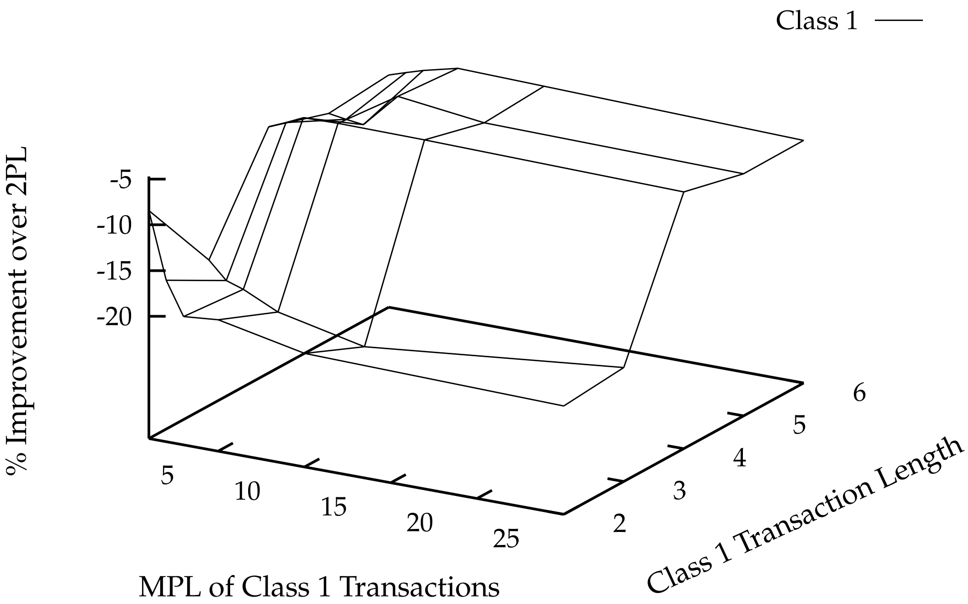
```
(LX 2) (LX 4) (UX 5) (LX 5) (U 5) (U *)
```

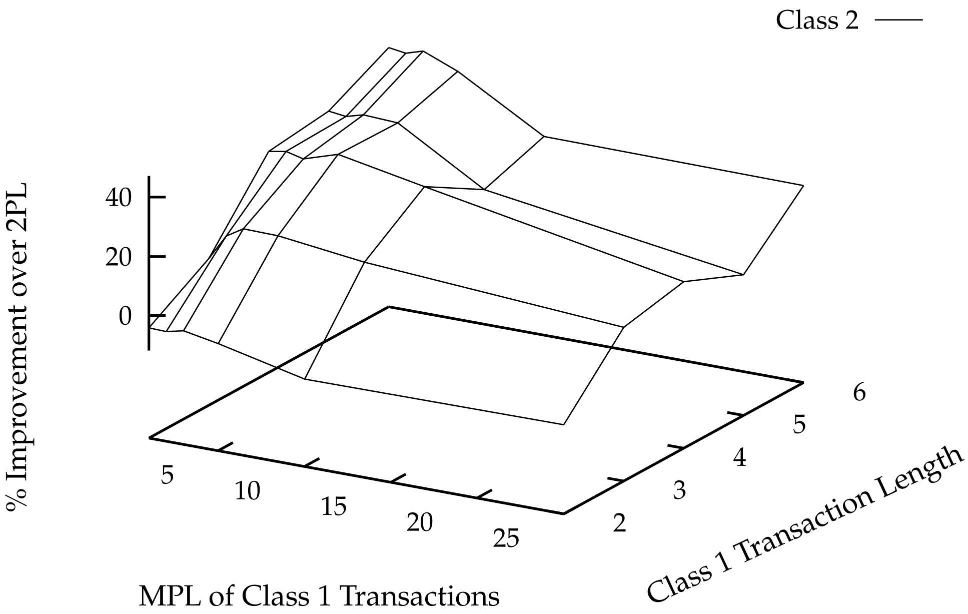
Even though T_4 is able to release a lock before locked point, we expect the benefits of using the DDG policy to be smaller as compared to the benefits in case of transaction T_1 considered earlier, that traversed many levels and released several locks before locked point. To quantify this effect, we fixed the write probability of Class 1 transactions at 0.4 and ran simulations fixing the transaction length to 2, 3, 4, 5 and 6 levels, respectively. In Figure 5.11, we show the relative improvement in Class 1 and 2 transaction response time as a function of Class 1 transaction length and multiprogramming level. From this we can see that when Class 1 transactions access only two levels, there is no improvement between the Class 2 response times. Class 1 response times degrade because the advantage



(Positive improvement indicates that DDG performs better than 2PL.)

Figure 5.10: Effect of transaction offset





(Positive improvement indicates that DDG performs better than 2PL.)

Figure 5.11: Effect of transaction length

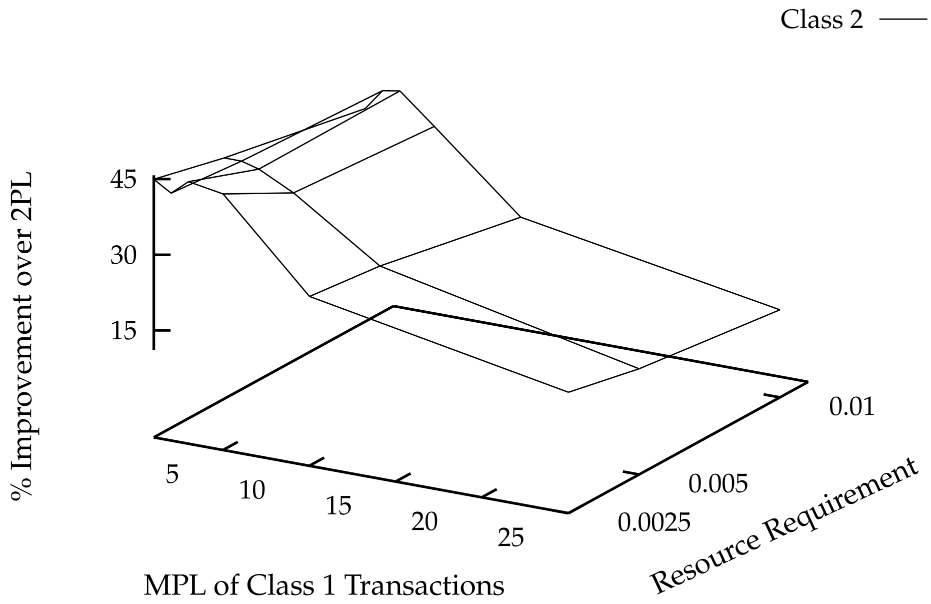


Figure 5.12: Results are robust: effect of resource requirement

of release of locks before locked point is not sufficient to cover the extra overhead. When Class 1 transactions access six levels, the DDG policy improves Class 2 response time by about 50% whereas the degradation in Class 1 response time is much lesser. This means that the DDG policy is more suitable when the transactions traverse several levels in the graph.

5.1.6 Robustness of Results

In this section, we report the robustness of our results to the change in values of different system parameters and assumptions. In most cases, we show improvement in response time only for Class 2 transactions, because improvements in Class 1 response times were insignificant. In all cases, we assume that the Class 1 write probability is 0.2.

Resource Requirement

In the baseline APACS workload, the processing requirement for each operation was set to 5 milliseconds. We re-ran the experiment of Section 5.1.5 by changing the resource requirement to 2.5 milliseconds and 10 milliseconds. The relative improvements in response time for Class 2 transactions are shown in Figure 5.12. We can see that the improvements are maintained across a range of processing requirements.

Buffer Hit Ratio

The *buffer hit ratio* defines how many times a desired data item can be found in the cache of the KBMS. By varying this parameter, we were able to model situations ranging from the whole knowledge base fitting in main memory to the whole knowledge base residing on secondary storage. Again, we re-ran the experiments of Section 5.1.5 with a hit ratio of 0.5 and 1.0. The relative improvements in Class 2 response are shown in Figure 5.13.

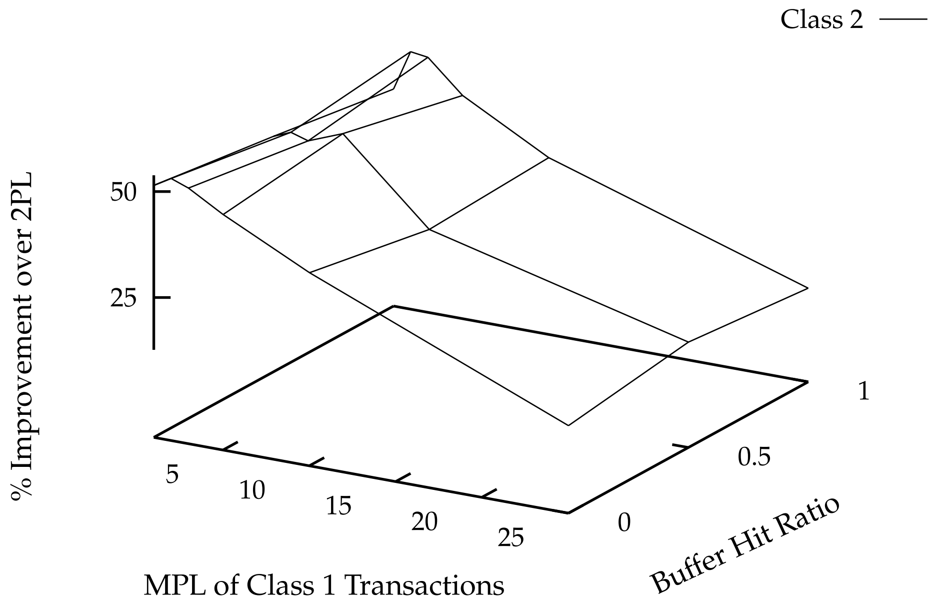


Figure 5.13: Results are robust: effect of buffer hit ratio

At high buffer hit ratio, the absolute values of response time are lower as compared to the corresponding values of response time at low buffer hit ratio (the actual values are not shown here). However, we can see that the relative improvement in the Class 2 transaction response time is maintained across different values of the buffer hit ratio. These results show that by changing the buffer hit ratio the absolute values of the response time given by the two policies are affected in the same way and that the DDG policy will lead to improvement in Class 2 response times regardless of whether the knowledge base is stored on disk or in main memory.

Class 1 Think Time

Class 1 think time defines the rate of arrival of Class 1 transactions. In the baseline experiment, think time was set to 1 second. We varied its value by setting it to 50, 100 and 200 seconds. These values were chosen to set think time to less than, greater than, and much greater than Class 1 response time as obtained in the baseline case. The relative improvements between the response times under the 2PL and the DDG policies for Class 2 are shown in Figure 5.14. The main effect of the variation in think time is that, at higher think times, the improvement in Class 2 response time gets better. This makes intuitive sense, because as the think time increases, fewer of Class 1 transactions are active (which corresponds to lower levels of multiprogramming), and as a result, the improvement in Class 2 response is much better

Class 2 Arrival Rate

In the baseline experiment, the Class 2 arrival rate was set to 25 transactions per second. This arrival rate corresponded to the situation in the APACS when there is a fault in the plant and several alarms are triggered at the same time. Under normal conditions, the arrival rate of Class 1 transactions is 5 transactions per second. To see the sensitivity of

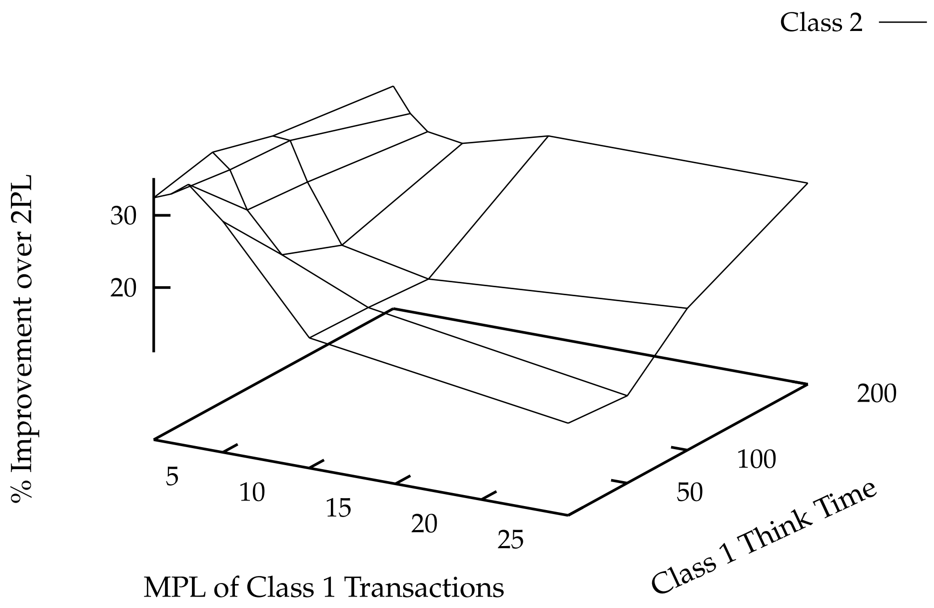


Figure 5.14: Effect of Class 1 think time

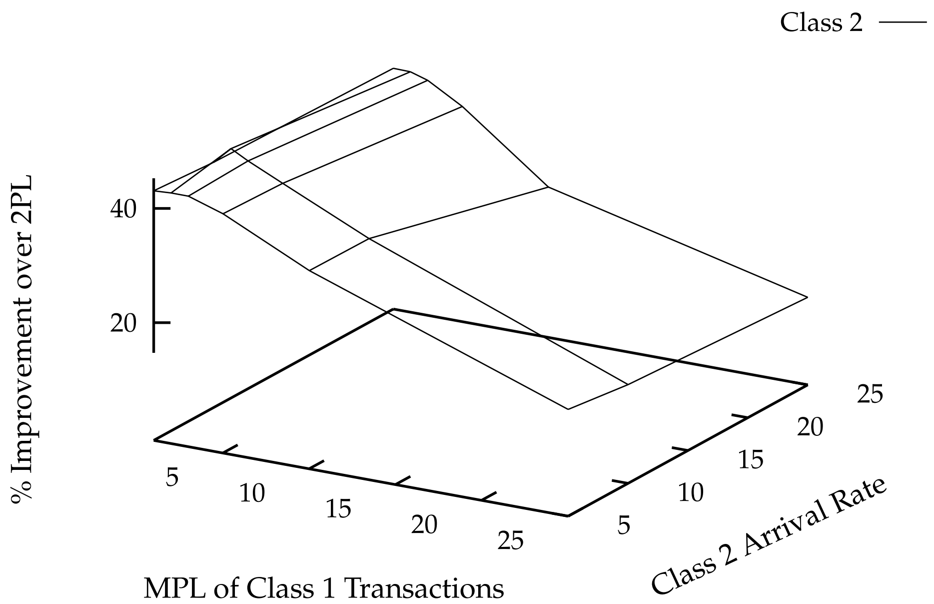


Figure 5.15: Effect of Class 2 arrival rate

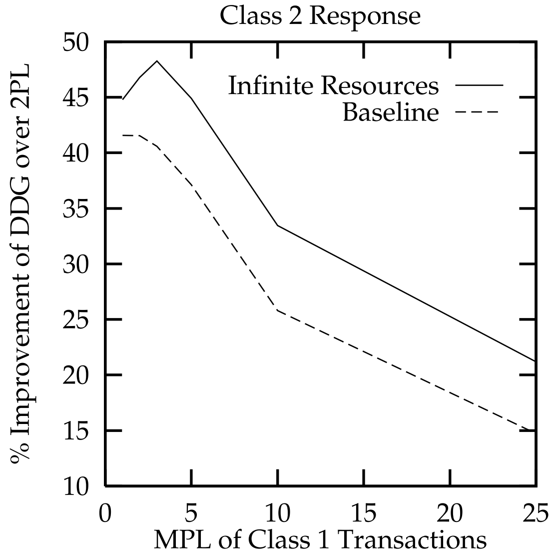


Figure 5.16: Effect of resource availability

our results to the value of Class 2 arrival rate, we re-ran the baseline experiment by setting the Class 2 arrival rate to 5 transactions per second and 10 transactions per second. The relative improvements between the response times under the 2PL and DDG policies for Class 2 transactions are shown in Figure 5.15. From these results, we can see that the relative behavior of the two policies as seen in the baseline case is maintained even when the arrival rate changes. From the present and the previous experiment, we can conclude that the results are more sensitive to a change in Class 1 think time than to change in Class 2 arrival rate.

Resource Availability

Resource availability defines the CPU and disk resources used to implement the KBMS. To study the bounding case, we re-ran the baseline case with the assumption of infinite resources, that is, assuming no queuing at the CPU or the disks. In Figure 5.16, we show the percentage improvement of the DDG policy over 2PL, for the cases when there are infinite resources and when there are limited resources. We can see that when there are infinite resources, the improvement in Class 2 response times is much better. We do not show the Class 1 response times because the Class 1 response time given by the two policies were not significantly different. Overall these results suggest that if resources are plentiful, the DDG policy will tend to yield a somewhat greater advantage over 2PL.

Sensitivity to Concurrency Control Overheads

We measured the concurrency control overheads from the implementation done as part of the simulation. These measurements are inherently error prone. Specifically, they are underestimate of the true values, because they do not include the cost of acquiring the semaphores on the shared data structures (for example, lock table). After studying our implementation, we determined that the cost of acquiring semaphores would be approx-

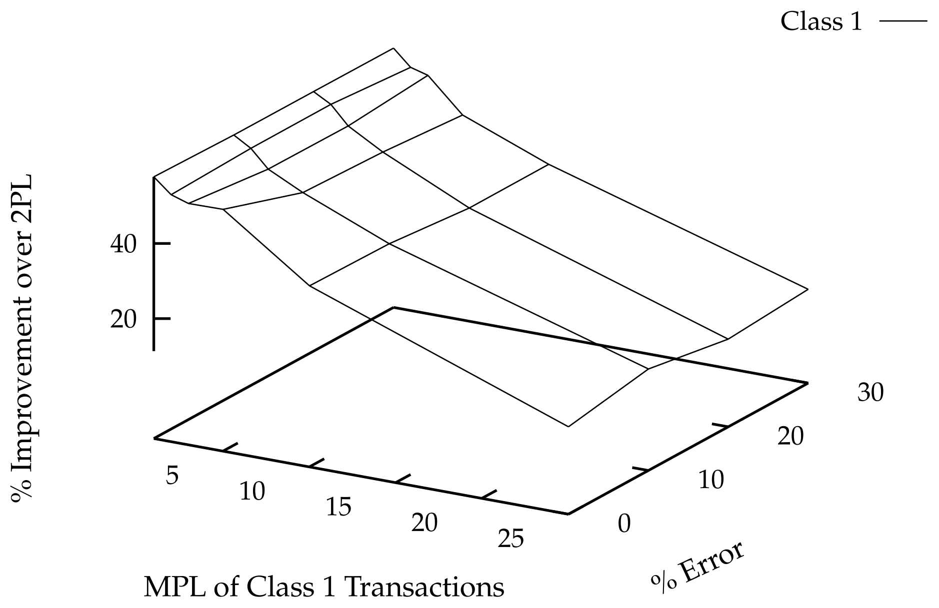


Figure 5.17: Sensitivity to concurrency control overheads

imately 10% of the total cost. Therefore, to study the sensitivity of our results to error in these measurement, we repeated the baseline experiment by increasing the value of concurrency control overheads by 10%, 20% and 30% for various values of Class 1 write probability. The percentage improvements in Class 2 response time for write probability 0.4 are shown in Figure 5.17. We can see that the improvements seem almost unaffected for 10-30% error range in the concurrency control overheads. This means that the results are robust to measurement errors in the range 10-30%.

Rate of Change in Structure

In the baseline experiment, 0.1% of Class 2 transactions changed the semantic relationships (or inserted/deleted nodes and edges) in the knowledge base. To study the sensitivity of our results to change in this parameter, we re-ran the baseline experiment by setting the value of this parameter to 0.5% and 1.0%. We did not try higher values, because changes in the semantic relationships are infrequent. The percentage improvement between the response times under the DDG and 2PL policies for both classes is shown in Figure 5.18. We can see that the relative behavior of the two policies remains the same in the range (0.1% − 1.0%) of variation of this parameter. The results do not significantly change because the relative percentage of the updates is small and the updates are local. Based on the above results, we can conclude that if the changes in the graph are infrequent the relative behavior of the two policies will be unaffected.

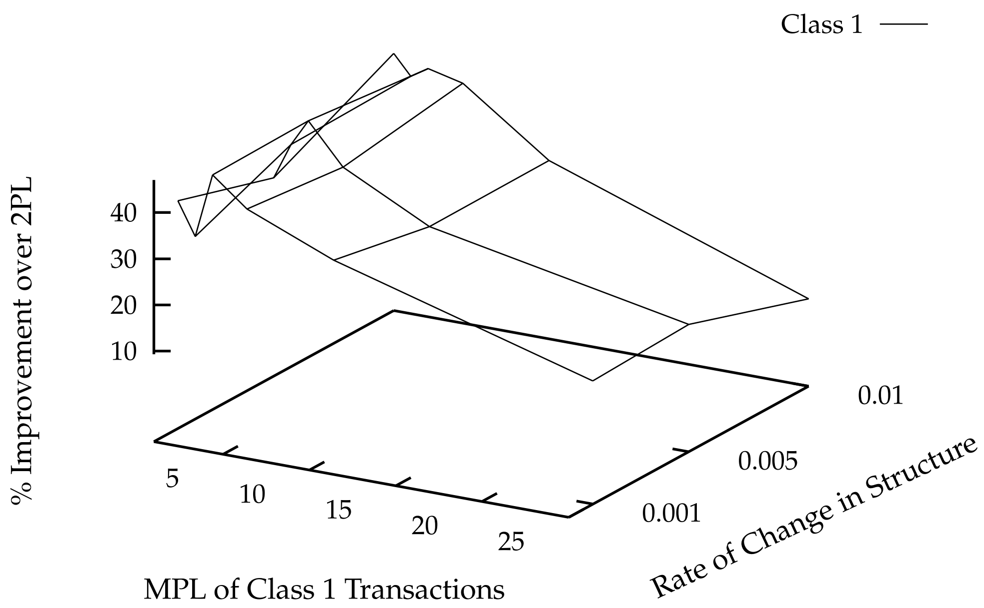


Figure 5.18: Effect of rate in change in structure

Effect of Model Configuration

In the APACS workload, it is clear that Class 1 transactions should be represented as a closed model and Class 2 transactions as an open model. Such a choice is natural because the arrival rate of Class 1 transactions depends on the number of transactions that are already active in the system, whereas for Class 2 transactions the arrival rate is independent of the number of transactions that are already active. But to satisfy our curiosity, we wanted to check if the results are sensitive to the assumptions about the model configuration. Therefore, we ran experiments by changing the model configuration so that the class throughputs are approximately the same. The percentage improvement in response time for the two classes of transactions are shown in Figure 5.19. In Figure 5.19, Open-Open represents the situation when both Class 1 and Class 2 transactions are represented as an open model. Other labels should be interpreted in a similar fashion. We can see that the improvements obtained by using the DDG policy as compared to 2PL degrade when Class 1 transactions are represented by an open model.

The above result can be explained by considering the mean queue length in a queuing network. In general, mean queue length for an open system is higher than the mean queue length of a closed system with the same utilization (Zahorjan 1983). As a result, for two systems with the same throughput, one of which is open and the other closed, the response time given by an open system is higher than the response time given by a closed system. Therefore, by representing the Class 1 transactions as an open system, we get higher response time, so much so, that this shifts the relative behavior of the two algorithms.

In practice, workload representation needs to be resolved by looking at the real system and deciding which model is a more accurate representation. From our discussions with the designers of the APACS (Wang 1993), it appeared that Class 1 transactions are better represented by a closed system, and therefore, the relative behavior as reported in the baseline case in Section 5.1.5 is realistic.

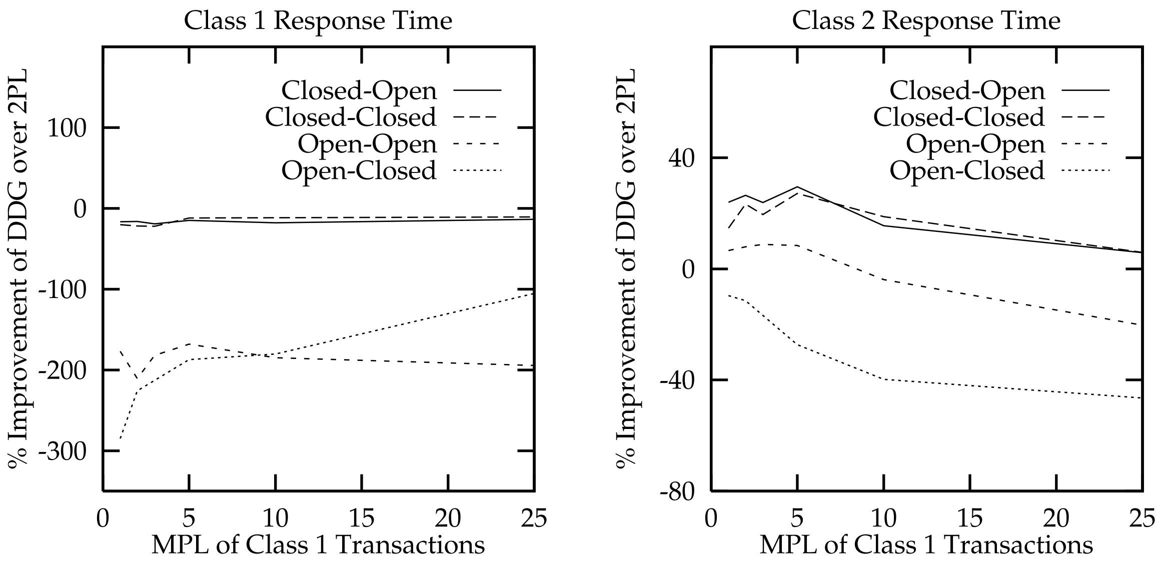


Figure 5.19: Effect of model configuration

5.1.7 Summary of Experimental Results

We can summarize the results of the previous subsections as follows:

- A knowledge base may contain several semantic relationships. The choice of semantic relationship to be used by the concurrency control algorithm depends on the transaction mix and the implementation used. Using a semantic relationship that is traversed by a transaction helps the concurrency control manager to determine the entities that might be accessed by that transaction, but may require other transactions, that do not follow this relationship, to lock more nodes than they would otherwise. In the APACS knowledge base, the graph defined by isA and instanceOf relationships was found to be the best choice, indicating that more semantic information may not be always useful.

- For APACS workload, depth-first traversal leads to a better response time as compared to the breadth-first traversal.

- At heavy data contention, the DDG policy improves the response time of Class 2 transactions by about 50%. In other words, it can cut the Class 2 response time by about one half. This behavior is maintained even if some of the transactions do not follow the graph that is used for concurrency control. The improvements are better if the transactions traverse several levels in the knowledge base.

- In comparison to 2PL with 2° isolation, the DDG still outperforms 2PL at high data contention. But for this effect to be observed, the write probability has to be quite high. This makes the DDG policy quite attractive for high contention environments, because not only can it give better response time, it also maintains isolation of the database. When the contention is very low, 2PL with 2° isolation gives much better performance, but at the cost of a potentially inconsistent database.

Properties of Subsumption Relationship	
Number of Nodes	1015
Number of Roots	1
Number of Leafs	891
Fanin	
Maximum	2
Average	1.09
Fanout	
Maximum	245
Average	8.5
Depth	
Maximum	10
Average	5.09
Cycles	None

Table 5.7: Structural properties of semantic relationships in the Wines KB

- The results are robust to changes in the values of the resource requirements, buffer hit ratio and rate of update to the structure. They are, however, dependent on think time, resource availability and model configuration for Class 1 transactions. At lower think times, the improvement is Class 2 response time is less pronounced. If the resources are plentiful, the improvements in the response time by using the DDG policy are better. If an open model is assumed for Class 1 transactions, the degradation in Class 1 response time using the DDG policy is worse and the improvement in Class 2 response time is less.

5.2 Case Study 2: Wines Knowledge Base

In this section, we compare 2PL and the DDG policy for a knowledge base that represents information about wines and meal courses (Brachman et al. 1990). In this section, we first describe this knowledge base and then discuss our experimental results.

5.2.1 Overview of the Wines Knowledge Base

The wines knowledge base has been developed at AT&T Bell Labs, New Jersey (Brachman et al. 1990). This is a small system (approximately 400 objects) that is used for demonstration purposes. We took this knowledge base and extended it by adding several objects so that the final size of the knowledge base was about 1015 objects.

The objects represented in the knowledge base are related to each other by subsumption relationships. There is a subsumption relationship between two objects A and B, if the attributes of object A are a subset of the attributes of object B (Borgida et al. 1989). Thus, each object can be represented by a node and each subsumption relationship by an edge. Using this approach, we constructed a graph corresponding to the Wines knowledge base. The structural properties of the knowledge base are shown in Table 5.7. The graph corresponding to this knowledge base is not restricted to a tree but is, however, acyclic.

As in the APACS knowledge base, the transactions on the Wines knowledge base can be classified into two categories. Class 1 transactions are long and involve adding an object definition to the knowledge base. Such a transaction begins at the root of the knowledge base and traverses it until it finds the right location for the object definition and then adds it there. Such transactions perform a traversal of the knowledge base. Class 2 transactions are short. They involve updates on the attribute values, lookup of concept definitions, or changing of concept descriptions.

Since Wines knowledge base is not a production knowledge base, there is hardly any information available about the proportion of the transactions in the two classes. Therefore, we made assumptions about the proportion of the transactions in the two classes. In particular, we represented the workload with a closed model assuming 50 terminals active at any given time, each having a think times drawn from an exponential distribution with mean 10 seconds. Half of the transactions are in Class 1 and the other half are in Class 2. We fixed the write probability of Class 2 transactions to 0.5 and varied the write probability of Class 1 transactions. The Class 1 transactions traversed from three to six levels in the knowledge base leading to an transactions which on an average accessed 85 objects. The rest of the parameters were kept at the same values as in the APACS workload.

5.2.2 Results for the Wines Knowledge Base

We report results only for the baseline experiment, where we vary the write probability and the multiprogramming level of Class 1 transactions while keeping other parameters constant. The improvements obtained by using the DDG policy as compared to 2PL policy are shown in Figure 5.20. For Class 1 transactions, at low values of write probability, there is some degradation in response time from using the DDG policy rather than 2PL. But as the write probability increases, there is less degradation, and in fact, the response times given by the DDG policy are comparable to 2PL when the write probability is higher. For Class 2 transactions, there is a consistent improvement in response time when the multiprogramming level is between 1 and 10. The improvement can be as high as 80% as the write probability increases.

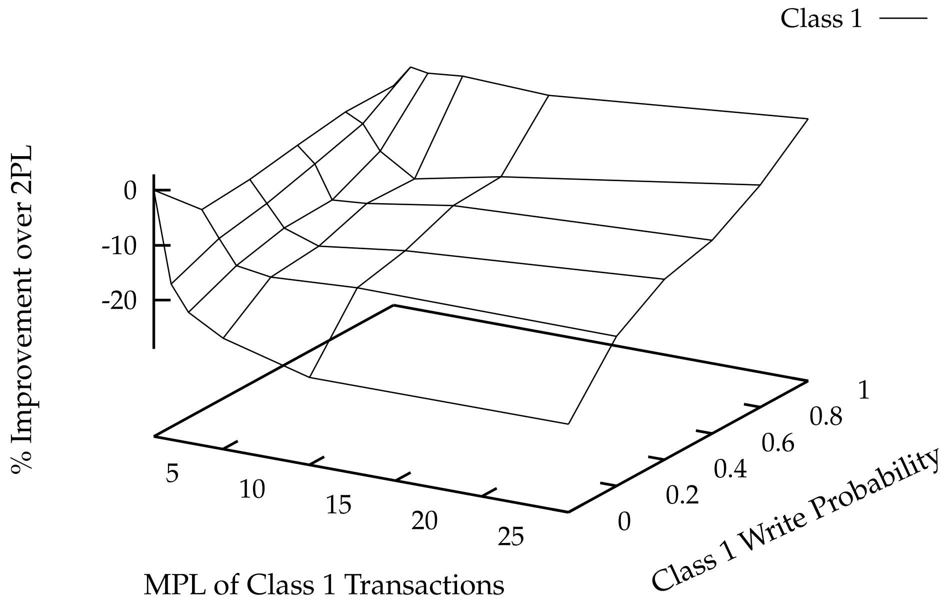
The behavior of Class 1 transactions can be explained by observing that, at write probability 0.0, it is not possible to release any locks before commit time. Therefore, the cost of the DDG policy is not offset by the benefits of increased concurrency. As the write probability increases, however, the benefits of increased concurrency obtained by pre-release of locks offsets the higher running cost of the DDG policy. The behavior of Class 2 transactions can be explained in a similar way.

5.3 Case Study 3: Toronto Virtual Enterprise Knowledge Base

The third knowledge base that we considered was from the Toronto Virtual Enterprise (TOVE) project. We first give a brief description of this knowledge base and then present the experimental results.

5.3.1 Overview of the TOVE Knowledge Base

The TOVE knowledge base is being developed as part of the enterprise integration project in the Department of Industrial Engineering at the University of Toronto (Fox, Chionglo and



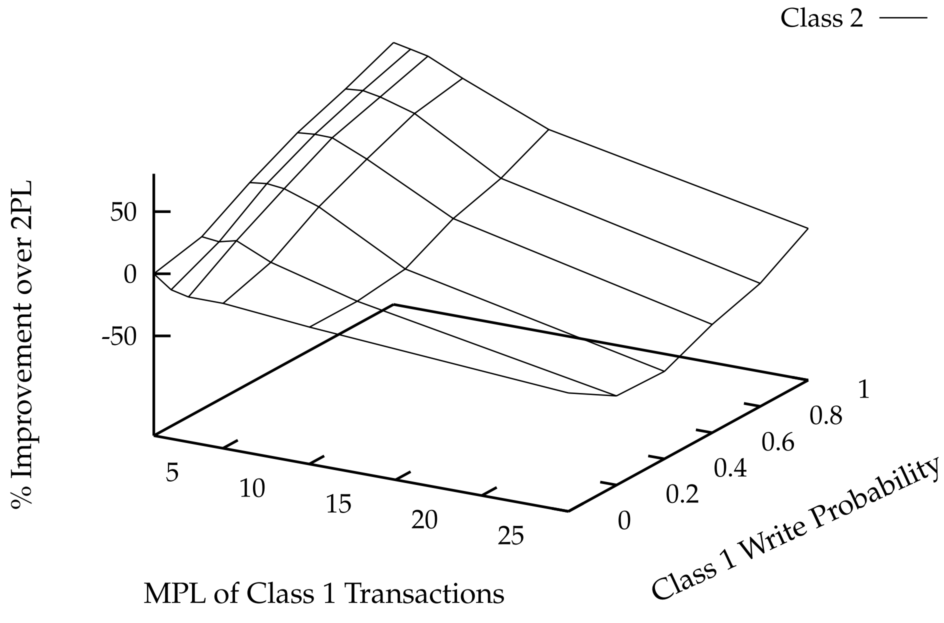


Figure 5.20: Wines knowledge base — baseline experiment

Semantic Relationship	Number of Nodes/ Roots/Leaves	Fan In Max/Avg	Fan Out Max/Avg	Depth Max/Avg
`isA`	1104/7/1039	1/1	132/16.87	7/5
`instanceOf`	224/48/176	1/1	43/3.66	2/2
`conjuncts`	437/132/305	1/1	1/1	2/2
`disjuncts`	264/132/132	1/1	1/1	2/2
`consumes`	191/107/84	4/1.27	1/1	2/2
`produced`	182/91/91	1/1	1/1	2/2
`enabled_by`	180/90/90	1/1	1/1	2/2
`partOf`	91/7/57	4/1.23	6/3.14	4.01/6

Table 5.8: Structural properties of semantic relationships in the TOVE KB

Fadel 1993). The goal of enterprise integration is to support communication of information and the coordination and optimization of enterprise decisions and processes in order to achieve higher levels of productivity, flexibility and quality.

The TOVE KB stores information about an enterprise, for example knowledge about design, manufacturing, marketing and field services. The information is represented by means of activities, states, resources and parts. The objects in the knowledge base are organized by means of classes which are related to each other by `isA` relationships. A class may have several instances. In addition, objects in the KB can be related to each other by several semantic relationships. For example, a state *A* may have a `Conjuncts` relationship with states *B* and *C* which would mean that *A* is true if and only if *B* and *C* are true.

The structural properties of the Tove knowledge base are shown in Table 5.8. The `isA` relationship is a tree. The `partOf` relationship is an acyclic graph. All other relationships are shallow in the sense that the maximum depth is two levels.

5.3.2 Results for the TOVE Knowledge Base

We assumed the same workload for this knowledge base as in the Wines knowledge base. We assumed two classes of transactions. Class 1 transactions perform traversal and Class 2 transactions perform lookup and update. Again, Class 1 transactions traversed from three to six levels in the knowledge base leading to transactions which on an average accessed 110 objects. We assumed a closed model with 50 terminals each having a think time of 10 seconds. We assumed that the number of transactions in the two classes is equal. We fixed the write probability of Class 2 transactions to 0.5 and varied the Class 1 write probability between 0.0 and 1.0. The results are shown in Figure 5.21. Here again, we can see that, at low write probabilities, there is slight degradation in Class 1 response time, but Class 2 response times show consistent improvement as the write probability increases. In fact the improvements can be of the order of 50-60% as the write probability is above 0.4. This shows that the relative behavior of the two policies is maintained for different knowledge bases.

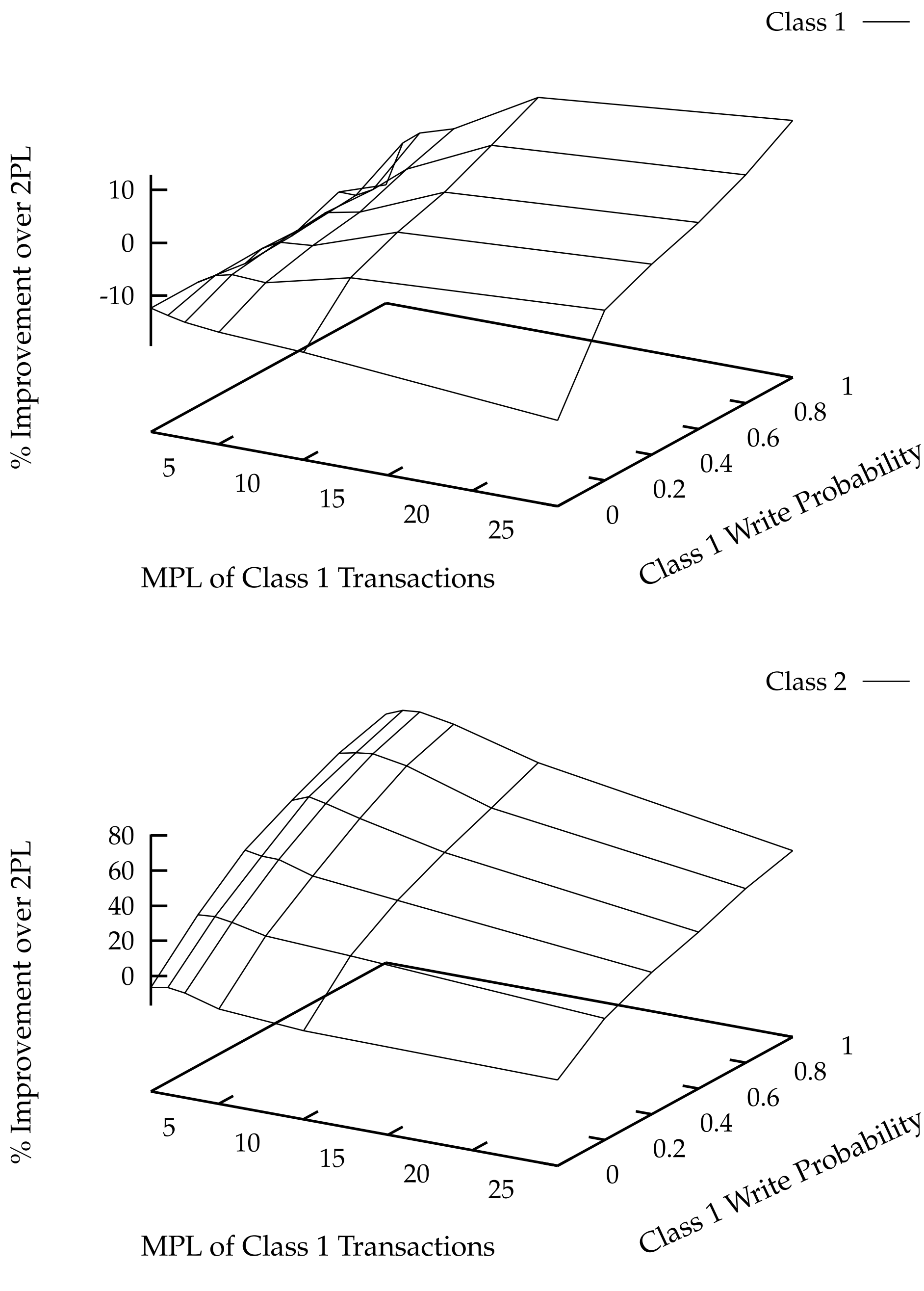


Figure 5.21: TOVE knowledge base — baseline experiment

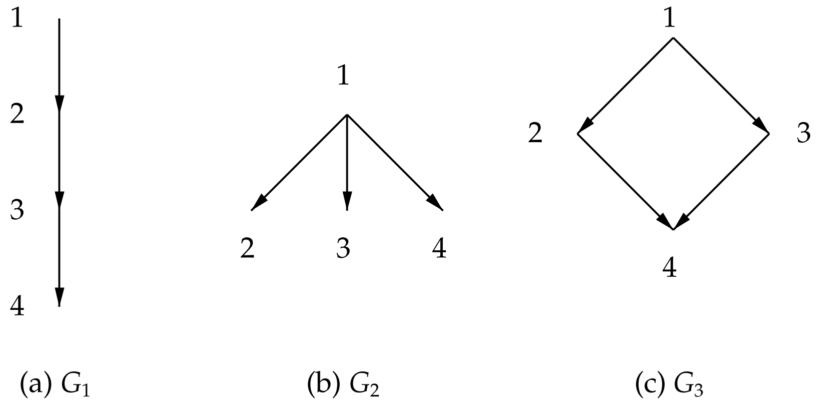


Figure 5.22: Distinguishing the structures in graphs

5.4 A Straw Analysis to Estimate the Relative Behavior

In the previous three sections, we considered three case studies and showed that, at high values of write probability, the DDG policy considerably speeds up Class 2 transactions. The relative improvement in the three cases was different. In this section, our objective is to understand these differences and see if we can identify some simple measures by which we can predict the relative behavior of the two policies.

In Figure 5.23, we show the relative improvement in response times given by the DDG and 2PL policies for the three knowledge bases. These results correspond to write probability 0.4. In this experiment, we used the same workload for the three knowledge bases and also ensured that the transaction lengths are the same. Specifically, we assumed 25 terminals with 0 think time for Class 1 and an arrival rate of 25 transactions per second for Class 2 transactions. This means that the improvements are entirely due to the knowledge base structure. We can see that for Class 1 transactions, the degradation is the least for APACS, followed by TOVE and then Wines knowledge base. For Class 2 transactions, the improvements are the least for the Wines knowledge base. At low multiprogramming levels, the improvements are the highest for the APACS knowledge base, and at high multiprogramming levels, the improvements are slightly higher for the TOVE knowledge base. In fact, the difference between TOVE and APACS at high multiprogramming levels is not significant.

From the above results we can order the relative behavior of 2PL and DDG policies on three knowledge bases as: Wines < TOVE < APACS. We now attempt to explain this difference in terms of knowledge base structure.

To motivate how these differences could be explained, consider the directed graphs shown in Figure 5.22, which have same number of nodes. Suppose, that the transactions perform traversal along a graph and lock all entities in exclusive mode. In case of G_1, once a node has been processed by a transaction, lock on it node has to be held only until we have locked its (only) one successor. In case of G_2, lock on node 1 has to be held until we have locked all three successors (provided the transaction needs to access all three of them). Thus, lock holding time in case of G_1 is likely to be lower as compared to lock holding time in case of G_2. The difference can be captured by measuring the fanout of the graph. Similarly, to lock a node, which is not the first node locked by a transaction, we need to lock only one other node in G_1 and G_2, whereas in G_3, to lock node 4, we need to

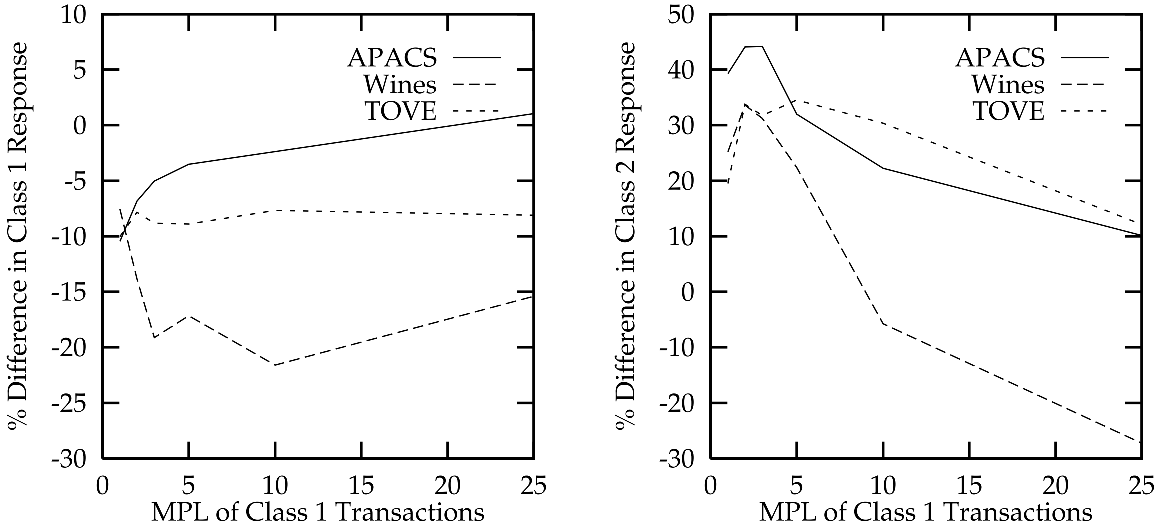


Figure 5.23: Comparison of relative behavior on three knowledge bases

lock both nodes 2 and 3. In this respect, the structure of graph G_3 is less desirable than G_1 or G_2. With this background, we describe the empirical measures that we have devised to capture the above differences.

5.4.1 Measuring Fanout

We experimented with two metrics to measure fanout: average fanout and the ratio of maximum fanout to the knowledge base size. We found that the average fanout does not give enough discrimination between the knowledge bases. In contrast, the ratio of maximum fanout to knowledge base size called *fanout factor*, provides more discrimination. In general, the smaller the value of this factor, the more likely it is that the DDG policy will perform better. For example, the values of fanout factors for the APACS, TOVE and Wines knowledge bases are respectively, 0.09, 0.103 and 0.24. This orders the knowledge base structures in the same order as the percentage improvement in Class 2 response time obtained by using the DDG policy over 2PL.

5.4.2 Measuring Fanin

As in the case of fanout, we experimented with two metrics to measure the fanin: average fanin and the ratio of maximum fanin to the knowledge base size. Here again, the *fanin factor*, that is, the ratio of the maximum fanin to the knowledge base size, provides another distinction between the three knowledge bases. The smaller the value of this factor, the more likely it is that the DDG policy will perform better. For example, the value of fanin ratio for the APACS, TOVE and Wines knowledge bases are respectively, 0.0003, 0.0004 and 0.001. This orders the knowledge base structures in the same order as the percentage improvement in Class 2 response time obtained by using the DDG policy over 2PL.

5.4.3 Measuring Structuredness

In a tightly structured knowledge base, the number of locks necessary to enforce the rules of the locking policy is almost equal to the number of locks needed by a transaction. We experimented with two metrics to measure the structuredness. The first metric, called *depth factor of dominator tree*, $df(G)$, measures the relative structure of the knowledge base, G, and its dominator tree, T. The second metric, called *lock magnification* measures the number of locks that will be acquired for each node if it were not the first node accessed by the transaction. We now define these metrics rigorously.

Let G be the graph corresponding to the knowledge base and T be its dominator tree. Let $d(A, G)$ denote the length of the shortest path from the root of G to A. Then the depth factor of dominator tree can be defined as follows:

$$df = \left[1 - \frac{\sum_{A \in G} (d(A,G) - d(A,T))}{\sum_{A \in G} d(A,G)}\right]$$

The numerator sums up for each node the difference in its depth in a graph G and in the dominator tree of G and it is normalized by dividing it by the sum of the depth of all the nodes in the graph. For a tree, the numerator is equal to zero, giving a depth factor equal to 1, and thus, a tree is tightly structured. The smaller the value of this metric, the less structured is the knowledge base. For example, the value of the dominator factor for the APACS, TOVE and Wines knowledge bases are respectively, 1.0, 0.975 and 0.887. This orders the knowledge bases in the same order as the percentage improvement in Class 2 response time obtained by using the DDG policy over 2PL.

Let $L_1(A)$ (and $L_{\neq 1}(A)$) denote the number of nodes that need to be locked to satisfy the locking rules in order to lock A given that A is (is not) the first item locked by a transaction. Then lock magnification, $lm(G)$ is defined as:

$$lm(G) = \frac{\sum_{v \in G} L_{\neq 1}(v)}{\sum_{A \in G} L_1(A)} = \frac{\sum_{A \in G} L_{\neq 1}(A)}{\mid G \mid}$$

We compute $L_{\neq 1}(A)$ as the number of descendants of the dominator of A that are ancestors of A. In an ideal situation, the value of lock magnification would be equal to 1.0. The closer the value is to 1.0, the better will be the relative performance of the DDG policy. For example, the value of lock magnification for the APACS, TOVE and Wines knowledge bases are respectively, 1.0, 1.39 and 1.29. This metric shows that DDG policy will work better for the APACS knowledge base as compared to the TOVE and Wines knowledge bases. It also suggests that it should work better on Wines knowledge base as compared to the TOVE knowledge base, which is contrary to what we observed. However, if we view this along with other metrics considered above, we could argue that in the overall behavior, the effect captured by lock magnification does not predominate. Furthermore, while computing lock magnification, we have assumed that to lock A, a transaction also locks all the descendants of the dominator of A that are ancestors of A. A traversal transaction will usually contain more than one node and some of the ancestors might be locked anyway because a transaction needs to access them. Thus, this metric will not be always accurate.

5.5 Pragmatic Concerns

In this section, we critically assess some of the results presented in the present chapter. We first comment on the relatively small size of the knowledge bases that we considered and then discuss how these results would be influenced by recovery considerations. We conclude this section by discussing how both 2PL and the DDG policy could be used in the same system to give a hybrid algorithm that switches between the two policies depending on the load conditions.

5.5.1 Scaling up the Results to Larger Knowledge Bases

Consider two databases KB_1 and KB_2. KB_1 has a size of D entities and has N active transactions. KB_2 has a size bD entities and bN active transactions, where $b > 0$. It has been previously shown (Tay, Goodman and Suri 1985; Tay, Suri and Goodman 1985) that for two-phase locking, the probability of conflict is the same in KB_1 and KB_2, and the throughput of KB_2 is b times the throughput of KB_1. The above result can be used for scaling up the results from a small database to a larger database. For example, the results from a database of size 3000 entities and 50 transactions can be considered representative of the results of a database with identical values for all the parameters except that it has 30000 entities and 500 transactions ($b = 10$).

The result mentioned in the previous paragraph was shown only for two-phase locking and when the access patterns are uniform, that is, there is an equal probability of access for each item in the database. Therefore, we cannot directly apply it to a database such as ours, in which transactions perform traversals and where an algorithm other than two-phase locking is used. The scaling up of the results for 2PL, however, suggest that results for the DDG policy should scale in a similar fashion.

Another important aspect in scaling up of these results to larger knowledge bases is the run-time cost for incremental maintenance of graph properties, which include, strongly connected components and dominator relationships. The run-time cost of incremental algorithms is highly dependent on the nature of changes in the graph. For example, if the insertions and deletions to the graph cause only local changes in a graph property, the run-time cost is minimal. On the other hand, if the changes are such that they lead to a change in a substantial part of the solution, the computational cost can be excessive. It is difficult to comment on this aspect in the absence of a specific knowledge base and its workload. In general, if the database is highly structured (we gave a formalization of this notion in Section 5.4.3), the locality of changes is almost assured, and the incremental algorithms, and therefore, the results of this thesis will scale up to larger problems.

5.5.2 Recovery Alternatives

When a transaction releases locks before it commits, it can lead to two kinds of problems. First, transactions may read uncommitted values leading to cascading aborts (Bernstein, Hadzilacos and Goodman 1987). Due to the possibility of cascading aborts and the semantics of commit, the commit of a transaction that reads an uncommitted value must be delayed until the transaction that writes the uncommitted value commits. Second, it is not possible to do recovery by just restoring the before images of the entities written by an aborting transaction (Alonso, Agrawal and Abbadi 1994). Because of these problems, an appropriate recovery scheme must be designed and its cost taken into account for any performance comparisons. Let us briefly describe two recovery alternatives and how they might influence the results presented in this chapter.

One approach to support recovery in such a scenario is to use compensating transactions (Garcia-Molina and Salem 1987; Korth, Levy and Silberschatz 1990). In such a scheme, a transaction T_i that reads (or writes) an uncommitted value written by a transaction T_j is allowed to commit without waiting for the commit of T_j. In case T_j aborts, instead of a conventional rollback scheme (Bernstein, Hadzilacos and Goodman 1987), one must execute a *compensating* transaction that wipes out from the database the effects of T_i. The obvious difficulties with this approach are that it may not be always possible to compensate for the effects of a committed transaction, and even if it is possible, it may not be straightforward to design a compensating transaction. The advantage of this approach is that if the cost of compensation is comparable to the cost of rollback, there is no performance degradation. If inexpensive compensation is feasible, the performance results of this chapter directly apply.

Another approach to support recovery when a transaction releases locks before commit time is to delay the commit of some transactions. A transaction T_i that reads uncommitted value written by a transaction T_j is not allowed to commit until T_j commits. One must modify the conventional recovery scheme so that it does not always restore the before images of the items written by an aborting transaction (Alonso et al. 1994; Alonso, Agrawal and Abbadi 1994). The advantage of this approach is that it is much easier to implement as compared to compensating transactions. The disadvantage of this scheme is that due to delayed commit, one may not get full benefit of the non-two-phase behavior of the locking policy. A precise evaluation of a recovery scheme based on delayed commit is beyond the scope of the thesis and is left for future work.

5.5.3 A Hybrid Algorithm

Let us consider a scheme in which the concurrency control manager can switch between 2PL and the DDG policy depending on the load conditions, thus always giving a performance level which is the better of the two algorithms. Since most of our performance results are expressed as a function of write probability, to be able to use our results, such an algorithm would have to monitor the write probability of transactions. Write probability can be easily computed in a system by keeping a running count of the number of all accesses and the number of accesses that involved an update. Such a system would begin by using 2PL as the initial algorithm. If the percentage of updates is above a certain limit (for example, 20% for the APACS workload), the concurrency control manager will switch to the DDG policy. When the write probability reduces from a high value to a low value (less than 20% for the APACS workload), the hybrid algorithm would switch to 2PL. To do this, we need to see how one could switch from one algorithm to the other.

In general, if we allow the transactions locked according to 2PL and the DDG policy to execute simultaneously, it can lead to non-serializable schedules. For example, consider a knowledge base with three nodes 1, 2 and 3. Suppose in the knowledge base graph there is an edge from node 1 to node 2 and an edge from node 2 to node 3. Consider transactions T_1 and T_2 such that T_1 is locked according to 2PL and T_2 is locked according to the DDG policy. Here is a possible interleaving:

```
T1: (LX 1) (LX 2) (U 1)                          (LX 3) (U *)
T2:                        (LX 1) (LX 3) (U *)
```

The interleaving shown above is non-serializable because it is not equivalent to either T_1 followed by T_2 or T_2 followed by T_1. Therefore, we need to introduce a transition phase in which we switch from one algorithm to the other. When we switch from 2PL to the DDG policy, during the transition phase, all transactions that are locked according to the DDG policy should not release any locks before the locked point. Such transactions will, thus, effectively behave as transactions locked according to two-phase locking. The transition phase must last until the last of all the transactions that were running under two-phase locking commit. After this point, all active transactions are running under the DDG policy, and therefore, can release locks before the locked point without jeopardizing serializability. While switching from the DDG policy to 2PL, for any new transaction in the transition phase, we continue to use the DDG policy but do not permit it to release any locks before its locked point, thus satisfying the rules of both the DDG policy and 2PL. The transition phase lasts until the last of the active transactions that had begun before the transition phase commits. After the transition phase, all transactions use 2PL.

5.6 Chapter Summary

In this chapter, we quantitatively evaluated the implementation choices for the DDG policy, and its relative behavior in comparison to two-phase locking. To accomplish this, we used three real knowledge base applications. The workload consisted of a mix of long transactions (Class 1) that performed traversals and short transactions (Class 2) that performed lookup and updates.

The implementation alternatives that we considered included the selection of a subset of relationships in the knowledge base to be used for concurrency control, and the dependence of concurrency on the traversal strategy used to search through the knowledge base. We showed that appropriate choice of the graph to be used for concurrency control can sometimes lead to an orders of magnitude difference in the response time (Figure 5.2). Similarly different traversal strategies led to big differences in the amount of concurrency permitted (Figure 5.6).

The DDG policy led to better response time than 2PL for Class 2 transactions at heavy data contention (Figure 5.9). The improvements were greater when Class 1 transactions traversed several levels (Figure 5.11) and followed the graph that is used for concurrency control (Figure 5.10). We also compared the DDG policy with 2PL with degree 2 isolation and showed that at higher levels of data contention, not only did the DDG policy produce correct schedules but it also led to better performance (Table 5.6). This makes the DDG policy quite attractive for high contention environments.

We studied the robustness of our results to various assumptions about the workload and system resources. We showed that in the range of parameters considered, the improvements in Class 2 response were not substantially influenced by assumptions about the resource requirement to process each request (Figure 5.12), buffer hit ratio (Figure 5.13) and Class 2 arrival rate (Figure 5.15). The improvements were found to be sensitive to resource availability (Figure 5.16), Class 1 think time (Figure 5.14) and the work load representation (Figure 5.19).

The structure of the knowledge base influenced the performance of the DDG policy significantly (Figure 5.23). To obtain a better understanding of this phenomenon, we compared the relative improvements on three different knowledge bases. We presented some empirical results which can be easily computed from the graph and can give a quick estimate of the relative behavior of the two policies. For example, low values for the fanin factor, fanout factor and a high value for the depth factor of the dominator tree imply a good potential for obtaining improvement by using the DDG policy as compared to 2PL.

We also considered a hybrid algorithm that can switch between 2PL and the DDG policy depending on the load conditions. A system using such a hybrid algorithm would almost always work better than a system using only 2PL.

The results of this chapter provide an insight into the relative behavior of the DDG and 2PL policies. The most positive aspect of these results is that they are quantitative in nature: Given an application and its workload, a system designer can use these results to decided whether the improvements offered by the DDG policy are worth the extra effort. We believe that in applications with unbalanced workload (a mix of short and long transaction), where the response time of short transactions is crucial, the DDG policy has potential of being a viable solution.

Chapter 6

Conclusions

In this chapter we summarize the contributions of this thesis and identify some open problems.

6.1 Contributions of the Thesis

The primary contribution of this thesis is the development of an algorithm called the Dynamic Directed Graph (DDG) policy that can handle cycles, insertions and deletions in the directed graph corresponding to a knowledge base. This algorithm is an extension of the DAG policy. It includes structure maintenance operations to take care of the changing graph. Its locking rules are more general than the locking rules of the DAG policy because they take care of cycles and insertions and deletions in the graph. The algorithm supports both shared and exclusive locks and has a desirable property that it is *well-structured*. The DDG policy with only exclusive locks is *deadlock-free*, but the DDG-SX policy, in which both shared and exclusive locks are permitted, is not deadlock free.

To analyze the properties of the DDG policy, we develop a general theory of non-serializable schedules found in databases that undergo insertions and deletions. We show that, in such systems, the serializability graph of a *canonical non-serializable schedule*, a schedule in which all transactions except one are executed serially, is more general than the serializability graph of a canonical schedule found in databases that do not undergo insertions and deletions. We show the utility of this result by using it to prove the correctness of the DDG policy and two other locking policies.

We simulate the performance of the DDG policy and compare it to the performance of 2PL. While simulating the DDG policy, we consider several issues that arise in implementing it. We give a detailed design of a prototype implementation and highlight the computations at various stages of the execution of a transaction. We identify efficient algorithms that are necessary for performing these computations (for example, to compute *dominator tree* and to compute *strongly connected components*). In addition, we develop an algorithm that can incrementally compute strongly connected components. We highlight the assumptions that need to be made about the transactions and give a mechanism that controls the release of locks by a transaction before its locked point.

The evaluation of the algorithm is done in the context of three real applications (one of these is more extensively treated than the other two). We characterize the workload using a two-class system, in which Class 1 transactions consist of long transactions that access a large number of entities, and Class 2 transactions consist of short transactions that perform

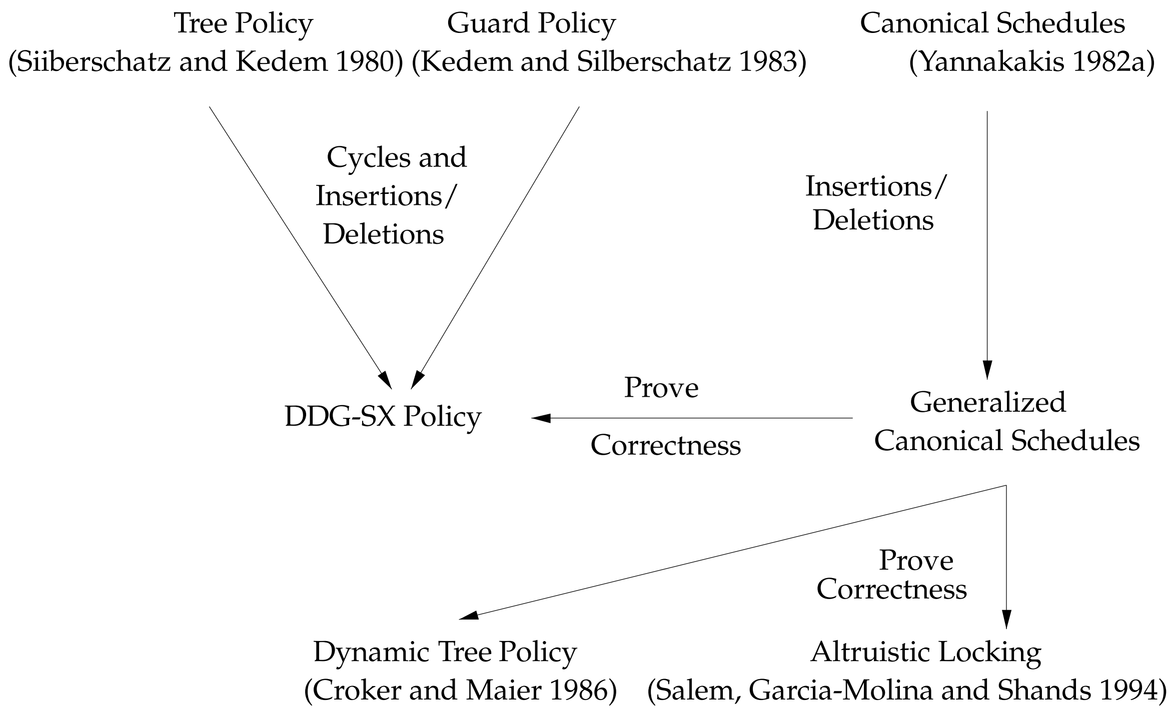


Figure 6.1: Relationship to past work on algorithms

lookups and updates. Under these conditions, we show that at high contention, the Class 2 response time obtained using the DDG policy is significantly better than the corresponding response time obtained using 2PL. In addition, we also show how to choose a subset of the semantic relationships of the knowledge base that should be used for concurrency control and the effect of traversal strategies on transaction response time.

Based on the evaluation of the algorithm, we proposed a hybrid technique that can switch between 2PL and the DDG policy depending on the load conditions. Such a technique almost always performs better than a system that uses only two-phase locking.

In this thesis, we used directed graph abstraction as a device to see how effectively one can exploit the semantic structure of a knowledge base to offer more efficient concurrency control than would have been possible otherwise. The DDG policy that was developed in the process of doing so has, however, a general applicability: it is applicable to any system in which the information is represented as a directed graph. Our theoretical results get tied with knowledge base applications in Chapter 5 when we consider three real knowledge base applications and evaluate the DDG policy.

Let us now discuss how the results of this thesis advance the current state of the art. The relationship of our algorithmic results (Chapter 2) to the previous work is shown in Figure 6.1. The key algorithmic ideas on which we have based our work are tree policy that dealt only with exclusive locks (Silberschatz and Kedem 1980) and the guard policy that dealt with both shared and exclusive locks (Kedem and Silberschatz 1983). The tree policy and guard policy, however, dealt only with static graphs. Our contribution is the extension of these algorithms to the DDG policy that provides a treatment of cycles and insertions and deletions to the graph. The proof technique that we have used is based on the idea of canonical schedules (Yannakakis 1982a). Our extension to this technique includes the treatment of insertions and deletions to the database giving a more general version of

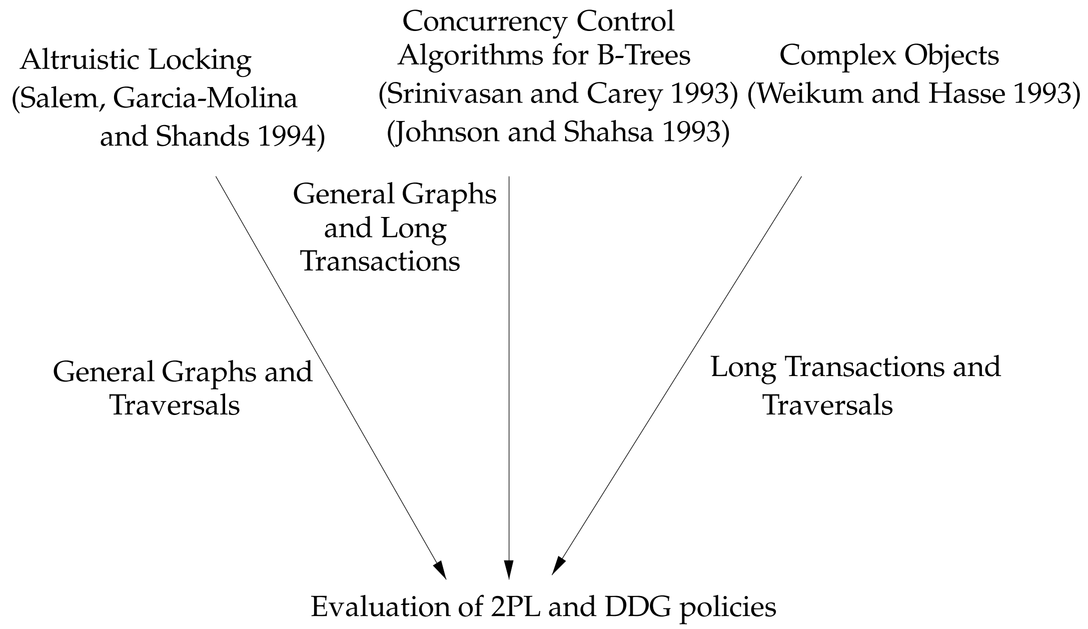


Figure 6.2: Relationship to past work on performance analysis

the canonical schedules theorem (Theorems 1 and 6). In the process of generalizing the canonical schedules theorem, we have introduced some new concepts: distinction between structural and value states of a database, proper schedules, and definition of locking policy as a function.

The past work that is the closest to the results presented in Chapter 3, is the implementation of concurrency control algorithms for B-Trees (Bayer and Schkolnick 1977). Due to the restricted shape and restricted set of transactions on a B-Tree, a simple technique such as *lock-coupling* (recall the definition from Section 2.7.2) suffices to implement a non-two-phase locking policy. The implementation techniques that we have presented in Chapter 3 are applicable in a more general setting. For example, the database does not have to be a tree or all the transactions do not have to start from the root, which is the case in a B-Tree. The implementation ideas for a non-two-phase locking that we have introduced include explicit computation of graph properties, introduction of a count to keep track of the number successors of a node that are yet to be locked, and an efficient mechanism to check for release of locks before a transaction reaches its locked point.

There has been considerable amount of work studying the performance of two-phase locking (Agrawal, Carey and Livny 1987; Tay 1987; Ryu and Thomasian 1990; Yu, Dias and Lavenberg 1993). Most previous work has focused on short transactions (of the order of 10-20 data items/transaction in a database of size 1000) and assumed uniform access patterns for the access (each item in the database can be accessed with equal probability) of the database. A skewed access pattern was also considered, in which a fraction b of all access requests fall in a fraction $1 - b$ of the database, for $0.5 < b < 1.0$. It was shown that the performance of 2PL under skewed access pattern is the same as that for a uniform access pattern for a database of a smaller size (that is a function of b).

In Figure 6.2, we show some of the papers that are related to our work on performance analysis in some aspects. For example, in the performance evaluation of altruistic locking (Salem, Garcia-Molina and Shands 1994), long transactions were explicitly considered (upto

800 items/transaction in a database of size 1000). But this study was limited to uniform access and skewed access patterns. In the performance evaluation of B-Tree algorithms (Srinivasan and Carey 1993; Johnson and Shasha 1993), the database was restricted to a B-Tree and transactions traverse the database from root to leaf. Similarly in the study of 2PL for a complex object database (Weikum and Hasse 1993), a random graph database was considered and transactions were limited to a traversal of the database upto only two levels. The distinguishing features of our work are long transactions (average length of 140 data items and a maximum length of over 1000 data items in a database of size about 3000) and access patterns (depth-first traversal and breadth-first traversal) that match well with the requirements of knowledge base applications.

6.2 Future Work

The results of this thesis can be extended in several directions as suggested in this section.

The Right Transaction Model for Knowledge Bases

The algorithmic development in this thesis has been done under the framework of serializability. After looking at some applications, such as APACS and TOVE, it appears to us that a somewhat different model may be needed. In the conventional model of a transaction, one of the assumptions is that if a transaction reads an item twice it gets the same value. In the APACS application, if a Class 1 or traversal transaction has read the value of an alarm, and while it is still active the value of the alarm changes, then it is mandatory that the traversal transaction should be notified. After receiving such a notification, a transaction may re-orient its traversal or may restart from the beginning. Such a requirement needs to be incorporated into a transaction model for knowledge bases, perhaps, by using ideas similar to "notify" locks (Gray and Reuter 1993).

Append-only Knowledge Bases

Several knowledge bases are append-only in nature, that is, new information may be added to the knowledge base but the old information is never changed. For such a system, known as an *immutable database*, or *time domain addressing system* or *version-oriented system*, one can devise a concurrency control scheme that unifies the data in the database with the recovery log. Even in the best attempt to address this problem (Reed 1983), several implementation problems could not be worked out. This direction is, however, very promising especially for specific domains such as medical records, engineering drawings, source-code control and system dictionaries, where version management is an important issue.

Workload Models

In this thesis, the access patterns that we considered were limited to depth-first traversal and breadth-first traversal. The access patterns in real knowledge based systems are likely to be more general. For example, one possible access model is to represent the traversal of a transaction by a set of three probabilities $\langle p, q, r \rangle$. At each step of a transaction, with a probability p, it continues traversal in the same direction, with a probability q, it stops, and with a probability r, it continues traversal in a different direction. A better understanding of the access patterns is possible only through observation on real systems. An accurate characterization of knowledge base workloads remains an open problem.

Multiple Granularity Protocols

The granularity of a data item refers to its relative size. For instance, the granularity of a class is *coarser*, and the granularity of an attribute *finer*, than that of an object. Using coarse granules incurs low overhead due to locking, since there are fewer locks to manage. On the other hand, it reduces concurrency since operations are more likely to conflict. Selecting a granularity of locks requires striking a balance between locking overhead and amount of concurrency. A multiple granularity protocol allows transactions to acquire locks at different levels of granularity (Gray et al. 1976). In the current version of our algorithm, we have assumed uniform granularity of locking. In the future, we plan to explore different levels of granularities that should be supported in a concurrency control algorithm for knowledge bases. In fact, efforts in this direction are under way at a research group elsewhere (de Ferreira Rezende and Härder 1994).

Benchmarks for Knowledge Bases

The knowledge bases considered in our performance studies contained less than 3000 objects. One of the reasons for this limited size was that the larger knowledge bases do not exist, and if they do, it is hard to get access to them. One approach to circumvent this problem is to synthetically generate the knowledge bases and the workload. There have been several such proposals for benchmarks in the database community (Cattell and Skeen 1992; Anderson et al. 1990; Carey, Dewitt and Naughton 1993; Gray et al. 1994) and the knowledge base community (Baader et al. 1992; Speele and Schouwenberg 1994; Karp, Paley and Greenberg 1994). These proposals are based on random graphs and do not in any rigorous way capture the semantic structures of knowledge bases. Therefore, we believe that we should attempt to more formally (Nytro 1992) understand the structures found in knowledge bases and then incorporate those formalisms into the benchmarks.

Recovery Techniques

As already discussed in Section 5.5.2, we need to design an appropriate recovery scheme for the DDG policy. Most conventional methods work on the assumption that the locks are held until the commit time and cannot deal with the situation in which locks can be released before commit time. In Section 5.5.2, we discussed two alternatives to deal with this problem: using compensating transactions or delaying the commit of some transactions. More research in this direction is necessary to decide which of the schemes would be more suitable for the DDG policy.

Generalization of the Theoretical Framework

There are several open theoretical problems that have been not solved in this thesis. For example, one could generalize our results on canonical schedules (in particular, Theorem 1) to define a general class of dynamic hyper-graph locking policies (Yannakakis 1982a). We feel that there is a need for a unifying framework that can relate several recent attempts of extending two-phase locking (for example, altruistic locking (Salem, Garcia-Molina and

Shands 1994) and ordered shared locking (Agrawal and Abbadi 1990)). Furthermore, we could devise techniques to ensure the deadlock-freedom of the DDG-SX policy (Fussell, Kedem and Silberschatz 1981; Yannakakis 1982b).

Implementation as Part of a Real System

We need to undertake an implementation of the DDG policy as part of a real system and observe its performance in a production setting. This will help in bringing realism into the quantitative evaluation and help us in validating our simulation model.

6.3 General Conclusions

The results of this thesis show that the knowledge base design is considerably influenced by the implementation considerations. For example, the presence of cycles in a knowledge base influences the design of the locking policy and the amount of concurrency permitted by it. It means that, for performance reasons, we should try to design a knowledge base with few and small cycles. This is an example of studying the implementation considerations as a function of the data model features, and we expect that this will be an important research methodology in the design of knowledge based systems.

Similarly, we found that the depth-first traversal strategy led to a better response time as compared to breadth-first. Traditionally, traversal strategy has been under the control of a query processor and the designs of transaction manager and query processor have been considered in isolation. Our result on the influence of traversal strategy on concurrency control performance shows that the interaction between query processor and concurrency control can play an important role in the overall performance of the system, and thus, presents an alternative way to approach the DBMS design.

An interesting aspect of our work is the synthesis of different research areas. Most of the work on concurrency control algorithms either deals with correctness issues or with performance considerations or with implementation problems. We have tried to deal with all the three aspects of this problem. Such an approach is quite effective in striking a balance between theory and practice. It also helps in giving a better understanding into the implementation bottlenecks that need further formalization.

In conclusion, we note that database technology needs to place more emphasis on the integration of modeling features and implementation techniques coming from particular streams of database research, such as ones on object-oriented, deductive, active, temporal and spatial databases. Such integration is essential for the generation of a technology that fully addresses the modeling and performance requirements of future applications.

Appendix A

Analysis of Simulation Results

A.1 Introduction

In this appendix, we give details about the analysis of the simulation output. Specifically, we consider the influence of the warm up period, determination of batch size and simulation length, confidence interval computation, and finally, the comparisons of the response times. More details about the statistical basis of the analysis can be found elsewhere (Law and Kelton 1991; Welsh 1983). We illustrate these computations using an example data point from the baseline experiment described in Section 5.1.5. We fix the Class 1 write probability at 0.4 and the multiprogramming level at 5.

A.2 Warmup Period

Let $X_1, X_2, \ldots, X_N$ be the observations of a random variable X (for example, response time). In general, because the system is started "empty" and "idle", a "warmup" period is necessary. Therefore, $n_0 \leq N$ of the observations must be ignored as they do not correspond to the steady state distribution of the sample mean $\overline{X}$. In order to determine the extent of warmup period in our simulation, we plot in Figure A.1, the running value of mean response time for the DDG policy for both Class 1 and Class 2. We can see that the mean response time stabilizes after approximately 3000 observations of Class 1 response time, and about 200,000 observations of Class 2 response time.

One approach to deal with this problem is to ignore the observations that are not in the steady state (3000 observations for Class 1 and 200,000 observations for Class 2). The approach that we followed was to conduct a simulation run long enough such that the influence of the initial transient is insignificant on the response time. For example, in the above example, for Class 1, the mean response time without ignoring the initial transient was 83.77 and after ignoring the initial transient was 83.95. The corresponding values for Class 2 response time were 3.32 and 3.38. Since the two are approximately equal, not ignoring the initial transient does not significantly influence our result. A similar analysis was done for the variance of the response time and the same behavior was observed.

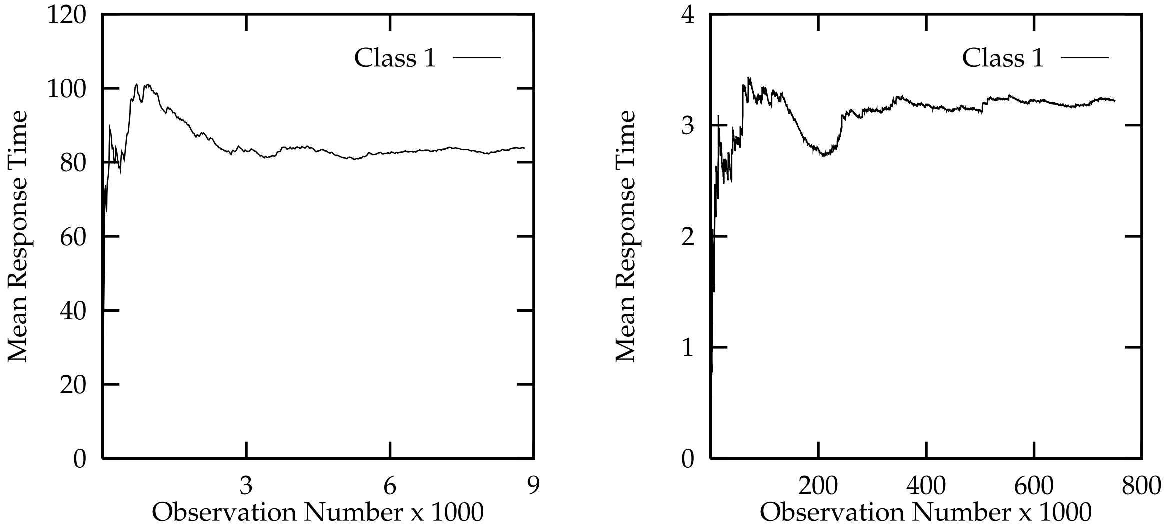


Figure A.1: Choosing warmup period

A.3 Determination of Batch Size

We used the batch means method to compute the confidence interval on the mean response time. This involves dividing the observations into batches of equal size B. A key problem then is to determine the batch size such that the observation from each batch are independent and identically distributed random variables. For a random variable X with observations $X_1, \ldots, X_N$, such that the sample mean is $\overline{X}$, we define the correlation function $\rho(k), k < N$, as follows:

$$\gamma(k) = \frac{1}{N-k} \sum_{n=1}^{N-k} (X_n - \overline{X})(X_{n+k} - \overline{X}), k \geq 0$$

$$\rho(k) = \gamma(k)/\gamma(0)$$

If the batch size is chosen such that the correlation coefficient between the observations of the successive batches is approximately equal to zero, we can claim that the observations in each batch are independent. Since the system is in a steady state, the observations are also identically distributed.

In order to determine the correlation function in the simulation output, we plot in Figure A.2 the correlation coefficient for both Class 1 and Class 2 response times for the DDG policy. We can see that for Class 1 response time, the correlation function approaches 0.0 as the batch size becomes greater than 30. For Class 2 response time, the correlation function approaches 0.0 as the batch size approaches 500. We used a batch size of 30 for Class 1 and 100 for Class 2. This means that the confidence levels for Class 2 response times may be higher than the actual. We check the extent of this error in the next section.

A.4 Confidence Intervals and Simulation Length

We construct 95% confidence intervals on the mean response time for both Class 1 and Class 2. If $S(N)$ denotes the standard deviation of N batch means, then approximate $100(1-\alpha)\%$

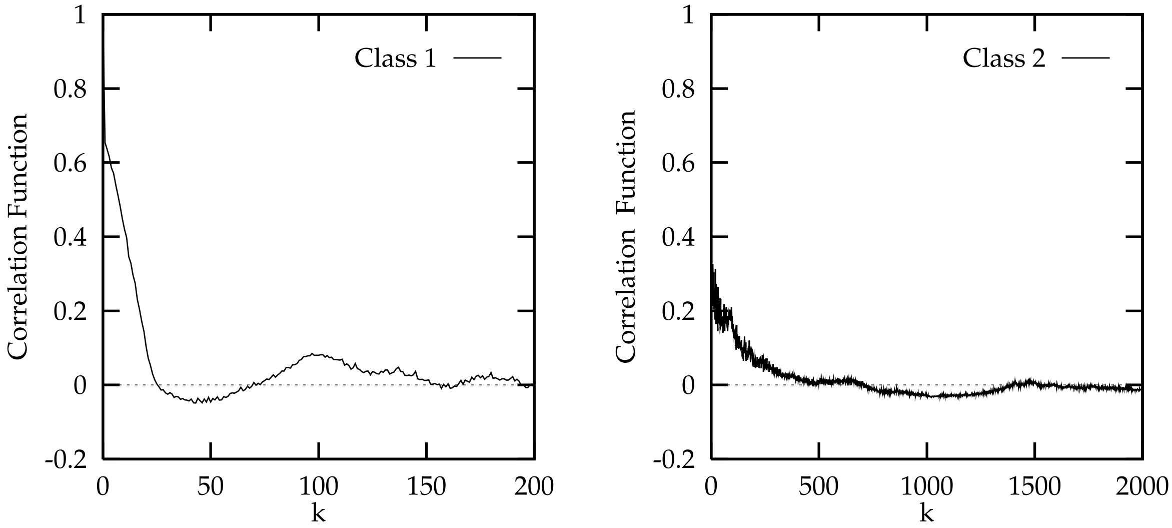


Figure A.2: Choosing batch size

Response Time	Number of Batches	Batch Mean	Batch Standard Deviation	Confidence Interval Half Length	
				Absolute	% of Mean
DDG Class 1	294	83.78	34.22	3.92	4.67
2PL Class 1	252	76.68	30.83	3.82	5.0
DDG Class 2	7501	3.22	7.27	0.17	5.1
2PL Class 2	5883	6.28	9.61	0.25	3.9

Table A.1: An example computation of confidence intervals

confidence intervals can be computed as:

$$\overline{X} \pm z_{1-\alpha/2}\frac{S(N)}{\sqrt{N}}$$

where $z_{1-\alpha/2}$ is the upper $1-\alpha/2$ critical point for a standard Normal variable. To obtain 95% confidence interval, we set $\alpha = 0.05$ giving $z_{0.975} = 1.96$. An example computation is shown in Table A.1.

If we calculate the confidence interval for Class 2 response time using a batch size of 500, we get confidence interval half length of 0.25 (7.65% of the mean) for the DDG policy and 0.42 (6.7% of the mean) for 2PL. By comparing them with the values shown in the Table A.1, we can see that our estimates based on these calculations might be some what optimistic.

To determine the simulation length, we use a sequential sampling procedure. After every batch we compute the confidence interval on the mean response time and compare its half length with the current estimate of the mean value. If the half length of the confidence interval is greater than 5% of the current estimate of the mean value we continue to collect more samples, otherwise, we stop.

Class	$R_{2PL} - R_{DDG}$	S	z	Probability($R_{2PL} \geq R_{DDG}$) $\Phi(z)$
Class 1	-7.09	2.8	2.45	0.02
Class 2	3.06	0.15	-17.02	1.0

Table A.2: Comparison of response times

A.5 Comparison of the Two Policies

Since the batch means are independent and identically distributed random variables, by using central limit theorem, the distribution of the estimate of the overall mean can be approximated by Normal distribution. Let R_{2PL} and R_{DDG} be the sample mean response times and S^2_{2PL} and S^2_{DDG} the sample batch variances of 2PL and the DDG policy respectively as obtained from the simulation. Furthermore, let N_{2PL} and N_{DDG} be the number of batches for 2PL and DDG respectively. $R = R_{2PL} - R_{DDG}$ has a Normal distribution with mean $R_{2PL} - R_{DDG}$ and variance $S^2 = \frac{S^2_{2PL}}{N_{2PL}} + \frac{S^2_{DDG}}{N_{DDG}}$.

Using this, we can compute the probability that mean response times of the two policies are different, thus, giving the statistical significance of the comparison. To do this, we compute $z = -(R_{2PL} - R_{DDG})/S$. Then the probability that the $R_{2PL} \geq R_{DDG}$ is given by $\Phi(z)$ where $\Phi(z)$ is the area under the standard Normal curve from z to ∞. An example of this computation is shown in Table A.2.

Appendix B

Statistical Significance of Comparisons

In this appendix, we give detailed computations of the statistical significance of the comparisons presented in Chapter 5. All computations are presented in tabular form. Before proceeding further, let us first explain some notation.

The alternatives being compared are denoted by I and II. For example, if we are comparing traversal strategies, "I" may denote BFT and "II" may denote DFT. The first column of the table (labeled MPL) gives multiprogramming level. N_i, R_i and S_i respectively denote the number of batches, mean and standard deviation for the response time given by alternative i, where i can be either I or II. $R_I - R_{II}$ and S_{I-II} denote the mean and standard deviation of the distribution of the difference of the two mean response times being compared. The standard deviation is computed using the expression given in Section A.5. z is computed as $-(R_I - R_{II})/S_{I-II}$. $\Phi(z)$ denotes the area under the standard normal curve from z to ∞ and gives the value of Probability(I $\geq$ II), which is the probability that the mean response time given by alternative I is greater than the mean response time given by alternative II.

Let us now present the actual computations. With each table we cite the corresponding experiment from Chapter 5.

MPL	I=isA+instanceOf +linkedTo+partOf			II=isA+instanceOf			R_I-R_{II}	S_{I-II}	z	$\Phi(z)$
	N_I	R_I	S_I	N_{II}	R_{II}	S_{II}				
1	162	101.43	36.68	138	99.96	35.52	1.47	4.18	-0.35	0.64
2	154	86.74	32.58	170	80.16	31.05	6.58	3.54	-1.86	0.97
3	171	80.49	31.92	182	76.44	31.33	4.05	3.37	-1.20	0.88
5	198	79.23	33.76	191	74.19	31.17	5.04	3.29	-1.53	0.94
10	209	78.11	33.05	219	74.62	30.21	3.49	3.07	-1.14	0.87
25	209	76.37	68.90	222	71.69	59.42	4.68	6.21	-0.75	0.77

Comparison of Class 1 response time

MPL	I=isA+instanceOf +linkedTo+partOf			II=isA+instanceOf			R_I-R_{II}	S_{I-II}	z	$\Phi(z)$
	N_I	R_I	S_I	N_{II}	R_{II}	S_{II}				
1	5001	1.25	2.78	4197	0.94	2.16	0.31	0.05	-5.90	1.00
2	4060	2.17	4.10	4159	1.64	3.22	0.53	0.08	-6.45	1.00
3	4194	2.87	5.11	4235	2.20	4.11	0.67	0.10	-6.63	1.00
5	4780	3.97	6.89	4319	3.12	5.55	0.85	0.13	-6.46	1.00
10	5000	5.98	13.96	4999	5.35	14.11	0.63	0.28	-2.26	0.99
25	4997	12.45	50.34	4997	10.96	43.53	1.49	0.94	-1.58	0.94

Comparison of Class 2 response time

The last column, labeled $\Phi(z)$, gives the Probability(I $\geq$ II).
This table corresponds to Figure 5.3.

Table B.1: Choice of Graph: Dynamic Implementation

MPL	I=isA+instanceOf +linkedTo+partOf			II=isA+instanceOf			$R_I - R_{II}$	S_{I-II}	z	$\Phi(z)$
	N_I	R_I	S_I	N_{II}	R_{II}	S_{II}				
1	137	97.04	34.44	195	95.17	35.72	1.87	3.90	-0.48	0.68
2	160	80.25	30.82	178	77.40	29.97	2.75	3.31	-0.86	0.81
3	172	75.20	29.90	183	73.93	30.25	1.27	3.19	-0.40	0.66
5	199	73.59	31.49	196	71.54	30.40	2.05	3.11	-0.66	0.75
10	239	74.18	29.19	286	72.25	28.95	1.93	2.55	-0.76	0.78
25	335	72.46	61.83	341	70.81	57.91	1.65	4.61	-0.36	0.64

Comparison of Class 1 response time

MPL	I=isA+instanceOf +linkedTo+partOf			II=isA+instanceOf			$R_I - R_{II}$	S_{I-II}	z	$\Phi(z)$
	N_I	R_I	S_I	N_{II}	R_{II}	S_{II}				
1	4039	1.17	2.53	5665	0.88	2.02	0.29	0.05	-5.84	1.00
2	3903	2.05	3.88	4219	1.54	3.04	0.51	0.08	-6.52	1.00
3	3949	2.61	4.62	4127	2.11	3.94	0.50	0.10	-5.33	1.00
5	4459	3.63	6.09	4272	3.05	5.47	0.58	0.12	-4.66	1.00
10	5423	5.76	12.89	6319	5.15	12.43	0.61	0.23	-2.61	1.00
25	7499	11.39	44.32	7499	10.56	39.76	0.83	0.69	-1.20	0.88

Comparison of Class 2 response time

The last column, labeled $\Phi(z)$, gives the Probability(I $\geq$ II).
This table corresponds to Figure 5.4.

Table B.2: Choice of Graph: Static Implementation

MPL	I=BFT			II=DFT			R_I-	S_{I-II}	z	$\Phi(z)$
	N_I	R_I	S_I	N_{II}	R_{II}	S_{II}	R_{II}			
1	143	84.51	30.64	213	83.01	31.52	1.50	3.27	-0.46	0.68
2	173	70.26	28.03	191	68.49	28.08	1.77	2.93	-0.60	0.73
3	191	67.49	28.24	195	64.65	27.35	2.84	2.83	-1.01	0.84
5	194	65.20	27.57	207	62.48	27.21	2.72	2.68	-1.01	0.84
10	223	64.17	29.02	302	62.91	25.00	1.86	2.80	-0.45	0.67
25	376	64.66	56.32	391	62.11	49.31	2.55	2.90	-0.88	0.81

Comparison of Class 1 response time

MPL	I=BFT			II=DFT			R_I-	S_{I-II}	z	$\Phi(z)$
	N_I	R_I	S_I	N_{II}	R_{II}	S_{II}	R_{II}			
1	3674	1.96	3.41	5415	0.79	1.78	1.17	0.06	-18.37	1.00
2	3703	2.84	4.06	4022	1.40	2.70	1.44	0.08	-17.33	1.00
3	3936	3.49	4.57	3849	1.91	3.56	1.58	0.10	-16.14	1.00
5	3857	4.41	5.70	3957	2.69	4.70	1.72	0.14	-11.99	1.00
10	4365	6.86	13.66	5831	4.44	10.30	2.42	0.32	-7.58	1.00
25	7500	11.32	38.84	7501	9.20	33.82	2.12	0.45	-4.72	1.00

Comparison of Class 2 response time

The last column, labeled $\Phi(z)$, gives the Probability(I $\geq$ II).
This table corresponds to Figure 5.5.

Table B.3: Choice of traversal strategy

MPL	I=2PL			II=DDG			R_I-R_{II}	S_{I-II}	z	$\Phi(z)$
	N_I	R_I	S_I	N_{II}	R_{II}	S_{II}				
1	218	69.73	31.26	316	77.21	32.62	-7.48	2.80	2.67	0.00
2	279	54.34	25.72	271	59.94	27.32	-5.60	2.26	2.47	0.01
3	273	48.63	22.65	247	54.94	25.50	-6.31	2.12	2.97	0.00
5	250	43.80	20.99	226	49.85	22.77	-6.05	2.01	3.00	0.00
10	231	41.19	18.96	240	46.66	21.93	-5.47	1.89	2.90	0.00
25	602	40.21	46.93	537	44.91	44.57	-4.70	2.71	1.73	0.04

Comparison of Class 1 response time

MPL	I=2PL			II=DDG			R_I-R_{II}	S_{I-II}	z	$\Phi(z)$
	N_I	R_I	S_I	N_{II}	R_{II}	S_{II}				
1	4642	1.07	2.76	7443	1.13	2.97	-0.06	0.05	1.28	0.10
2	4655	2.02	4.18	4984	2.03	4.35	-0.01	0.09	0.11	0.46
3	4078	2.88	5.59	4156	2.78	5.46	-0.10	0.12	0.80	0.79
5	3381	4.09	7.22	3454	3.94	6.99	0.15	0.17	-0.87	0.81
10	2939	6.41	10.07	3442	6.06	9.68	0.35	0.25	-1.44	0.93
25	7500	10.27	24.38	7500	9.79	21.34	0.48	0.37	-1.28	0.88

Comparison of Class 2 response time

The last column, labeled $\Phi(z)$, gives the Probability(I $\geq$ II).
This table corresponds to Figure 5.7.

Table B.4: Comparison of DDG and 2PL at write probability 0.0

MPL	I=2PL			II=DDG			$R_I - R_{II}$	S_{I-II}	z	$\Phi(z)$
	N_I	R_I	S_I	N_{II}	R_{II}	S_{II}				
1	218	96.92	43.38	227	108.67	47.79	-11.75	4.32	2.72	0.00
2	238	81.97	38.33	272	90.42	41.52	-8.45	3.54	2.39	0.08
3	259	77.78	38.01	285	86.27	40.67	-8.49	3.37	2.52	0.01
5	252	76.69	36.89	294	83.78	40.92	-7.09	3.33	2.13	0.02
10	295	76.64	36.48	297	82.80	39.85	-6.16	3.14	1.96	0.03
25	324	74.18	74.86	299	80.80	84.73	-6.62	6.43	1.03	0.15

Comparison of Class 1 response time

MPL	I=2PL			II=DDG			$R_I - R_{II}$	S_{I-II}	z	$\Phi(z)$
	N_I	R_I	S_I	N_{II}	R_{II}	S_{II}				
1	6430	1.97	4.65	7503	0.86	3.04	1.11	0.07	-16.47	1.00
2	5944	3.50	6.82	7500	1.51	4.53	1.99	0.10	-19.37	1.00
3	6128	4.57	8.40	7500	2.18	6.06	2.39	0.13	-18.62	1.00
5	5883	6.28	11.45	7501	3.22	8.66	3.06	0.18	-17.02	1.00
10	6909	8.14	20.56	7501	5.58	20.11	2.56	0.34	-7.54	1.00
25	7497	13.78	57.50	7501	11.78	63.23	2.00	0.99	-2.02	0.98

Comparison of Class 2 response time

The last column, labeled $\Phi(z)$, gives the Probability(I $\geq$ II).
This table corresponds to Figure 5.8.

Table B.5: Comparison of DDG and 2PL at write probability 0.4

Appendix C

Sample Transactions

C.1 Example of Class 1 Transaction

In Figure C.1, we show an example transaction that performs a traversal along the `isA` relationship. Using the second parameter of the function `select` (line 8), it specifies whether to select direct instances (leading to a traversal over two levels) or to return all instances (leading to a traversal up to the maximum depth of the knowledge base). In Figure C.2, we show another transaction that performs traversal along the `linkedTo` relationship in the knowledge base. If there is a change in flow at a component, then this program propagates the change to the components that are connected to it. For example, consider the component `flow_assembly` (Line 4). If the pressure at its input node rises, then the pressure at its output node (the node that has a `linkedTo` relationship to it) also rises. This is done using the rules specified in Lines 7-8. This process continues in one direction across all the components that are linked to each other as each of these components has a similar rule. Again in this transaction, we know the node from which the transaction will begin traversal. These examples demonstrate that when a transaction begins it has information about the starting node and the number of levels to traverse.

C.2 Example of Class 2 Transaction

Consider the transaction shown in Figure C.3. This transaction loads the value of pressure for the component `HX_FLOW_ASSEMBLY`. The transaction accesses two slots: `narrow_pressure_instrument` (Lines 3-5) and `wide_pressure_instrument` (Lines 7-9) and updates their value. When such a transaction begins execution we know the entities that it is going to access.

```
void ClassBuffer::Search_ins(const char* dbname, char* class_name)
  { PClass *pcls = PClass::find(class_name);
    if (pcls == NULL) {
        printf("Class '%s' is not found!\n", class_name);
        return;
  }
  LinkVstr<ApacsObj> ins =
          pcls->select( dbname, FALSE, NULL_PREDICATE );
  for (int i = 0; i < ins.size(); i++)
   {
        attrs_def* tmp;
        tmp = ins[i]->VDump();
        InstInfo* i_temp = new InstInfo((char*)ins[i]->name, tmp);
  }
  delete pcls;
}
```

Figure C.1: Traversal along `isA` and `instanceOf`

```
group flow-propogation
      $t:symbol number
      block
            object =c[flow_assembly]
            node =input:$c.input_node
            node =output:$c.output_node
            rule flow-through-component-forward :forward
                    0,$t ($input.flow') => 0,$t ($output.flow')
            rule flow-through-component-backward :backward
                    0,$t ($output.flow') => 0,$t ($input.flow')
      block
            object =c[hx_flow_assembly]
            node =input:$c.input_node
            node =output:$c.output_node
            rule flow-through-component-forward-hx :forward
                    0,$t ($input.flow') =>
                          0,$t ($output.flow'),
                          0,$t ($output.pressure')
            rule flow-through-component-backward-hx+ :backward
                    0,+ ($output.flow') =>
                          0,+ ($input.flow'),
                          0,- ($output.pressure')
            rule flow-through-component-backward-hx- :backward
                    0,- ($output.flow') =>
                          0,- ($input.flow'),
                          0,+ ($output.pressure')
      block
            object =c[container_assembly]
            node =input:$c.input_node
            rule flow-through-component-forward-container :forward
                   0,$t ($input.flow') =>
                          0,$t ($c.level')
```

This transaction has been coded in a home-grown rule language. Its meaning can be understood by following the explanation given in Section A.1.

Figure C.2: Traversal along `linkedTo` relationship

```
void  HX_FLOW_ASSEMBLY::VLoadSlot(char* slot, char* Vval) {
  FLOW_ASSEMBLY::VLoadSlot(slot, Vval);
  if (!strcmp(slot, "narrow_pressure_instrument")) {
        narrow_pressure_instrument =
              (MEASUREMENT_INSTRUMENT*)ApacsObj_find(Vval);
  }
  if (!strcmp(slot, "wide_pressure_instrument")) {
        wide_pressure_instrument =
              (MEASUREMENT_INSTRUMENT*)ApacsObj_find(Vval);
  }
}
```

Figure C.3: An update transaction

References

Adams, E. W. and Miller, M. H. T. C. 1989. Object Management in CASE Environment. In *Proceedings of the 11th International Conference on Software Engineering*, pages 154–163.

Agrawal, D. and Abbadi, A. E. 1990. Locks with Constrained Sharing. In *9TH ACM Symposium on Principles of Database Systems*, pages 85–93.

Agrawal, R., Carey, M. J., and Livny, M. 1987. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems*, 12(4):609–654.

Aho, A., Sethi, R., and Ullman, J. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company.

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. 1987. *Data Structure and Algorithms*. Addison-Wesley Publishing Company.

Alonso, G., Agrawal, D., and Abbadi, A. E. 1994. Reducing Recovery Constraints on Locking Based Protocols. In *13TH ACM Symposium on Principles of Database Systems*, pages 129–138.

Alonso, G., Vingralek, R., Agrawal, D., Breitbart, Y., Abbadi, A. E., Schek, H.-J., and Weikum, G. 1994. Unifying Concurrency Control and Recovery of Transactions. *Information Systems*, 19(1):101–115.

Alpern, B., Hoover, R., Rosen, B. K., Sweeney, P. F., and Zadeck, F. K. 1990. Incremental Evaluation of Computational Circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42, San Francisco.

Anderson, T. L., Berre, A. J., Mallison, M., Porter, H. H. I., and Schneider, B. 1990. The Hypermodel Benchmark. In *Proceedings of Extending Database Technology*, pages 317–331. Springer Verlag.

Baader, F., Hollunder, B., Nebel, B., Profitlich, H.-J., and Franconi, E. 1992. An Empirical Analysis of Optimization Techniques for Terminological Representation Systems. In *Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 270–281.

Bayer, R. and Schkolnick, M. 1977. Concurrency of Operations on B-trees. *Acta Informatica*, 9(1):1–21.

Beck, H. W., Gala, S. K., and Navathe, S. B. 1989. Classification as a Query Processing Technique in the CANDIDE Semantic Data Model. In *IEEE International Conference on Data Engineering*, pages 572–581.

Berghouti, N. S. and Kaiser, G. E. 1991. Concurrency Control in Advanced Database Applications. *Computing Surveys*, 23(3):269–318.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Welssley Publishing Company.

Bernstein, P. A., Rothnie, J. B. J., Goodman, N., and Papadimitriou, C. H. 1978. The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case). *IEEE Transactions on Software Engineering*, 4(3):154–168.

Borgida, A., Brachman, R. J., McGuinness, D. L., and Resnick, L. A. 1989. CLASSIC: A Structural Data Model for Objects. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 58–67, Portland, Oregon.

Borgida, A. and Patel-Schneider, P. F. 1994. A Semantics and Complete Algorithm for Subsumption in CLASSIC Description Logic. *Journal of Artificial Intelligence Research*, 1:277–308.

Brachman, R. J., McGuinness, D. L., Patel-Schneider, P. F., and Resnick, L. A. 1990. Living with CLASSIC: When and How to Use a KL-ONE Like Language. In Sowa, J., editor, *Principles of Semantic Networks*. Morgan Kaufmann Publishers, CA.

Buckley, G. and Silberschatz, A. 1984. On the Heterogeneous Guard Locking Protocol. *The Computer Journal*, 27(1):86–87.

Carey, M. J., Dewitt, D. J., and Naughton, J. F. 1993. The OO7 Benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington D.C.

Carey, M. J., Krishnamurthi, S., and Livny, M. 1990. Load Control for Locking: The 'Half-and-Half' Approach. In *Proceedings of the 1990 ACM Symposium on Principles of Database Systems*, pages 72–84.

Carey, M. J. and Muhanna, W. A. 1986. The Performance of Multiversion Concurrency Control Algorithms. *ACM Transactions on Database Systems*, 4(4):338–378.

Caroll, M. D. 1988. Data Flow Analysis via Dominator and Attribute Updates. Technical Report LCSR-TR-111, Rutgers University.

Cattell, R. G. G. and Skeen, J. 1992. Object Operations Benchmark. *ACM Transactions on Database Systems*, 17(1):1–31.

Chan, A., Fox, S., Lin, W.-T. K., Nori, A., and Ries, D. 1990. The Implementation of an Integrated Concurrency Control and Recovery Scheme. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 271–280, Atlantic City, NJ.

Chaudhri, V. K. 1994. Source Listings for APACS, Wines and Enterprise Integration Knowledge Bases. Available by World Wide Web using URL ftp://ftp.cs.toronto.edu/pub/kbms/vinay/test-data.

Chaudhri, V. K. and Hadzilacos, V. 1995. Safe Locking Policies for Dynamic Databases. In *14TH ACM Symposium on Principles of Database Systems*. To Appear.

Chaudhri, V. K., Hadzilacos, V., and Mylopoulos, J. 1992. Concurrency Control for Knowledge Bases. In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning*, pages 762–773.

Chaudhri, V. K., Hadzilacos, V., Mylopoulos, J., and Sevcik, K. 1994. Quantitative Evaluation of a Transaction Facility for a Knowledge Base Management System. In *Proceedings of the Third International Conference on Knowledge Management*, pages 122–131, Gaithersberg, MD.

Chaudhri, V. K. and Mylopoulos, J. 1995. Efficient Algorithms and Performance Results for Multi-user Knowledge Bases. Technical report, University of Toronto.

Croker, A. and Maier, D. 1986. A Dynamic Tree-Locking Protocol. In *Proceedings of the International Conference in Data Engineering*, pages 49–56, Los Angeles, CA.

de Ferreira Rezende, F. and Härder, T. 1994. A Lock Method for KBMSs Using Abstraction Relationships' Semantics. In *Proceedings of the Third International Conference on Knowledge Management*, pages 112–121.

Dietz, P. F. and Sleator, D. D. 1987. Two Algorithms for Maintaining Order in a List. In *Proceedings of the Nineteenth ACM Symposium on Theory of Computing*, pages 365–372, New York.

Ellis, G. 1995. *Managing Complex Objects*. PhD thesis, University of Queensland, Australia. Forthcoming.

Eswaran, K., Gray, J., Lorie, R., and Traiger, I. 1976. The Notions of Consistency and Predicate Locks in Database Systems. *Communications of the ACM*, 19(9):624–633.

Farrag, A. A. and Ozsu, M. T. 1989. Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM Transactions on Database Systems*, 14(4):503–525.

Filman, R. E. 1989. New Generation Knowledge System Development Tools. Technical Report DARPA Contract No. F30602-85-C-0065, IntelliCorp Inc.

Fox, M. S., Chionglo, J. F., and Fadel, F. G. 1993. A Common-sense Model of the Enterprise. In *Proceedings of the Second Industrial Engineering Research Council (IERC)*, pages 425–429. Institute of Industrial Engineering.

Fussell, D., Kedem, Z., and Silberschatz, A. 1981. A Theory of Correct Locking Protocols for Database Systems. In *Proceedings of the Seventh International Conference on Very Large Databases*, pages 112–124.

Garcia-Molina, H. 1983. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2):186–213.

Garcia-Molina, H. and Salem, K. 1987. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 249–259. ACM Press.

Genesereth, M. R. and Nilsson, N. J. 1987. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, Los Altos, CA.

Gray, J. 1981. The Transaction Concept: Virtues and Limitations. In *7th International Conference on Very Large Databases*, pages 393–481, Cannes, France. Springer Veralg. Lecture Notes in Computer Science No. 60.

Gray, J. and Reuter, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., San Mateo.

Gray, J., Sundaresan, P., Englert, S., Baclawski, K., and Weinberger, P. J. 1994. Quickly Generating Billion-Record Synthetic Databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 24–27, Minneapolis.

Gray, J. N. 1992. Personal Communication.

Gray, J. N., Lorie, R. A., Putzolu, G. R., and Traiger, I. L. 1976. Granularity of Locks and Degrees of Consistency in a Shared Database. In Nijssen, G. M., editor, *Modeling in Database Management Systems*, pages 365–394. North Holland Publishing Company.

Guha, R. V. 1994. Personal Communication.

Guha, R. V. and Lenat, D. B. 1994. Enabling Agents to Work Together. *Communications of the ACM*, 37(7):126–142.

Holt, R. C. 1972. Some Deadlock Properties in Computer Systems. *Computing Surveys*, 4(3):179–196.

Ishida, T., Yokoo, M., and Gasser, L. 1990. An Organizational Approach to Adaptive Production Systems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 52–58, Boston, MA.

Italiano, G. F. 1986. Amortized Efficiency of a Path Retrieval Data Structure. *Theoretical Computer Science*, 48(2,3):273–281.

Italiano, G. F. 1988. Finding Paths and Deleting Edges in Directed Acyclic Graphs. *Information Processing Letters*, 28(1):5–11.

Johnson, T. and Shasha, D. 1993. The Performance of Current B-Tree Algorithms. *ACM Transactions on Database Systems*, 8(1):51–101.

Karp, P. D., Paley, S. M., and Greenberg, I. 1994. A Storage System for Scalable Knowledge Representation. In *Third ACM International Conference on Information and Knowledge Management*, pages 97–104, Gaithersberg, MD.

Kedem, Z., Mohan, C., and Silberschatz, A. 1982. An Efficient Deadlock Removal Scheme for Non-Two Phase Locking Protocols. In *Proceedings of the Eighth International Conference on Very Large Databases*, pages 91–97.

Kedem, Z. M. and Silberschatz, A. 1981. A Characterization of Database Graphs Admitting a Simple Locking Protocol. *Acta Informatica*, 16(1):1–13.

Kedem, Z. M. and Silberschatz, A. 1983. Locking Protocols: From Exclusive to Shared Locks. *Journal of the Association for Computing Machinery*, 30(4):787–804.

Korth, H. F., Kim, W., and Banchilhon, F. 1988. On Long Duration CAD Transactions. *Information Sciences*, 46:73–107.

Korth, H. F., Levy, E., and Silberschatz, A. 1990. Compensating Transactions: A New Recovery Paradigm. In *Proceedings of 16th International Conference on Very Large Databases*, pages 95–106, Brisbane, Australia.

Lanin, V. and Shasha, D. 1990. Tree Locking on Changing Trees. Technical Report 503, New York University.

Lavenberg, S. S. 1983. *Computer Performance Modeling Handbook*. Academic Press, New York.

Law, A. M. and Kelton, W. D. 1991. *Simulation Modeling and Analysis*. McGraw-Hill New York, NY.

Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevcik, K. C. 1984. *Quantitative System Performance: Computer System Analysis using Queueing Network Models*. Prentice-Hall, Inc., Englewood Cliffs, NJ.

Lengauer, T. and Tarjan, R. E. 1979. A Fast Dominator Algorithm for Finding Dominators in a Flow Graph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141.

Levesque, H. J. and Brachman, R. J. 1985. A Fundamental Tradeoff in Knowledge Representation and Reasoning. In Brachman, R. J. and Levesque, H. J., editors, *Readings in Knowledge Representation*, pages 42–70. Morgan Kaufmann Publishers, CA.

Livny, M. 1990. DeNeT User's Guide (Version 1.5). Technical report, University of Wisconsin.

Lynch, N. 1983. Multilevel Atomicity—A New Correctness Criterion for Database Concurrency Control. *ACM Transactions on Database Systems*, 8(4):484–502.

Mays, E., Lanka, S., Dionne, B., and Weida, R. 1991. A Persistent Store for Large Shared Knowledge Bases. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):33–41.

Miranker, D. P. 1991. Speical Issue on the Parallel Execution of Rule Systems. *Journal of Parallel and Distributed Computing*, 13(4).

Mohan, C., Fussell, D., and Silberschatz, A. 1982. Compatibility and Commutativity in Non-Two-Phase Locking Protocols. In *8TH ACM Symposium on Principles of Database Systems*, pages 283–292.

Mohan, C., Kedem, Z., Fussell, D., and Silberschatz, A. 1985. Lock Conversion in Non-Two-Phase Locking Protocols. *IEEE Transactions on Software Engineering*, 11(1):15–22.

Mylopoulos, J., Borgida, A., Jarke, M., and Koubarakis, M. 1990. Telos: A Language for Representing Knowledge About Information Systems. *ACM Transactions on Information Systems*, 8(4):325–362.

Mylopoulos, J., Chaudhri, V. K., Plexousakis, D., and Topaloglou, T. 1992a. A Performance-Oriented Approach to Knowledge Base Management. In *Proceedings of the First International Conference on Knowledge Management*, pages 68–75, Baltimore, MD. Also appeared as: Adapting Database Implementation Techniques to Manage Very Large Knowledge Bases, In International Workshop on Building and Sharing Very Large Knowledge Bases, Dec 3-4, 1993, Tokyo, Japan, pages 215-226.

Mylopoulos, J., Kramer, B., Wang, H., Benjamin, M., Chou, Q. B., and Mensah, S. 1992b. Expert System Applications in Process Control. In *Proceedings of the International Symposium on Artificial Intelligence in Materials Processing Applications*, Edmonton.

Naqvi, S. and Tsur, S. 1989. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, New York.

Nytro, O. 1992. *Aspects of Structure in Horn Clause Programs*. PhD thesis, University of Trondheim, Norway.

Papadimitriou, C. 1979. Serializability of Concurrent Database Updates. *Journal of the Association for Computing Machinery*, 26(4):631–653.

Papadimitriou, C. 1986. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, MD.

Plexousakis, D. 1993. Integrity Constraint and Rule Maintenance in Temporal Deductive Knowledge Bases. In *Proceedings of the 19th International Conference on Very Large Databases*, pages 146–157, Dublin, IR.

Raschid, L., Sellis, T., and Lin, C.-C. 1988. Exploiting Concurrency in a DBMS Implementation for Production Systems. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, pages 34–45, Austin, TX.

Reed, D. 1983. Implementing Atomic Actions on Decentralized Data. *ACM Transactions on Computer Systems*, 1(1):3–23.

Ryu, I. K. and Thomasian, A. 1990. Analysis of Database Performance with Dynamic Locking. *Journal of the Association for Computing Machinery*, 37(3):491–523.

Salem, K., Garcia-Molina, H., and Shands, J. 1994. Altruistic Locking. *ACM Transactions on Database Systems*, 19(1):117–164.

Sauer, C. and Chandy, K. M. 1975. Approximate Analysis of Central Server Models. *IBM Journal of Research and Development*, 19(3):301–313.

Schieber, B. and Vishkin, U. 1988. On Finding Lowest Common Ancestors: Simplification and Parallelization. *SIAM Journal of Computing*, 17(6):1253–1262.

Schwarz, P. M. and Spector, A. Z. 1984. Synchronizing Shared Abstract Data Types. *ACM Transactions on Computer Systems*, 2(3):223–250.

Sevcik, K. C. 1983. Comparison of Concurrency Control Methods using Analytical Models. In Mason, R. E. A., editor, *Information Processing*, pages 847–858.

Silberschatz, A. 1983. A Case for Non-Two-Phase Locking Protocols that Ensure Atomicity. *IEEE Transactions on Software Engineering*, 9(4):535–538.

Silberschatz, A. and Buckley, G. 1985. Beyond Two-Phase Locking. *Journal of the Association for Computing Machinery*, 32(2):314–326.

Silberschatz, A. and Kedem, Z. 1982. A Family of Locking Protocols for Database Systems that are Modeled by Directed Graphs. *IEEE Transactions on Software Engineering*, 8(6):558–562.

Silberschatz, A. and Kedem, Z. M. 1980. Consistency in Hierarchical Database Systems. *Journal of the Association for Computing Machinery*, 27(1):72–80.

Skarra, A. H. and Zdonik, S. B. 1989. Concurrency Control and Object-Oriented Databases. In Kim, W. and Lochovsky, F., editors, *Object Oriented Concepts, Databases, and Applications*, pages 395–421. ACM Press, New York.

Speele, P.-H. and Schouwenberg, M. 1994. First Practical Experiments with Random Plinius Terminological KBs in LOOM and CLASSIC. Technical Report KBS-Note-93-021, University of Twente.

Srinivasan, V. and Carey, M. J. 1993. Performance of B+ Tree Concurrency Control Algorithms. *The International Journal on Very Large Data Bases*, 2(4):361–406.

Tay, Y. C. 1987. *Locking Performance in Centralized Databases*. Academic Press, London.

Tay, Y. C., Goodman, N., and Suri, R. 1985. Locking Performance in Centralized Databases. *ACM Transactions on Database Systems*, 10(4):415–462.

Tay, Y. C., Suri, R., and Goodman, N. 1985. A Mean Value Performance Model for Locking in Databases: The No-Waiting Case. *Journal of the Association for Computing Machinery*, 32(3):618–651.

Thomasian, A. 1991. Centralized Concurrency Control Methods for High-End TP. *SIGMOD RECORD*, 20(3):106–115.

Ullman, J. 1988. *Principles of Database and Knowledge Base Systems, Volume 1*. Addison Wesley.

Wang, H. 1993. Personal Communication.

Weikum, G. and Hasse, C. 1993. Multi-Level Transaction Management for Complex Objects: Implementation, Performance, Parallelism. *The International Journal on Very Large Data Bases*, 2(4):407–453.

Welsh, P. D. 1983. Statistical Analysis of Simulation Output. In Lavenberg, S. S., editor, *Computer Performance Modeling Handbook*, pages 267–329. Academic Press, New York.

Yannakakis, M. 1982a. A Theory of Safe Locking Policies in Database Systems. *Journal of the Association for Computing Machinery*, 29(3):718–740.

Yannakakis, M. 1982b. Freedom from Deadlock of Safe Locking Policies. *SIAM Journal of Computing*, 11(2):391–408.

Yu, P. S., Dias, D. M., and Lavenberg, S. S. 1993. On the Analytical Modeling of Database Concurrency Control. *Journal of the Association for Computing Machinery*, 40(4):831–72.

Zahorjan, J. 1983. Workload Representation in Queueing Models of Computer Systems. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer System*, pages 70–81, Minneapolis.

Zdonik, S. B. and Maier, D., editors 1989. *Readings in Object-Oriented Databases*. Morgan Kaufmann Publishers, Inc., San Mateo, CA.

Index