Efficient Algorithms and Performance Results for Multi-User Knowledge Bases

Vinay K. Chaudhri and John Mylopoulos
Department of Computer Science,
University of Toronto Toronto, Ontario M5S 1A4

January 6, 1995

Abstract

The paper describes research efforts to develop efficient implementation techniques for large, shared knowledge bases, focusing on efficient concurrent access of large knowledge bases by multiple users.

We present an algorithm, called the Dynamic Directed Graph policy, originally proposed in [CHM92] which allows efficient interleaved execution of transactions against a large knowledge base with the intent of optimizing transaction throughput. The implementation of the policy and experimental evaluation results are also presented and discussed. The paper concludes with discussion on lessons learnt from this research.

Large knowledge bases containing millions of facts will soon be here, thanks to research efforts such as the Knowledge Sharing initiative [PFPS+92] and the CYC project [GL94]. However, tools for building knowledge bases do not scale up to accommodate such large knowledge bases. Our efforts are focusing on the adoption of database techniques to build knowledge base building tools that do scale up.

One of the requirements on such tools is that they accommodate efficient multi-user access of a single, large knowledge base by maximizing throughput, i.e., the number of user-defined transactions that are executed against the knowledge base per time unit. A comparable requirement for databases is addressed by concurrency control mechanisms that are routinely offered by database management systems, which have been shown to improve throughput by as much as an order of magnitude or more.

A comparable concurrency control algorithm, specifically designed for knowledge bases and called the Dynamic Directed Graph policy was proposed in [CHM92]. The main purpose of this paper is to describe an implementation of that policy and to present original performance results which compare the performance of the proposed policy against the performance of off-the-shelve concurrency control mechanisms designed for databases. The paper summarizes some of our findings and concludes with research directions.

The outline of this paper is as follows. In Section 1, we begin by motivating the problem. In Section 2, we present the Dynamic Directed Graph (DDG) policy. In Section 3, we briefly describe some problems that were faced while implementing the policy. In Sections 4-5, we present the evaluation of the algorithm for knowledge base applications, and in Section 6, discuss some related work. In Section 7, we discuss the lessons learnt from our research and conclude the paper in Section 8 with a summary.

1 Problem Statement

Concurrent processing of user requests can lead to large speed ups in user response time as compared to processing the requests of one user at a time [GR93]. Arbitrary concurrency can, however, lead to inconsistent information in the knowledge base and one must make sure that the concurrent executions are *serializable* [BHG87]: interleaved execution of a set of transactions must be equivalent to some serial execution of the same collection of transactions in the sense that it leaves the knowledge base in the same state and returns the same answers to the users.

Most commercial database systems use *locking-based* algorithms to ensure serializability. The best known locking algorithm, *two-phase locking* (2PL) [EGLT76], works along the following lines. Associated with each data item is a distinct "lock". A transaction must acquire a lock on a data item before accessing it. While a transaction holds a lock on a data item no other transaction may access it. A transaction cannot acquire any additional locks once it has started releasing locks (hence the name "two-phase" locking). In a simple generalization of this model, the transactions may hold *shared* and *exclusive* locks on data items.

Transactions in a knowledge base system often access large number of data items, for example, while proving a goal by backward chaining, a transaction may potentially access all the nodes that are descendants of that goal in the inference graph. In such situations, 2PL implies that a transaction will have to hold each lock until it finishes acquiring all the locks that it will ever need, thereby locking most of the knowledge base for other users. Hence, concurrency is significantly reduced when running such "global" transactions. For this reason, our research has been directed towards the development of new methods that only hold a small number of locks at any one time, even for global transactions.

Interestingly, knowledge bases generally possess much richer internal structure (e.g., generalization and aggregation hierarchies, deductive rules, temporal dimensions defined in terms of history or belief time, etc.) than that of traditional databases. Information about this structure can be potentially useful in allowing early release of locks. Indeed, a concurrency control algorithm does exist for databases that have a directed acyclic graph structure, and is accordingly called *DAG policy* [SK80, Yan82]. Under the DAG policy, a transaction may begin execution by locking any item. Subsequently, it can lock an item if it has locked all the predecessors of that item in the past and is currently holding a lock on at least one of those parents. Moreover, the DAG policy may only lock an item once. The DAG policy exploits the assumption that there are no cycles in the underlying structure and the structure does not undergo any change. Unfortunately, such a policy cannot be adopted for knowledge bases without modifications. The structure of a knowledge base is likely to contain cycles (e.g., the inference graph generated for a collection of recursive rules) and will undergo change (e.g., when rules are added or deleted).

In summary, neither 2PL nor DAG policy is, by itself, appropriate for knowledge bases. 2PL is too conservative, thereby causing reduced concurrency, while DAG policy do not provide sufficient functionality. Accordingly, we are proposing a new graph-based policy, the *Dynamic Directed Graph policy* (DDG) that can handle cycles and updates in the knowledge base and also allows release of locks before a transaction commits, thereby promising better performance than 2PL.

2 The Dynamic Directed Graph Policy

The kind of knowledge bases that we have in mind 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 support facilities for representing special kinds of knowledge (for example, temporal knowledge, incomplete knowledge, *etc.*). A large class of knowledge bases can be represented in terms of such directed graphs, including ones based on semantic nets, frames but also description logics or even logics [Ple93, BPS94]. Therefore, for the purposes of concurrency control, a knowledge base is a directed graph $G(V, E)$, where V is a set of nodes (e.g., Employee), and E is a set of edges which are ordered pairs of nodes (e.g., (Manager,Employee)).

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.

The DDG policy has three types of rules. *Preprocessing rules* convert an arbitrary graph to a rooted and connected graph. *Locking rules* specify how each transaction should acquire locks. *Maintenance rules* specify additional operations that must be executed by transactions to keep the structure rooted and connected. The rest of the discussion in this section focuses on locking rules. A detailed description of the DDG algorithm appears elsewhere [Cha94].

A transaction may lock a node in *shared* or *exclusive* mode [BHG87]. Two transactions may simultaneously lock a node only if both lock it in shared mode. The locking rules for the DDG policy are as follows:

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 T at most once.
L4. The first lock obtained by 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.

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 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 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 knowledge base and the transactions shown in Figure 1. (`LS` and `LX` respectively denote the acquisition of lock in shared and exclusive mode, `U` denotes release of lock on a node and (`U` *) denotes the release of all the locks held by a transaction.) T_1 begins by locking node 1 (locking rule L4) in shared mode and then locks node 2 in exclusive mode (locking rule L5). It locks the nodes 3 and 4 which form a strongly connected component in shared mode in one step (locking rule L5). It is able to do so because the condition L5b is satisfied for each path from node 2 to node 3 and 4. T_1 locks node 5 in exclusive mode and finishes

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

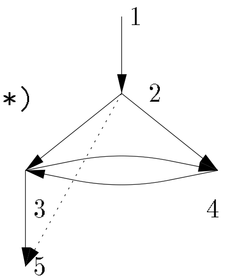


Figure 1: An Example Application of the DDG policy

execution. T_2 begins by locking both nodes 3 and 4 (locking rule L4) and then locks node 5 and finishes execution. If T_1 adds the edge $(2, 5)$ (locking rule L1), then T_2 will be unable to lock node 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 (locking rule L5a). T_2 must abort and start from node 2.

Theorem: The DDG policy produces only serializable schedules [Cha94].

In general, the DDG policy does not permit concurrency within cycles (see rule L5 above). This suggests that if a knowledge base contains very large cycles which need to be locked as one node, concurrency will be reduced. We have a version of the DDG policy that does permit concurrency within cycles [CHM92]. We adopted the above version, because the transactions in knowledge bases tend to access all the nodes on a cycle together, and therefore, the cycles are a natural unit of locking.

For a transaction to be able to satisfy locking rule L5 for all the nodes that it needs to lock, it has to begin by locking the dominator of all the nodes that it is going to access. This is not a contradiction to locking rule L4, which just says that to lock first node, no other condition needs to be satisfied.

3 Implementation of the DDG Policy

The DDG policy has been implemented in the DeNet [Liv90] simulation environment. The implementation ideas that we present here are independent of any specific system.

There are two main issues in the implementation of the DDG policy. First, to enforce the rules of the locking policy, we need to compute and maintain information about several graph properties. Second, we need a mechanism to decide the order in which the locks should be acquired and released.

To enforce the locking rules, we need information on the dominator relationships and the strongly connected components within the knowledge base graph. In our implementation, the dominator tree of the knowledge base is computed at compile time using a bit vector algorithm [ASU86]. Using this information, the dominator of the set of nodes in the transaction can be computed in time linear in the length of a transaction. The dominator information is maintained using an incremental algorithm [Car88]. The information on strongly connected components is computed at compile using depth-first search. There was no algorithm available for incrementally maintaining information on strongly connected components as the knowledge base evolves, and therefore, we developed an algorithm for this purpose [Cha94].

Let us now describe the order in which a transaction acquires and releases locks. A transaction always begins by locking the dominator of all the nodes that it might access. The dominator is computed on the assumption that a transaction may access all the descendants of the first node on which it requests a lock. Subsequently, every time a lock is requested, the locking conditions are checked, and if not enough predecessors are locked (see rule L5a), lock requests for them are issued recursively. Before a node A can be unlocked by a transaction T, following conditions must be satisfied:

U1. A is no longer needed by T, and
U2. Releasing the lock on node A does not prevent the locking of any of its successors at a later stage in the execution of T (as required by rule L5a), and
U3. 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).

To implement U1, we require T to send a message to the lock manager every time it has finished processing a node. This condition is automatically satisfied for the nodes that are not required by T, but are locked to satisfy the rules of the locking policy.

To implement U2, we have to know how many of the descendants might be later locked by T. Moreover, of all the predecessors of a node A, only one has to be kept locked until T locks A. Therefore, we distinguish one of the predecessors that needs to be locked until all the successors have been locked, and associate with it the number of successors that are yet to be locked. Once the number of successors yet to be locked for a node becomes zero, U2 is satisfied.

To implement U3, we check all the undirected paths from A to all the nodes that T may lock in future. U3 needs to be checked only when T acquires an exclusive lock or when T locks a node none of whose descendants will be locked by it in future. The check can be made more efficient by observing that if U3 is not satisfied for a node A, it is also not satisfied for descendants of A, and therefore, they need not be checked. More precise description of this process is available elsewhere [Cha94].

4 Evaluation of the DDG Policy

Our performance model is similar to an earlier model [ACL87] and 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 database.

Performance of the DDG policy was studied on a knowledge base under development for industrial process control [MKW+92]. The application is called Advanced Process Analysis and Control System (APACS). 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. There are several semantic relationships in this knowledge base. The `isA` relationship captures the 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.

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.

The knowledge base receives two classes of transactions. Class 1 transactions are long and traverse the knowledge base along one of its structural relationships. Class 2 transactions are short and look-up or update an attribute value and occasionally change the structural relationships in the knowledge base. The proportion of the transactions in Class 1 was determined to be 27% with the remaining 73% being in Class 2.

Since there are several semantic relationships in the knowledge base, we can selectively use one or more of these as the graph to be used for concurrency control. Making a semantic relationship known to concurrency control gives it information on the entities to be accessed by transactions that traverse that relationship, but on the other hand, for transactions that do not traverse this relationship, it may mean locking more nodes than they actually require. We analyzed this tradeoff [Cha94] and concluded that for the APACS workload, using the graph defined by the union of `isA` and `instanceOf` relationships leads to the best performance, and therefore, used it as the graph with respect to which the locking rules of the DDG policy were applied.

In the APACS application, Class 1 transactions do not specify the traversal strategy that should be used, and therefore, one can choose a strategy that leads to better performance. We considered two traversal strategies: depth-

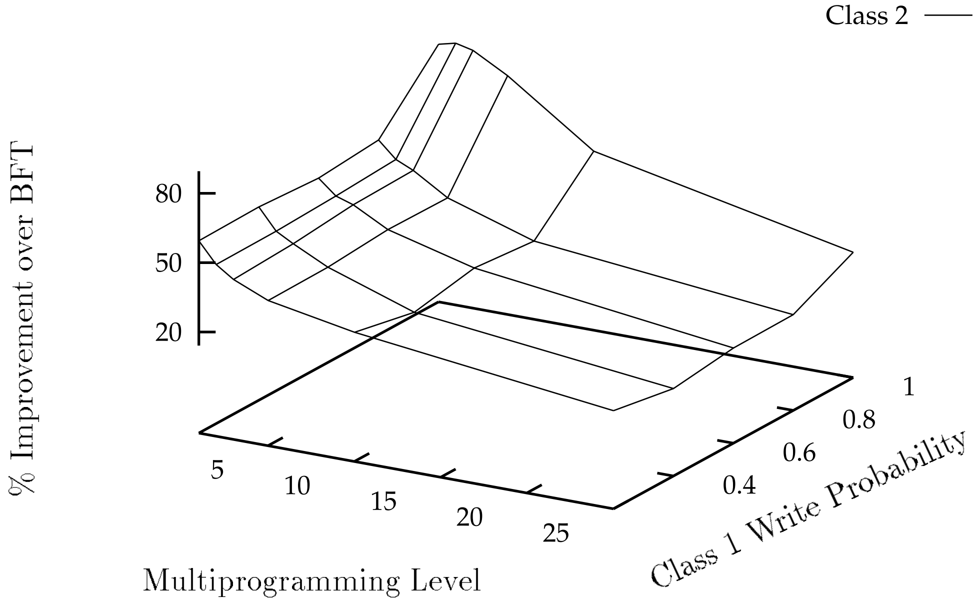
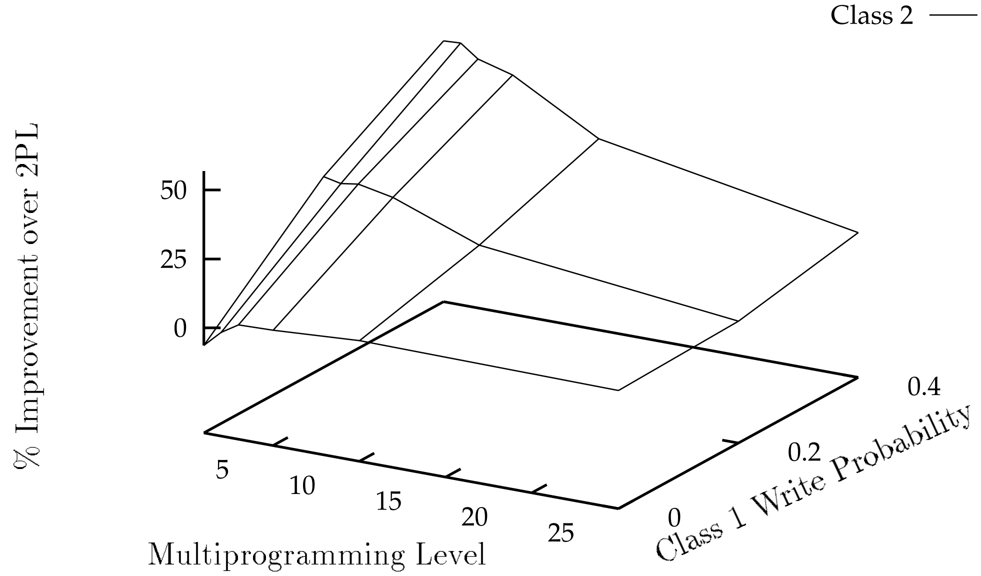


Figure 2: Choice of traversal strategy: relative response time

first traversal (DFT) and breadth-first traversal (BFT). In general, transactions using a DFT hold a lesser number of locks on average as compared to the average number of locks held while using a BFT. (Examples are available elsewhere [Cha94].) To quantify the difference, in Figure 2, we show the percentage improvement in the Class 2 response times as obtained when Class 1 transactions use DFT as compared to BFT. We do not show the improvement in Class 1 response time as it was found to be insignificant. Write probability denotes the proportion of requests of Class 1 transactions that are updates. 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. This result shows that in a multi-user knowledge base environment, while using the DDG policy, DFT is a more desirable traversal strategy than BFT.

In Figure 3, we plot the percentage improvement in response time of Class 2 transactions obtained by using the DDG policy as compared to two-phase locking. 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 $100 \times (R(j)_{2PL} - R(j)_{DDG})/R(j)_{2PL}$.

The results for Class 2 transactions indicate that when Class 1 transactions are read only, the performance of the DDG policy is comparable to 2PL. When Class 1 transactions also perform some updates, the DDG policy can improve considerably the response time of Class 2 transactions that are running concurrently. The results for Class 1 transactions are not shown here. We found that at low update probabilities there was slight degradation (about 10%) in Class 1 response time by using the DDG policy and at high update proba-



(Positive improvement indicates that DDG performs better than 2PL.)
Figure 3: Comparison of DDG and 2PL – percentage improvements

bilities there was no significant difference between the two algorithms. These results make sense, because when there are only shared locks in a transaction, the DDG policy cannot allow any release of locks before it commits, and therefore, it performs comparably. On the other hand, if the transactions are update intensive, the extra overhead is more than offset by the increased concurrency obtained due to lock pre-release. A more comprehensive description of experiments may be found elsewhere [Cha94].

In view of the above results, we designed an adaptive scheme which can switch between 2PL and the DDG policy depending on the load conditions. Such a scheme uses 2PL at low write probabilities and the DDG policy at high write probabilities, thus giving the best of the two algorithms. Since simultaneously using the two algorithms may lead to non-serializable schedules the adaptive scheme uses a transition phase while switching from one algorithm to the other. While switching from 2PL to the DDG policy, transactions locked according to the DDG policy are forced to behave as two-phase locked transaction by delaying the release of lock before commit time. While switching from the DDG to 2PL, the adaptive scheme waits until all active transactions running under the DDG policy finish execution. This results in an algorithm which is a hybrid of 2PL and the DDG policy and performs consistently better than a concurrency control algorithm that only uses 2PL.

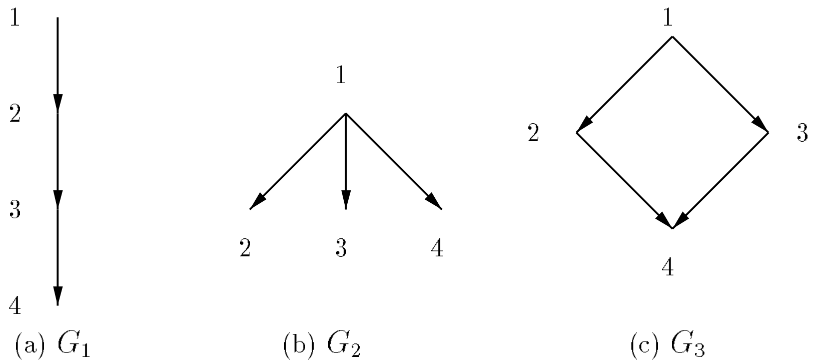


Figure 4: Distinguishing the structures in graphs

5 Desired KB Graphs for Better Performance

Since the DDG policy exploits the semantic structure of a knowledge base, its performance is sensitive to the properties of the knowledge base graph. To understand this, consider the directed graphs shown in Figure 4, which have the same number of nodes. Suppose, that a transaction T performs traversal along each graph and locks all entities in exclusive mode. In case of G_1, once a node (1, 2 or 3) has been processed by T, lock on it has to be held until we have locked its only 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, in case of G_1, lock holding time on node 1 is likely to be lower as compared to lock holding time in case of G_2. This effect can be captured by measuring the fanout of the graph. Furthermore, in G_1 and G_2, to lock a node which is not the first node locked by T, we need to lock only one other node, whereas in G_3, to lock node 4, we need to lock both nodes 2 and 3. In this respect, the structure of graph G_3 is less desirable than G_1 or G_2. This effect can be captured by measuring the fanin and analyzing the dominator tree of the knowledge base.

We formalized above intuitions by defining *fanin* factor, *fanout* factor and *depth* factor. Fanin (fanout) factor is the ratio of the maximum fanin (fanout) to the knowledge base size. Smaller the value of this ratio, greater are the benefits of using the DDG policy as compared to 2PL. Thus, a desirable graph for the DDG policy would be the one with fanin and fanout equal to 1 and an undesirable graph would be with fanin (or fanout) equal to $n-1$ where n is the number of nodes in the graph. Depth factor measures the closeness of the knowledge base graph to a tree. Before we define depth factor, we need following definitions. The dominator tree of a 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. Let $d(A, G)$ denote the length of the shortest path from the root of G to A. Then the depth factor of a knowledge base can be defined as follows:

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

If the knowledge base is a tree, the numerator is equal to zero, giving a depth factor equal to 1. A depth factor value of 1 is most desirable because then transactions lock exactly the same number of entities that they need. Closer the value of depth factor to 1, higher is the potential for obtaining improvement in performance using the DDG policy. Fanin, fanout and depth factors taken together can give some insight into the desirability of a graph if one wants to exploit semantic structure using the DDG policy.

As an example application of the above ideas, we considered two other knowledge bases in addition to APACS. The first of these, called Wines, is a description logic knowledge base developed at AT&T Bell Labs. The second knowledge base, called Tove, is a knowledge base developed at University of Toronto by the Enterprise Integration group. For the APACS, Tove and Wines knowledge bases the values of fanout factor are respectively, 0.09, 0.103 and 0.24, values of fanin factors are respectively, 0.0003, 0.0004 and 0.001 and values of depth factors are respectively 1.0, 0.975 and 0.887. Based on these metrics one would predict that the improvement in performance would be the highest for APACS, followed by Tove and Wines. Interestingly, our experiments showed that for a workload which was similar in every respect, the improvement in performance was the highest for the APACS knowledge base, followed by Tove knowledge base, and finally, the Wines knowledge base.

The results presented in this section are empirical and should not be taken to imply that they will work for every knowledge base application. Using fanin, fanout and depth factors, we were able to intuitively and experimentally explain the differences in relative performance on three knowledge bases that were considered. For any other application, these metrics can be a useful starting point in judging the suitability of the DDG policy.

6 Related Work

The present paper summarizes and enhances the results published earlier which focused on details of correctness [CHM92] and performance [CHMS94] issues. A comprehensive description of the results with all details can be found in the first author's doctoral dissertation [Cha94].

Knowledge base implementations that have tried to address the issue of concurrent access to a shared knowledge base include CYCL [GL94], K-REP [MLDW91] and PROTEUS [BCG+88]. In CYCL, a copy of the whole knowledge base is made and given 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. The K-REP system uses similar ideas and allows users to work on different versions of the knowledge base and provides facilities for merging the versions. The basic assumptions in the CYCL and K-REP 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. The CYCL and K-REP 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. PROTEUS system uses the concurrency control facilities of an underlying object-oriented system ORION. The concurrency control mechanism in ORION is 2PL with locking granularities defined based on aggregation and generalization hierarchies.

7 Lessons Learnt

An interesting aspect of our results is the integration of knowledge model features with low level implementation issues. The integration is reflected in the design of the DDG policy that abstracts the knowledge model into a directed graph, and in performance evaluation which is done as a function of data model features.

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. This indicates that if there are large cycles in the knowledge base that are repeatedly accessed by a transaction then concurrency will be limited. Similarly, we found that a depth-first traversal strategy led to a better response time as compared to a breadth-first strategy. 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.

In addition, we found that in order to get better advantage of the semantic structure, the knowledge base should be highly structured: it should be almost like a tree (high depth factor) and should not have large fanin and fanout (low fanin and fanout factors). This has an interesting implication for knowledge base implementations that pre-compute all subsumption relationships. Pre-computation of all subsumption relationships is equivalent to creating a large fanin and fanout in the subsumption hierarchy and in such a situation the advantages of using the semantic structure using the DDG policy will be reduced.

Our research approach also illustrates a general framework for incorporating database functionality into knowledge bases. The solution space to achieve this includes [MB90]: coupling an AI system with a database system (loose or tight coupling) or devising an integrated solution (in an evolutionary or a revolutionary way). We adopt and recommend an evolutionary approach, because it is pragmatic and consistent with software reusability.

From our research, we have learnt that an evolutionary approach to design an integrated AI-DB solution should be a two step process. In the first step, one should view the problem as a database tuning problem [Sha94], and if necessary, one should take the second step and solve a database kernel design problem. By solving a kernel design problem we mean addressing a core database concern such as storage design or query optimization. Let us explain this in more detail.

Let us illustrate the above philosophy by considering the problem of concurrent access addressed in the present paper. To support concurrent access to a knowledge base, one would start with a database solution which in this case is 2PL. To tune 2PL for long transactions, one would chop the long transactions into smaller transactions [Sha94] or impose a correctness criterion which is weaker than serializability. If the long transaction problem still persists one would change the database kernel and augment 2PL with the DDG policy and use an adaptive algorithm that consistently gives a better performance. Thus, these two steps taken together would give a comprehensive integrated solution for concurrent access in a knowledge base environment.

In summary, we feel that to develop a technology for constructing knowledge bases, we need to address the core database issues for knowledge models. A subset of these concerns may be addressed by tuning existing database products but a long term solution would require an integrated approach that makes fundamental changes in the database kernel.

8 Summary and Conclusions

In this paper, we considered the problem of supporting concurrent access to large knowledge bases. We argued that knowledge bases have the problem of long transactions which cannot be efficiently solved by existing database techniques, and that we can use the rich semantic structure of knowledge bases to devise more viable solution. We showed this by presenting the design, implementation and evaluation of an algorithm called Dynamic Directed Graph (DDG) policy that gives better response time than two-phase locking at high update rates. We showed that 2PL can be augmented with the DDG policy to give a hybrid algorithm that consistently performs better than a system that uses only 2PL. We also discussed the impact of our results on the knowledge base design and presented a refinement of the evolutionary paradigm for constructing knowledge base management systems. In our current work, we are addressing the issues of fault-tolerance for the DDG policy.

In conclusion, we would like to note that the results presented in this paper are at the intersection of knowledge base and database systems. Research combining techniques from these two fields will be of prime importance in future. One of the major technological innovations in the coming years would be development of cooperative information systems involving a large number of intelligent agents distributed over computer/communication networks. Developing such systems presents a unique opportunity in which techniques from both knowledge bases and databases will play a crucial role. The results presented in this paper make a modest contribution towards this goal.

References

[ACL87] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems*, 12(4):609–654, December 1987.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[BCG+88] Nat Ballou, Hong-Tai Chou, Jorge F. Garza, Won Kim, Charles Petrie, David Russinoff, Donald Steiner, and Darrell Woelk. Coupling an Expert System Shell With and Object-Oriented Database System. *Journal of Object Oriented Programming*, 1(2):12–21, 1988.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Welssley Publishing Company, 1987.

[BPS94] Alex Borgida and Peter F. Patel-Schneider. A Semantics and Complete Algorithm for Subsumption in CLASSIC Description Logic. *Journal of Artificial Intelligence Research*, 1:277–308, 1994.

[Car88] M. D. Caroll. Data Flow Analysis via Dominator and Attribute Updates. Technical Report LCSR-TR-111, Rutgers University, May 1988.

[Cha94] Vinay K Chaudhri. *Transaction Synchronization in Knowledge Bases: Concepts, Realization and Quantitative Evaluation.* PhD thesis, University of Toronto, Toronto, 1994. Available by WWW URL ftp://ftp.cs.toronto.edu/pub/kbms/TSynchronization.ps.Z.

[CHM92] Vinay K Chaudhri, Vassos Hadzilacos, and John Mylopoulos. Concurrency Control for Knowledge Bases. In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning*, pages 762–773, 1992.

[CHMS94] Vinay K Chaudhri, Vassos Hadzilacos, John Mylopoulos, and Ken Sevcik. 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, 1994.

[EGLT76] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The Notions of Consistency and Predicate Locks in Database Systems. *Communications of the ACM*, 19(9):624–633, 1976.

[GL94] R. V. Guha and Douglas B. Lenat. Enabling Agents to Work Together. *Communications of the ACM*, 37(7):126–142, 1994.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, Inc., San Mateo, 1993.

[Liv90] Miron Livny. DeNeT User's Guide (Version 1.5). Technical report, University of Wisconsin, 1990.

[MB90] John Mylopoulos and Michael Brodie. Knowledge Bases and Databases: Current Trends and Future Directions. In *Lecture Notes in Computer Science, Vol. 474: Information Systems and Artificial Intelligence: Integration Aspects*, New York, 1990.

[MKW+92] 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.

[MLDW91] E. Mays, S. Lanka, B. Dionne, and R. Weida. A Persistent Store for Large Shared Knowledge Bases. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):33–41, 1991.

[PFPS+92] Ramesh Patil, Richard E. Fikes, Peter F. Patel-Schneider, Don Mackay, Tim Finin, Thomas Gruber, and Robert Neches. The DARPA Knowledge Sharing Effort: Progress Report. In *The Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 777–788, Boston, MA, 1992.

[Ple93] D. Plexousakis. 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, August 1993.

[Sha94] Dennis E. Shasha. *Database Tuning — A Principled Approach.* Prentice Hall, NJ, 1994.

[SK80] A. Silberschatz and Z. M. Kedem. Consistency in Hierarchical Database Systems. *Journal of the Association for Computing Machinery*, 27(1):72–80, 1980.

[Yan82] Mihalis Yannakakis. A Theory of Safe Locking Policies in Database Systems. *Journal of the Association for Computing Machinery*, 29(3):718–740, July 1982.