Quantitative Evaluation of a Transaction Facility for a Knowledge Base Management System

Vinay K. Chaudhri Vassos Hadzilacos John Mylopoulos Kenneth C. Sevcik
{vinay, vassos, jm, kcs}@cs.toronto.edu
Department of Computer Science, University of Toronto,
Toronto, M5S 1A4, Canada

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. In this paper, we examine a novel concurrency control policy called Dynamic Directed Graph (or DDG) policy that effectively exploits the rich semantic structure of a knowledge base.

Our analysis is carried out in the context of a real knowledge based system application from which knowledge base structure and workload parameters are computed. These serve as a basis for studying the implementation alternatives that arise as a result of knowledge base characteristics. The implementation alternatives that we consider include selection of portions of the knowledge base structure to be exploited for concurrency control, and also the dependence of concurrency on the traversal strategy used to search through the knowledge base. We analyze the effects of various workload parameters and conclude that the DDG policy improves substantially the response time for short transactions when there is heavy data contention.

1 Introduction

As end user demands on database technology increase, so does the need for ever richer data models which, nevertheless, need to be implemented efficiently and must "scale up" to meet the challenge of ever larger and ever more complex applications. Unfortunately, richer data models, which adopt modeling features from object-oriented, deductive, temporal and spatial data models lead to complex databases with rich processing requirements involving sophisticated retrieval and inference procedures. For the purpose of this paper, we refer to databases with rich modeling and processing capabilities as knowledge bases.

One of the implementation issues that has received little attention for knowledge bases is concurrency control. Clearly, it is inefficient to lock out all but one user of a large, shared knowledge base. Even when the whole knowledge base fits in the main memory, concurrent processing is necessary to guarantee an acceptable response time to users [14]. What is needed is a concurrency control policy that exploits the organizational and semantic structure of knowledge bases to offer efficient and correct concurrent execution of transactions. This paper examines one such policy, called the Dynamic Directed Graph (DDG) policy [6, 7].

In this paper, we study the implementation alternatives that arise as a result of knowledge base characteristics and evaluate the performance of the DDG policy in comparison to two-phase locking (hereafter 2PL) [9]. We choose 2PL as a yardstick in evaluating the new policy because it is the best-known and the most-used concurrency control policy.

Our analysis is conducted through the implementations of the DDG and 2PL policies in the DeNet simulation environment [15], calibrated on the basis of knowledge base and workload parameters obtained from an actual knowledge base application. The workload consists of a mix of two classes of transactions, with Class 1 consisting of transactions that perform traversal along one of the several semantic relationships in the knowledge base and Class 2 containing transactions that involve lookup and update of attribute values and occasionally an update of some semantic relationships. Implementation alternatives that we consider include selection of portions of the knowledge base structure to be exploited for concurrency control, and also the dependence of concurrency on the traversal strategy used to search through the knowledge base. We analyze the effects of various workload parameters and conclude that the DDG policy reduces substantially (by about 50%) the response time for short transactions when there is heavy data contention.

The organization of the paper is as follows. In Section 2, we describe our model of knowledge bases and review the rules of the Dynamic Directed Graph policy. In Section 3, we describe an example application which we use in the performance study. In Section 4, we describe the simulation model and experimental set up. In Section 5, we discuss the performance results, and finally, we complete the paper in Section 6 with related work and conclusions.

2 Dynamic Directed Graph Policy

In this section, we first give our model of a knowledge base and then describe the rules of our locking policy.

For the purpose of this paper, knowledge bases are assumed to support an object-oriented representational framework with an assertional sub-language used for both deductive rules and constraints. Also, possibly, they might

To appear in the Proceedings of the ACM Third International Conference on Information and Knowledge Management, Nov 29 – Dec 2, 1994, Gaithersberg, MD.

support facilities for representing special kinds of knowledge (for example, temporal knowledge, incomplete knowledge, *etc.*). Examples of such systems include Telos [17], CLASSIC [5], CANDIDE [3] and several other deductive and object-oriented systems.

Given such a knowledge base, it is natural to view it as a directed graph. For example, if the knowledge base has a class `Employee` which is a specialization of class `Person` then both `Employee` and `Person` can be represented as nodes, and an edge between them would represent the `isA` relationship. Similarly, for a collection of deductive rules, one can generate a dependence graph in which each predicate is a node and there is an edge between two predicates A and B if B appears in the body of predicate A [26]. Such a dependence graph can be extended to deal with an object-oriented and temporal knowledge base such as Telos [20].

2.1 Model of a Knowledge Base

We assume that the knowledge base is a directed graph $G = (V, E)$, where V is a set of nodes (for example `Employee`), and E is a set of edges which are ordered pairs of nodes (for example, (`Employee, Person`)). For the purposes of this paper, we assume that we are given a graph representation of the knowledge base.

Transactions on such a knowledge base are a sequence of operations. Formally, an operation is a pair (`a` e), where `a` is an action (one of `INSERT, DELETE, READ, WRITE`), and e is an entity, which is a node or an edge. Using `INSERT` and `DELETE` operations, transactions can modify the graph. The transactions can acquire two kinds of locks – shared and exclusive, which have their usual semantics [4, 10, 19].

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 *entry point* of an SCC, G_i, is a node A such that there is an edge (A, B) in G, B is in G_i, but A 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 in the same strongly connected component as A. Thus, the root node dominates all the nodes in the graph including itself. All nodes in a strongly connected component dominate each other.

With this model, we give a brief description of our locking policy in the next sub-section. More details can be found elsewhere [6, 7].

2.2 Description of the Locking Policy

The Dynamic Directed Graph (DDG) policy has three types of rules: preprocessing rules, structure maintenance rules and locking rules. The preprocessing rules and the structure maintenance rules ensure that the graph is always rooted and connected; this is necessary for the correctness of the locking policy. These rules may some time require the addition of some artificial nodes to the graph, for example, when there is no root in the graph. The locking rules specify how the locks should be acquired. Because of space restrictions, we describe only the locking rules. We assume that the preprocessing rules have been applied to the graph and the resulting graph is rooted and connected. Let us now specify the locking rules.

Locking Rules

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. The first lock obtained by a transaction T can be on any node. If the first node locked by T belongs to a non-trivial SCC, all nodes on that SCC are locked together in the first step.

For each subsequent lock,
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 the graph 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 also 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, 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 $A_1, \ldots, A_p$.

Theorem 1 *The DDG policy always produces serializable schedules [6].*

Our locking policy is based on the tree policy (TP) [23]. The tree policy, when generalized to DAG policy [29], only deals with acyclic graphs which do not undergo change (that is, there are no insert and delete operations). The locking rules of the DDG policy (along with the preprocessing and structure maintenance rules that are not defined here) allow inserts and deletes in the graphs, and the graph can contain cycles.

The DAG policy, as considered in our paper, is different from the DAG locking that supports multiple granularities (MGDAG) [10, 11]. The MGDAG policy is a two-phase locking policy and supports multiple granularities of locking by using "intention" locks. In contrast, the DAG policy (as well as the DDG policy) is a non-two-phase locking policy, and at present, does not incorporate intention locks.

To be able 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.

Sometime transactions need to re-read data. In view of locking rule L3, locks on such data items are not released until the transaction is definite, that it is not going to read a data item in future. This assumption is easily enforced in knowledge bases when the transactions perform traversals along some semantic relationships.

In 2PL, a transaction must hold all the locks until it reaches its *locked point* – that is, the point when it has acquired all the locks that it will ever need. In fact, to ease recovery, most commercial systems use *strict* two-phase locking in which the locks are held until a transaction commits [4]. Under the DDG policy, locks need not be held until the locked point, thus permitting more concurrency than 2PL. This is an especially desirable property, because in a knowledge base traversal the transactions access a large number

Semantic Relationship	Number of Nodes/ Roots/Leaves	Fan In Max/ Avg	Fan Out Max/ Avg	Depth Max/ Avg	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

Transaction Type	Proportion
Class 1	25 Terminals
traversal along	
`isA+instanceOf`	62%
`linkedTo`	25%
`partOf`	13%
Class 2	25 alarms/s
Update	49.45%
Look up	49.45%
Change Relationship	0.10%

Table 1: (a) Structural properties of APACS KB (b) APACS workload

of data items and holding locks until the locked point significantly impairs the concurrency. Locking policies similar to the DAG policy have been extensively used to increase concurrency on index structures [16].

3 An Example Application

We tested the DDG policy on a knowledge base from the Advanced Process Analysis and Control System (APACS) that is being designed in collaboration with Ontario Hydro and CAE Electronics [18]. APACS is a prototype of a process control system intended to aid a plant operator during an emergency. It stores information about the components in the feedwater system, for example, boilers, valves, preheaters, alarms, etc. This knowledge base is used for alarm filtering and handles a total of approximately 2000 types of alarms and displays designed to assist the operators. 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 in 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 between them. For example, there is an edge from node A to node B, if the object represented by B is a part of the object represented by node A.

The most frequently occurring semantic relationships in this knowledge base are shown in Table 1(a). 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 sub-part relationship. And finally, the `Equipment` relationship associates a piece of equipment with each alarm.

The APACS workload, shown in Table 1(b), was obtained by interviewing the scientists who developed this system [27]. The knowledge base receives two classes of transactions. Transactions in the first class are generated because of the diagnostic and monitoring processes which consist of (partial) traversal along some semantic relationship. The distribution of traversals along different semantic relationships is shown in Table 1(b). The symbol "+" represents union of the graphs. Thus, `isA+instanceOf` represents the union of the graphs defined by the `isA` and `instanceOf` relationships. 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 sometimes they may also perform updates, with write probability varying between 0.0 and 0.5. (Write probability is the proportion of accesses performed by a transaction that are updates.) 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. Thus, the write probability for these transactions is fixed at 50%. 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 paper, we refer to the workload shown in Table 1(b) as the *baseline APACS* workload.

4 Simulation Model and Experimental Setup

We studied the performance of the DDG policy using a simulation model and compared it to strict 2PL. In this section, we describe our simulation model, measurement results and the simulation parameter settings.

4.1 Simulation Model

We model the concurrency control algorithm in detail while the rest of the components of the knowledge base management system (KBMS) are abstracted into just one component, called the transaction manager. The components of our model and their inter-connections are shown in Figure 1. Our model is based on an earlier model which is often used in the performance studies of concurrency control algorithms [2]. The parameters of the simulation model are shown in Table 2. Here we only highlight the differences between our model and the one used in the previous studies [2].

First, our workload consists of two classes of transactions, and we model Class 1 transactions as a closed system and Class 2 transactions as an open system. We decided to choose an open model for Class 2 transactions, because the arrival rate of Class 2 transactions (which are mostly alarms) does not depend on how many alarms have been already received.

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 rows in Table 2). The *InitCost* captures the cost of initiating a transaction. It includes the cost of creating entries in the transaction table and the cost of any preprocessing. The *LockCost* denotes the cost of a call to the lock manager. In case of the DDG policy, if a call also results in the release of some locks then this parameter will also include the cost of unlocking. The *CommitCost* denotes the cost of committing a transaction. It also includes the cost of releasing any remaining locks at commit time. The *AuxCost* is the run-time cost of maintaining auxiliary information which

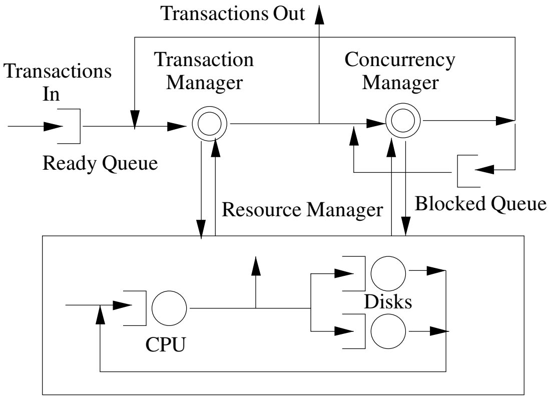

Figure 1: Performance model of the KBMS

Parameter	Meaning
NumTerminals	Number of Class 1 Terminals
ThinkTime	Think time for Class 1 terminals
ArrivalRate	Arrival rate of Class 2 transactions
UnitCPU	CPU time for processing an entity
diskIOTime	Time to perform one disk IO
HitRate	Buffer pool hit probability
TxnLength	Transaction Length
WriteProb	Write probability
InitCost	Cost of initiating a transaction
LockCost	Cost of Acquiring a lock
CommitCost	Cost of committing a transaction
AuxCost	Cost of updating graph properties

Table 2: Simulation model parameters

includes the cost of maintaining information on dominators and strongly connected components and is incurred only by the DDG policy.

Finally, we have defined the length of Class 1 transactions as the number of levels in the graph that a transaction traverses, 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 the more traditional definition where transaction length 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 Setup

We implemented 2PL and the DDG policies in the concurrency control manager of our simulation model. The implementation was done in the DeNet simulation environment on a DECStation 5000 (model 132) [15]. A detailed design of the implementation and sample source transactions are available elsewhere [6].

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 difference 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 by using the DDG policy instead of 2PL is computed as $(R(j)_{2PL} - R(j)_{DDG})/R(j)_{2PL} * 100$.

We employed a batch means method for the statistical data analysis of our results, and each simulation was run long enough to obtain sufficiently tight confidence intervals. The reported mean values are accurate within 5% at the 90% confidence level [13].

The parameter values in Table 2 were assigned values based on the APACS baseline workload. Class 1 ThinkTime was set to 1 sec. The values of the parameters UnitCPU and DiskIOTime were established based on the characteristics of the APACS application and were fixed at 5 msec and 10 msec respectively. Initially, HitRate was set to 0.0, and then was varied between 0.0 and 1.0 to study its influence on the results. Setting the HitRate to 1.0 represents the situation in which the knowledge base completely fits in main memory. We measured the concurrency control overheads from our implementation and gave them as parameters to the simulation model. Detailed measurement data can be found elsewhere [6].

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 insertion/deletion operations. The Class 1 transactions traverse from 3 to 6 levels. This means that the Class 1 transactions access, on an average, 130 data items, with the longest transaction accessing more than 1000 data items. The write probability for Class 2 transactions is fixed at 0.5 in all the experiments.

In all the experiments, we use a load control strategy in which we control the multiprogramming level of Class 1 transactions only. Class 2 transactions are processed as they arrive. We used this load control strategy, because our preliminary experiments showed that it leads to a better response time for Class 2 transactions as compared to a strategy in which the multiprogramming level of both Class 1 and Class 2 transactions is controlled [6]. As a result, the X axis in all the graphs refers to the multiprogramming level of Class 1 transactions.

5 Experimental Results

Our experiments can be classified under 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. 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. Workload parameters include: write probability, distribution of traversals along different semantic relationships, transaction length, think time of Class 1 transactions, arrival rate of Class 2 transactions and rate of insertions and deletions into the graph. The system parameters that we considered are buffer hit ratio, resource requirement and concurrency control overheads. In this paper, we restrict our presentation to only a subset of these experiments. A comprehensive description of the experiments is available elsewhere [6].

5.1 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. We do

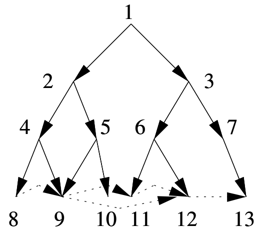

Figure 2: An example graph G

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

5.1.1 Choice of Semantic Relationship to be Used for Concurrency Control

As discussed in Section 3, 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.

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 with high frequency, the next question is whether we should consider its union with some other semantic relationships. This might ease the computation of dominator in some cases. (Recall that a transaction begins by locking the dominator of all the nodes that it expects to access.) In order to understand how this computation is affected by the choice of graph, let us first explain how the dominator is computed.

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*. In a *static implementation*, when knowledge base is compiled, with each node A, we associate another node B, that dominates all descendants of A. The 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 a *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.

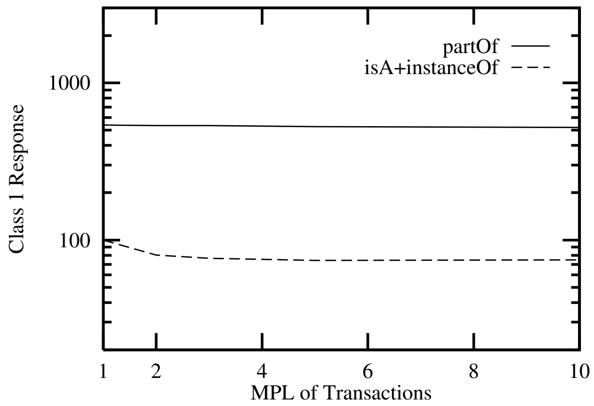

Figure 3: Use a graph that is traversed with high frequency

As an example, consider the graph G shown in Figure 2. 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 G_1 and the graph defined by type 2 edges G_2. 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 the graph G_2 up to 3 levels. If the graphs do not change in the meantime, T_1 accesses nodes 6, 11 and 12, and T_2 accesses nodes 8, 9, 11 and 12.

First, consider static implementation. If we use G for concurrency control, the dominator node for node 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, locking more nodes than it actually needs. 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. On the other hand, 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. If we use G for concurrency control, the node from which T_1 and T_2 should begin execution is accurately known, thus avoiding restarts, but some transactions need to lock more nodes than they actually need. On the other hand, if we use G_1 for concurrency control, the transactions that follow this graph need not lock more nodes than they need, but the transactions that do not follow it may need to restart. This shows an inherent tradeoff in the choice of the graph that is used for concurrency control: using G may require some transactions to lock more nodes than they need, whereas, using G_1 may lead to some aborts.

Now, consider a dynamic implementation. If we use G 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. 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_3 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, which is node 1, and therefore, T_2 begins by locking node 1 and avoids restarts. In this case, using G_1 has the advantage that T_1 did not have to lock more nodes than were necessary. Using G does not offer any advantage,

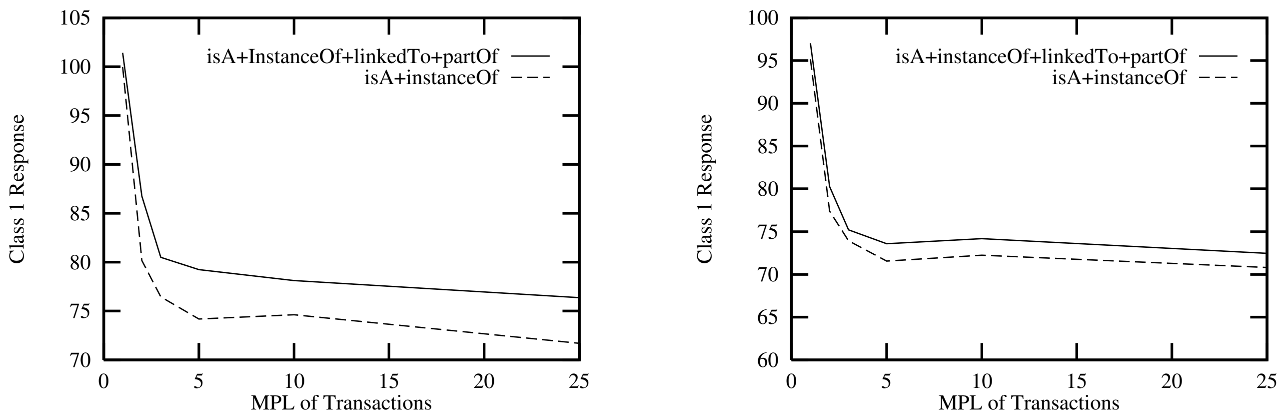

Figure 4: (a) Dynamic implementation (b) Static implementation

and 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` relationship. We compare the response times in Figure 3. We can see that using the graph defined by the union of `isA` and `instanceOf` leads to clearly better response times for Class 1 transactions. The same trend was observed for Class 2 transactions.

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 shown in Figure 4(a), we obtained lower response times for the case when we used the `isA` + `instanceOf` graph. (We show only Class 1 response times, as Class 2 response times were almost unaffected.)

Finally, we considered a static implementation and ran experiments for the same two cases as considered in the previous paragraph. Again, the `isA` + `instanceOf` graph led to lower response times. The response times are shown in Figure 4(b). On analyzing the response times, we found that the increased locking by using the `isA` + `instanceOf` + `linkedTo` + `partOf` far outweighed the aborts caused when we use `isA` + `instanceOf` graph. This shows that using more semantic information may not be always 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 choice is governed by the frequencies of traversal of various semantic relationships and the implementation used to compute the dominator information.

Since the rest of the experiments in this paper are based on a dynamic implementation, we decided to choose the union of `isA` and `instanceOf` as the graph for concurrency control for rest of our experiments.

5.1.2 Depth-first Traversal is Better Than Breadth-first Traversal

We consider two types of *transaction access patterns*: breadth-first search (BFS) and depth-first search (DFS). In 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, as compared to BFS, the DFS strategy facilitates an earlier release of locks and on average holds fewer locks. For example, consider the graph shown in Figure 2, 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` A) denotes a lock acquisition on node A in shared mode, (`LX` A) denotes lock acquisition on node A in exclusive mode, (`U` A) denotes release of lock on node A, and (`U *`) denotes release of all the locks that are currently held.)

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

If T_3 performs the same operations in a depth-first fashion, its execution is 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 *)
```

By comparing the above two schemes, we can see that by using the DFS, 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, and since node 2 is locked in exclusive mode, locks are not necessary for enforcing the locking rules of the DDG policy (in specific, locking rule L5b) for any nodes that T_1 locks in the future. Thus, there is an inherent tradeoff in the choice of the traversal strategy. The BFS strategy tends to release locks on the nodes that are closer to the root of the graph sooner, whereas, the DFS 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 DFS 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 DFS strategy has a greater potential of causing a bottleneck.

The above comments apply only if a transaction requests some exclusive locks. Had T_1 requested only shared locks, it could not have released any locks until the end of the transaction. This is because if it releases any lock before

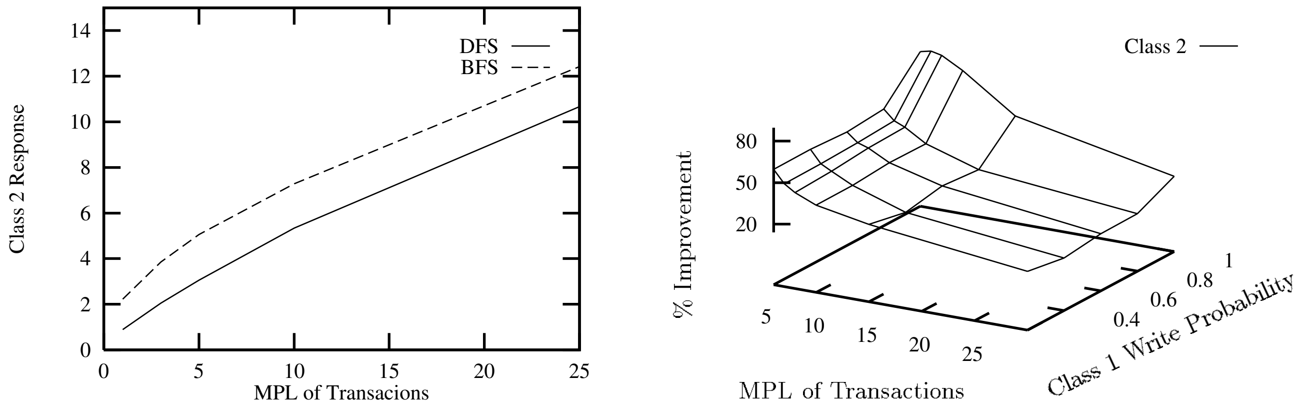

Figure 5: Choice of traversal strategy: (a) Absolute value (b) % Improvement

the locked point, it would not be able to satisfy the condition L5b for any of the nodes that it may lock later on. If all transactions access the data items in shared mode, the choice of traversal strategy is irrelevant.

In Figure 5(a), we show the response time of Class 2 transactions as Class 1 transactions use different traversal strategies. (We do not show Class 1 response times since they were almost unaffected.) 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 the DFS leads to much better response time for Class 2 transactions. To cross check the results, we compared the time spent by a transaction in different stages of execution. We found that under the BFS, the transaction spends more time in the blocked queue as compared to the time spent in the blocked queue under the DFS strategy.

Next, to see if DFS 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 DFS, Class 2 response times showed consistent improvement. In Figure 5(b) we show the percentage improvement in Class 2 response times as obtained by using DFS as compared to BFS. If $R(j)_{BFS}$ and $R(j)_{DFS}$ are the mean response times of transactions in the class j then the percentage improvement by using DFS instead of BFS is computed as $(R(j)_{BFS} - R(j)_{DFS})/R(j)_{BFS} * 100$. We can see that even at high write probabilities, by using DFS, Class 2 response times showed consistent improvement. These results suggest that the fact that DFS holds fewer locks at a given time has more influence on the response time than its holding locks on nodes near the root for a longer time. In fact, this indicates that in a multi-user environment, DFS is a more desirable traversal strategy than the BFS.

5.2 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 a depth-first traversal strategy. We vary other parameters one at a time and compare the performance of the DDG and the 2PL policies.

5.2.1 DDG Speeds up Class 2 Response at Heavy Data Contention

The write probability of transactions directly affects 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 locked point as compared to a situation in which it does not perform any updates. For example, consider transaction T_3 of Section 5.1.2. This transaction wishes to traverse the graph G_1 starting at node 1. If T_3 locks nodes 2 and 3 in exclusive mode, it can release some of the locks before its locked point – as shown in Section 5.1.2. 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 reaches its locked point (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 write probability: 0.0, 0.1, 0.2, 0.3 and 0.4 of Class 1. In Figure 6 we show the surface plots of the percentage improvements in response time over a range of probabilities between 0.0 and 0.4.

We can see that the mean response time for Class 1 transactions given by the DDG policy is higher than the one given by the 2PL by about 5% — which is not statistically significant. (Recall that the accuracy of the simulation is with in 5% of the mean.) For Class 2 transactions, there is almost no difference when write probability is 0.0, whereas when the write probability is 0.4, the response times given by the DDG policy are better by about 50%. On analyzing the Class 2 transaction response times, we found that at write probability 0.0, the time spent in the blocked queue by Class 2 transactions is comparable for both policies. At write probability 0.4, on the other hand, the time spent in the blocked queue by Class 2 transactions running under the DDG policy is much smaller. Interestingly, the variation in the time spent in the blocked queue had the same behavior as the response time indicating that the time spent in the blocked queue is the key determining factor for response time. Since in APACS, it is crucial to improve the response time of Class 2 transactions, this behavior gives the DDG policy an advantage over 2PL.

5.2.2 All Transactions do not have to Follow the Same Graph

The DDG policy performs 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, which is typical in many applications, 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 DDG policy to this effect.

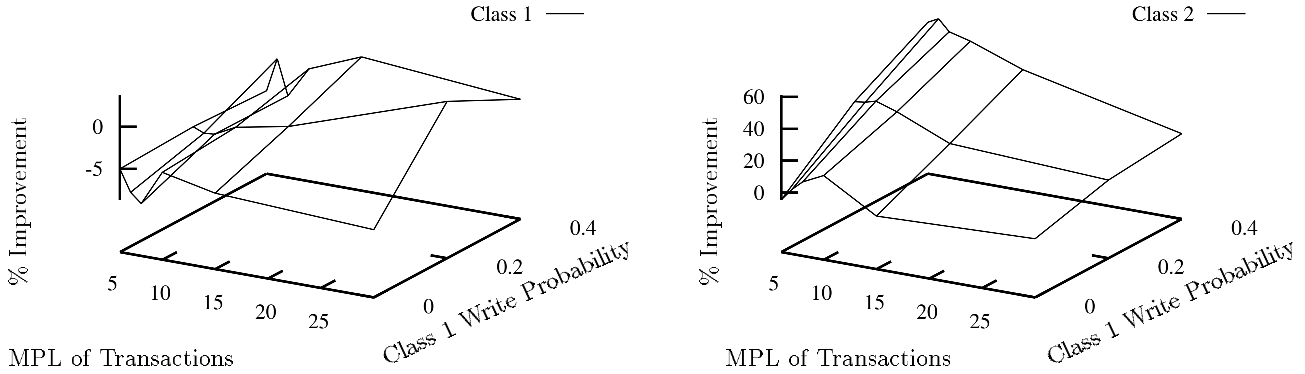

Figure 6: Comparison of DDG and 2PL – percentage improvements

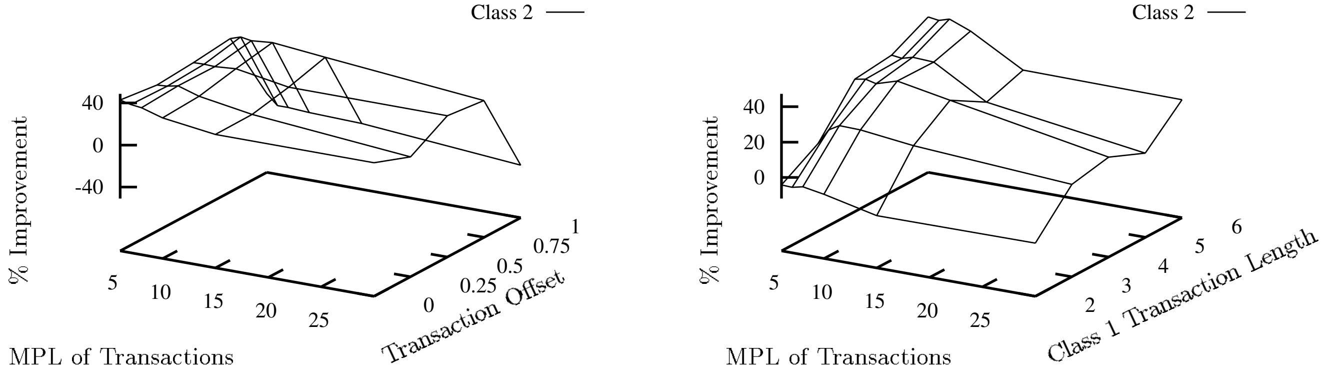

Figure 7: (a) Effect of transaction offset (b) Effect of transaction length

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 1(b)) 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 2, if we use G for concurrency control, then a transactions that accesses nodes 9 and 11 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 of the two policies is that the transaction offset can be computed based on the workload information, and thus gives a quick way of judging the benefits of the DDG policy. For example, for APACS, transaction offset can be computed from the workload information shown in Table 1.

To analyze this effect, we ran experiments for several values of transaction offsets between 0 and 100%. In Figure 7(a) we show the relative improvement in Class 2 response time for transaction offsets between 0 and 100%. We can see that even for high values of offsets, for example 50%, the DDG policy is able to give better response time for Class 2 transactions without seriously slowing down the Class 1 transactions (Class 1 response times are not shown here because the differences were not significant). But when the transaction offset is 100%, the performance of the DDG policy degrades. This degradation is due to excessive blocking caused by the skewed access patterns. 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.

5.2.3 Longer Transactions Lead to Better Improvements

In the baseline APACS workload, the transaction length varied between 3 and 6 levels. For example, for the graph G_1, consider 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 *)
```

In this transaction, even though we are able to release a lock before its locked point, we expect the benefits of using the DDG policy to be smaller as compared to the 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 7(a), we show the relative improvement in Class 2 expected response time as a function of Class 1 transaction length and multiprogramming level. From this we can see that when Class 1

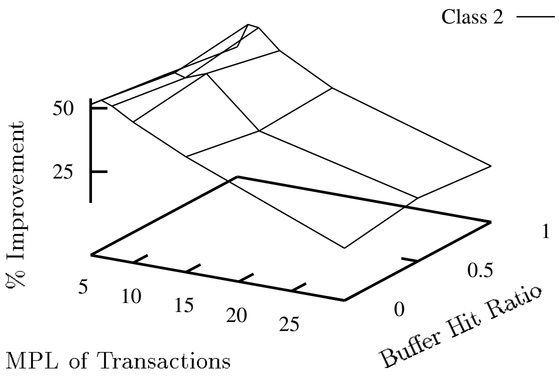

Figure 8: Effect of buffer hit ratio

transactions access only 2 levels, there is no difference between the two policies. When Class 1 transactions access 6 levels, the DDG policy improves the Class 2 expected response time by about 50%. This means that the DDG policy will be more suitable when the transactions traverse several levels in the graph.

5.2.4 Robustness of Results

In this section, we report the robustness of our results to the change in values of buffer hit ratio. The *buffer hit ratio* defines the proportion of times a desired data item can be found in the cache of the KBMS. By varying this parameter, we were able to model the situations ranging from the whole knowledge base fitting in main memory to the whole knowledge base residing on the secondary storage. Again, we re-ran the experiments of Section 5.2.1 with a hit ratio of 0.5 and 1.0. The relative improvements in Class 2 response time are shown in Figure 8. We can see that the relative improvements in the Class 2 transaction response time are maintained across different values of the buffer hit ratio. These results show 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.

6 Related Work and Conclusions

There have been several other proposals for locking policies that accommodate more concurrency than 2PL, for example, ordered shared locking and altruistic locking [1, 22]. These policies have potential for knowledge bases as well. However, they differ from the DDG policy in two important respects: the DDG policy is designed specifically for graph-structured databases and offers an application-dependent concurrency-control policy that can be tailored to specific applications using the type of analysis illustrated in this paper.

Similar comments apply to other performance studies of concurrency control algorithms, such as [2, 21, 25, 30] which do not assume a graph structured database and adopt a random access pattern model of database workload. Or for example studies of concurrency control algorithms for B-Trees in which the database is restricted to a tree [12, 24]. 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 match well processing requirements for knowledge based systems.

Performance of a multi-level implementation of the 2PL, with a detailed model of recovery, has been studied for complex objects [28]. 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 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.

The semantic structure of a knowledge base can be used in an alternative way to support multiple granularities of locking [8]. The emphasis in our work so far has been to use the semantic structure to allow release of locks before the locked point of a transaction.

The main contribution of this paper is the description of experimental evidence which suggests that the rich structure and processing needs of knowledge bases can be exploited to offer a more efficient concurrency control policy than 2PL. Evidence is produced on the the basis of a real knowledge based system application from which knowledge base structure and workload parameters are computed and serve as basis in exploring different implementation alternatives. These alternatives include selection of facets of the knowledge base structure to be used as the basis for concurrency control, also the relationship between the choice of traversal strategy and the amount of concurrency it permits. In addition, we analyze the effects of various workload parameters and conclude that the DDG policy improves substantially the response time for short transactions when there is heavy data contention.

In the future, we plan to extend this work in several directions. First, we plan to undertake an implementation of this algorithm in a knowledge base management system and to observe its behavior in a production setting. Second, we plan to consider other case studies with different knowledge bases and workloads to see if the same behavior is observed. Finally, we plan to look into the recovery issues to see which scheme would be the most appropriate for the DDG policy.

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 which fully addresses the modeling and performance requirements of future applications. The research approach in this paper is a step towards this integration goal.

Acknowledgments

This research was supported by the University of Toronto, the Information Technology Research Center of Ontario, the Natural Science and Engineering Research Council of Canada and the Institute of Robotics and Intelligent Systems. Several people have helped us in completing this work. We would like to thank Prof. G. Scott Graham for his advice on the simulation methodologies. Profs. Miron Livny and Prof. Michael J. Carey of University of Wisconsin (Madison) provided us the DeNet simulation environment and several useful suggestions for improving this work. Huaiqing Wang and Bryan Kramer of the APACS project provided the knowledge base and the workload used in this study. We also thank Profs. Dave Wortman and Songnian Zhou, for permitting us to use their machines, and to Jingwen Wang, for supporting the load sharing facility which was crucial for running the simulations. Finally, we thank the members of the KBMS group and the systems group for their support for this work.

References

[1] D. Agrawal and A. E. Abbadi. Locks with Constrained Sharing. In *9TH ACM Symposium on Principles of Database Systems*, pages 85–93, April 1990.

[2] R. Agrawal, M. J. Carey, and M. Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems*, 12(4):609–654, December 1987.

[3] H. W. Beck, S. K. Gala, and S. B. Navathe. Classification as a Query Processing Technique in the CANDIDE Semantic Data Model. In *IEEE International Conference on Data Engineering*, pages 572–581, 1989.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Welssley Publishing Company, 1987.

[5] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick. CLASSIC: A Structural Data Model for Objects. In *Proceedings of 1989 ACM SIGMOD International Conference on Management of Data*, pages 58–67, Portland, Oregon, 1989.

[6] V. K. Chaudhri. *Transaction Synchronization in Knowledge Bases: Concepts, Realization and Quantitative Evaluation.* PhD thesis, University of Toronto, Toronto, Forthcoming 1994.

[7] V. K. Chaudhri, V. Hadzilacos, and J. Mylopoulos. Concurrency Control for Knowledge Bases. In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning*, pages 762–773, 1992.

[8] F. de Ferreira Rezende and T. Härder. A Lock Method for KBMSs Using Abstraction Relationships' Semantics. In *Proceedings of the Third International Conference on Knowledge Management (to appear)*, 1994.

[9] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in Database Systems. *Communications of the ACM*, 19(9):624–633, 1976.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, Inc., San Mateo, 1993.

[11] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Database. In G. M. Nijssen, editor, *Modeling in Database Management Systems*, pages 365–394. North Holland Publishing Company, 1976.

[12] T. Johnson and D. Shasha. The Performance of Current B-Tree Algorithms. *ACM Transactions on Database Systems*, 8(1):51–101, March 1993.

[13] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis.* McGraw-Hill New York, NY, 1991.

[14] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis using Queueing Network Models.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

[15] M. Livny. DeNeT User's Guide (Version 1.5). Technical report, University of Wisconsin, 1990.

[16] C. Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transaction Operating on B-Tree Indexes. In *Proceedings of 16th VLDB Conference*, pages 392–405, Brisbane, Australia, 1990.

[17] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: A Language for Representing Knowledge About Information Systems. *ACM Transactions on Information Systems*, 8(4):325–362, October 1990.

[18] J. Mylopoulos, B. Kramer, H. Wang, M. Benjamin, Q. B. Chou, and S. Mensah. Expert System Applications in Process Control. In *Proceedings of the International Symposium on Artificial Intelligence in Materials Processing Applications*, Edmonton, August 1992.

[19] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, Rockville, MD, 1986.

[20] D. Plexousakis. Integrity Constraint and Rule Maintenance in Temporal Deductive Knowledge Bases. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 146–157, Dublin, IR, August 1993.

[21] I. K. Ryu and A. Thomasian. Analysis of Database Performance with Dynamic Locking. *Journal of the Association for Computing Machinery*, 37(3):491–523, July 1990.

[22] K. Salem, H. Garcia-Molina, and J. Shands. Altruistic Locking. *ACM Transactions on Database Systems*, 19(1):117–164, 1994.

[23] A. Silberschatz and Z. M. Kedem. Consistency in Hierarchical Database Systems. *Journal of the Association for Computing Machinery*, 27(1):72–80, 1980.

[24] V. Srinivasan and M. J. Carey. Performance of B+ Tree Concurrency Control Algorithms. *The International Journal on Very Large Data Bases*, 2(4):361–406, October 1993.

[25] Y. C. Tay. *Locking Performance in Centralized Databases.* Academic Press, London, 1987.

[26] J. Ullman. *Principles of Data Base and Knowledge Base Systems, Volume 2.* Addison Wesley, 1989.

[27] H. Wang. Personal Communication, 1993.

[28] G. Weikum and C. Hasse. Multi-Level Transaction Management for Complex Objects: Implementation, Performance, Parallelism. *The International Journal on Very Large Data Bases*, 2(4):407–453, October 1993.

[29] M. Yannakakis. A Theory of Safe Locking Policies in Database Systems. *Journal of the Association for Computing Machinery*, 29(3):718–740, July 1982.

[30] P. S. Yu, D. M. Dias, and S. S. Lavenberg. On the Analytical Modeling of Database Concurrency Control. *Journal of the Association for Computing Machinery*, 40:831–72, September 1993.