Safe Locking Policies for Dynamic Databases *

Vinay K. Chaudhri
vinay@ai.sri.com
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 USA

Vassos Hadzilacos
vassos@db.toronto.edu
Department of Computer Science
University of Toronto
Toronto, ON M5S 1A4 CANADA

January 13, 1998

Abstract

Yannakakis showed that a locking policy is not safe if and only if it allows a canonical nonserializable schedule of transactions in which all transactions except one are executed serially [Yan82]. In the present paper, we study the generalization of this result to a dynamic database, that is, a database that may undergo insertions and deletions of entities. We illustrate the utility of this generalization by applying it to obtain correctness proofs of three locking policies that handle dynamic databases.

Keywords: Concurrency Control, Correctness Issues

---

*This is an revised version of a paper with the same title that was published in the Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems May 22-25, 1995. The work presented in the paper was done while the first author was at University of Toronto, where it 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.

1 Introduction

A locking policy is called safe if any concurrent execution of transactions locked according to that policy is guaranteed to be correct. Yannakakis showed that a locking policy is not safe if and only if there exists a canonical nonserializable schedule in which all transactions except one are executed serially [Yan82]. This characterization is useful in proving the correctness of a wide range of locking policies: it restricts the schedules that a policy must avoid to a small and highly structured set. The result on canonical nonserializable schedules was shown only for static databases, that is, databases that do not undergo insertions and deletions of entities. In the present paper, we generalize this result to dynamic databases — i.e., databases that undergo insertions and deletions of entities — and consider three applications of this generalization.

In a dynamic database, some interleavings of a set of transactions do not make sense. Such a situation arises when transactions perform operations that are not applicable in certain states of the database. For example, consider a graph-structured database in which some of the transactions perform traversals. The order in which a traversal transaction accesses entities may be formalized as follows: a transaction may begin by accessing any node; subsequently, a node A may be accessed only if there is an edge from a previously accessed node to A. Consider a database with nodes A, B and C and edges (A, B) and (B, C). In this database, if a traversal transaction T_1 accesses node A, then the next node accessed by it can only be B, because B is the only node that is connected to A. If we interleave the execution of T_1 with the execution of another transaction T_2 that inserts the edge (A, C), T_1 could access C after A. In other words, T_1's access of C after A would make sense only if its execution is interleaved with the execution of T_2. We formalize the sensible interleavings of transactions by defining *proper schedules*.

The discussion of the previous paragraph implies that in dynamic databases, canonical nonserializable schedules must be proper in addition to satisfying all the properties required of such schedules in static databases. As a result, the structure of canonical nonserializable schedules in dynamic databases is not as simple as in static databases. Despite their greater complexity, however, canonical schedules in the new setting have sufficiently regular structure that they can be taken advantage of to simplify correctness proofs for locking policies for dynamic databases. Specifically, in this paper, we use this technique to prove the correctness of three locking policies (1) the Dynamic Directed Acyclic Graph (DDAG) policy which is an extension of the tree policy [SK80, KS83] and was designed for use in knowledge based applications, (2) Altruistic Locking [SGMS94] that was designed to deal with long-lived transactions, and (3) Dynamic Tree Policy [CM86] that allows transactions to dynamically define the structure (partial order on the entities) of a database. It is interesting to note that these locking policies were developed in quite different contexts.

The rest of this paper is organized as follows. In Section 2, we introduce our model of a dynamic database and highlight the differences from previous models. In Section 3, we state and prove the generalized version of the theorem on canonical schedules for dynamic databases. In Sections 4–6, we use the canonical schedules theorem to prove the correctness of the three locking policies considered in this paper. We discuss related work and conclude the paper in Section 7.

2 Model of a Dynamic Database

Let U be the universe of all entities that may exist in the database over its life time. A database is defined by a selection G of entities from the set U, $G \subseteq U$, and an assignment of values to the entities in G. Each distinct selection of entities is called a *structural* state of the database. Corresponding to each structural state, there can be several different assignments of values to the entities. Each such assignment is called a *value* state.

A transaction operates on the database by performing one or more operations from a set of operations O. In our model, we take $O = \{\texttt{R,W,I,D}\}$, where `R,W,I` and `D` are abbreviations for `READ, WRITE, INSERT` and `DELETE` respectively. The `INSERT` and `DELETE` operations change the structural state of the database by inserting or deleting an entity. The `WRITE` operation changes a value state by assigning a different value to some entity in the database. A `READ` operation does not have any effect on the state of the database.

Formally, a *step* is a pair (`a` e), where `a` is an operation (from the set `R, W, I, D`) and e is an entity. We say that a step is a `READ` (or `WRITE, INSERT, DELETE`) step if it involves a `READ` (respectively, `WRITE, INSERT, DELETE`) operation.

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

A *transaction* is a finite sequence of steps over $O \times U$. A *transaction system* is a collection τ of transactions. A *schedule* S of a transaction system τ is an ordering of the steps of some transactions of τ that preserves the order of steps of each transaction. Some such orderings may not be possible. A schedule S is *proper* for a structural state G, if $S(G)$ is defined.

As an example, consider the following schedule that begins when the database is empty.

```
T1: (I a) (I b)                   (W c) (I d)
T2:             (R a) (D b) (I c)
```

The above schedule is proper because each step is defined in the state in which it is executed even though execution of either transaction by itself would not be proper. On the other hand, consider the following interleaving of the same transactions.

```
T1: (I a) (I b) (W c)                   (I d)
T2:                   (R a) (D b) (I c)
```

The above interleaving is not proper, because T_1 writes the entity `c` when the database consists of only `a` and `b`.

A transaction models a particular interaction between a "user" (application program or human) and the database. A schedule models a particular interaction between concurrent users and the database. The question naturally arises how the individual users can ensure that the schedules produced by their interactions with the database are proper. To do so, they must have some way of knowing whether an entity is present in the current state of the database (in which case they may issue a `READ`, `WRITE` or `DELETE` operation) or absent from it (in which case they may issue an `INSERT` operation). A user may be able to ascertain the existence of an entity in a variety of ways, depending on aspects of the database system that are not represented in our model. For instance, users may have knowledge about the database and its other potential users from which they can infer that a certain entity is present in the database or absent from it. Alternatively, a user may rely on information obtained by reading an entity (the "system directory") that always exists and identifies all other entities presently in the database. Finally, the entities themselves may contain pointers through which the user can find other entities (as in the DDAG locking policy discussed in Section 4). Our model of the interaction between users and the database system is agnostic as to which of these (or any other) means users employ to ensure the properness of the schedules they produce.

Locked transactions are transactions with four additional operations, `LOCK-SHARED`, `LOCK-EXCLUSIVE`, `UNLOCK-SHARED` and `UNLOCK-EXCLUSIVE` (abbreviated by `LS`, `LX`, `US` and `UX` respectively). We use `LOCK` (abbreviated by `L`) and `UNLOCK` (abbreviated by `U`) as a generic notation for either type (shared or exclusive) of lock and unlock operations. Thus, the set of allowed operations is now $O_L = \{\mathtt{R,W,I,D,LS,LX,US,UX}\}$. A locked transaction is a sequence over $O_L \times U$.

We say that a locked transaction T holds an exclusive (respectively, shared) lock on an entity A in its prefix T' if there exists a ($\mathtt{LX}$ A) (respectively, ($\mathtt{LS}$ A)) step in T', that is not followed by a ($\mathtt{UX}$ A) (respectively, ($\mathtt{US}$ A)) step in T'. A locked transaction T is well-formed if an `INSERT`, `DELETE` or `WRITE` operation on an entity A takes place only if A is locked in exclusive mode in the prefix of T up to that point, and a `READ` operation on an entity A takes place only if A is locked in either shared or exclusive mode in the prefix of T up to that point. (Thus, before inserting an entity a transaction must lock it even though it does not actually exist in the database.) For the rest of this paper, we deal only with well-formed transactions.

A schedule S is a *legal schedule* if there is no prefix S' of S and distinct transactions T_1 and T_2 such that T_1 holds an exclusive lock on an entity A in S' and T_2 holds a shared or exclusive lock on A in S'.

Two steps *conflict* if they operate on a common entity and the operations of the two steps are not both in $\{\mathtt{R}, \mathtt{LS}, \mathtt{US}\}$. A schedule S is *serializable* if there is a serial schedule S' of the same locked transactions such that all conflicting steps of transactions in S appear in the same order in S as in S'.[1] The *serializability graph* $D(S)$ of S is a graph with a node for each transaction and an edge (T_i, T_j) if a step of T_i precedes in S a conflicting step of T_j. It is well known that S is serializable if and only if $D(S)$ is acyclic [EGLT76]. A transaction system τ is *safe* if any legal and proper schedule of τ is serializable.

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

We assume that a transaction locks an entity at most once, because a locking policy that allows a transaction to lock an entity twice is trivially unsafe [Yan82]. A *partial schedule* of a set of locked transactions $\{T_1, \ldots, T_m\}$ is a prefix of any schedule of these transactions.

The main difference between the model described above and that used by Yannakakis [Yan82] is the distinction between the structural and value states of the database. In the case of a static database, this distinction is not necessary, because the database is always in the same structural state. In the dynamic settling, the notion of structural state is needed to

[1] The notion of serializability that we use here is known as *conflict* serializability. Other related notions have also been defined and a discussion of different notions of serializability can be found elsewhere [Pap86, BHG87].

formalize the sensible interleavings of transactions in terms of proper schedules.

3 Theorem on Canonical Schedules

In this section we state and prove the theorem on canonical nonserializable schedules. We also discuss a simpler version of this theorem that applies when only exlusive locks are allowed.

3.1 Canonical Schedules Theorem with Shared and Exclusive Locks

Theorem 1 *A locked transaction system τ is not safe if and only if there are transactions $T_1, \ldots, T_k$ ($k > 1$) in τ, some c, $1 \le c \le k$, and an entity A^* such that*

(1) T_c locks A^* after it has unlocked some entity.

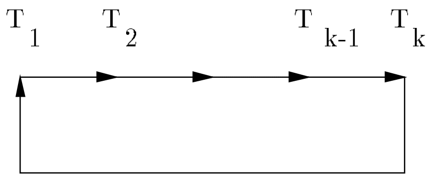
(2) Let T_c' be the prefix of T_c up to but excluding the ($\mathtt{L}\,A^*$) step. For each $i \neq c$, $1 \le i \le k$, there is a prefix T_i' of T_i, such that the partial schedule S' consisting of a serial execution of $T_1', \ldots, T_k'$ (in that order) satisfies the following conditions:

(a) Every T_i' that is a sink of $D(S')$ unlocks A^* having previously locked it in a mode that conflicts with the mode in which T_c locks A^*.

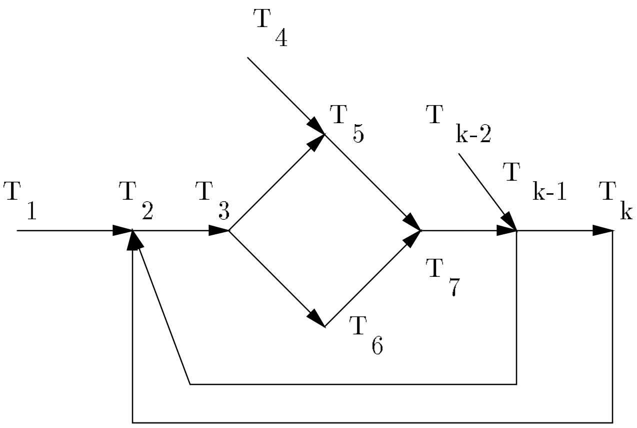
(b) S' can be extended to a complete legal and proper schedule.

This theorem characterizes unsafe (and hence safe) transaction systems. Condition 1 requires that at least one transaction T_c must violate the two-phase locking rule by locking an entity after it has unlocked some entity. If all transactions obey two-phase locking we can immediately conclude that the transaction system is safe.

Condition 2 requires the existence of a special kind of "dangerous" serial partial schedule S'. Although S' is serializable (in fact, serial), every completion of it is not. The next step of T_c to be executed after S' is ($\mathtt{L}$ A^*). When this step is executed, the serializability graph will have, in addition to the edges of $D(S')$, edges running from each sink of $D(S')$ back to T_c, resulting in at least one cycle. This is implied by Condition 2a. T_c' is not a sink of $D(S')$: if it



(a) Static system



(b) Dynamic System

Figure 1: Example serializability graphs for canonical schedules

was, it would have to lock A^* which is impossible since T_c locks A *after* the prefix T'_c (recall that a transaction may lock an entity only once). Condition 2b ensures that the potential danger inherent in $D(S')$ can, in fact, materialize. The fact that only those transactions can be sinks that have locked A^* in a mode that conflicts with T_c's subsequent lock of A^* ensures that the set of transactions involved in the "dangerous" serial schedule S' is, in some sense, minimal.

The benefit of the above theorem is that, in proving the correctness of a locking policy, one does not have to consider arbitrarily complex schedules but can just focus on canonical schedules. Due to the serial nature of this schedule, it is much easier to reason with it as compared to an arbitrary schedule.

The key difference between the above theorem and its analogue for static databases [Yan82] derives from the fact that the properness of schedules does not now come for "free": Special precautions must be taken to ensure that the canonical nonserializable schedule is, indeed, proper. This difference influences the theorem in two ways. First, the transaction T_c that locks A^* and creates the nonserializable schedule when S' is extended by (`L` A^*) step is not necessarily the first transaction in the sequence $T_1', \ldots, T_k'$. This is because the properness of the schedule involving transactions $T_c, \ldots, T_k$ may depend on the entities inserted/deleted by transactions $T_1, \ldots, T_{c-1}$. Second, $D(S')$ can have more than one source and more than one sink. Multiple sinks may arise when the transactions that are sinks of $D(S')$ lock A^* in shared mode. For static databases, $D(S')$ consists of a simple path and is closed by a back edge when T_c (which for static databases can be taken to be T_1) locks A^* (Figure 1(a)), whereas for dynamic databases the corresponding serialization graph is not necessarily a simple path (Figure 1(b)).

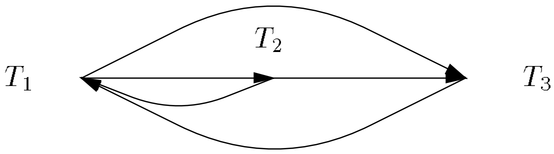
The need to restrict attention only to proper schedules makes the characterization of unsafe transaction systems more complex. To appreciate the necessity of the additional complexity, let us consider Yannakakis' characterization in the static setting and see why the straightforward generalization will not do. In the static case one defines an interaction graph of transactions in which each transaction is represented as a node, and there is an edge for every pair of conflicting steps in the transactions corresponding to those nodes. It then suffices to consider the schedules of only those transactions that form a chordless cycle in the interaction graph. This, as we shall see, is not sufficient in the dynamic setting.

Consider the nonserializable legal and proper schedule of three transactions shown in Figure 3.1. The interaction graph of these three transactions contains a pair of edges between any two transactions. Thus, the only chordless cycles are those involving two nodes. No schedule involving only two of the three transactions is proper (because then one of the two transactions accesses an entity that does not exist in the database). Thus, if we had restricted our attention to canonical schedules involving only transactions on chordless cycles, we would have concluded that there is no nonserializable canonical schedule (simply because there is no proper canonical schedule). We would have, therefore, incorrectly pronounced the transaction system safe, having missed the nonserializable schedule of Figure 3.1. Thus, in dynamic databases it is not sufficient to restrict our attention to canonical nonserializable schedules that correspond to chordless cycles of the interaction graph, leading to a more complex characterization in Theorem 1.

T_1: `(I a)`   `(D b) (D c)`

T_2:   `(W a) (I b)`

T_3:   `(D a) (R b) (I c)`



(a) Serializability Graph $D(S_p)$

3.2 Proof of the Canonical Schedules Theorem

We first establish some facts needed for the proof of Theorem 1

Lemma 1 *Let S be any schedule, and $\hat{S}$ be the schedule obtained from S by transposing two adjacent steps t and t' that belong to different transactions and do not conflict. If S is legal and proper, then $\hat{S}$ is also legal and proper, and $D(S) = D(\hat{S})$.*

Proof. Let T and T' be the transactions to which t and t' belong, respectively. Without loss of generality, assume t precedes t' in S, and let S' be the prefix of S up to (excluding) t.

First we show that if S is legal then so is $\hat{S}$. Suppose, for contradiction, that S is legal but $\hat{S}$ is not. Then in $S't't$ two distinct transactions hold conflicting locks on some entity while in $S'tt'$ they do not. This implies that t is an `UNLOCK` step and t' is a conflicting `LOCK` step. This contradicts the hypothesis that t and t' do not conflict.

Next we show that if S is proper and legal then $\hat{S}$ is also proper. Suppose, for contradiction, that S is proper and legal but $\hat{S}$ is not proper. Thus, although $S'tt'$ is proper, $S't't$ is not. This means that, for some entity A, either t is an (`I` A) step and t' is a (`R` A), (`W` A) or (`D` A) step; or t is a (`D` A) step and t' is an (`I` A) step. In either case, in $S'tt'$, two transactions are holding conflicting locks on A, contradicting the fact that S is legal.

Since any two conflicting steps are in the same order in S as in $\hat{S}$, $D(S) = D(\hat{S})$. □

Let S be a schedule, S' be a prefix of S, and T' be subsequence of S'. Define $\textbf{move}(S, S', T')$ to be the schedule obtained from S by moving the steps of T' to follow all other steps of S' while preserving the order of any two steps in T', as well as the order of any two steps not in T'. More precisely, $\textbf{move}(S, S', T')$ is the permutation $\hat{S}$ of S so that for any two steps t, t' in S:

- if t, t' are both in T' or both not in T', then t is before t' in $\hat{S}$ if and only if t is before t' in S;
- if t is in T' and t' is not, then t' is before t in $\hat{S}$ if and only if t' is in S'.

Lemma 2 *Let S be any schedule, S' be any prefix of S, and T' be any transaction that is a sink in $D(S')$. If S is legal and proper, then* $\mathbf{move}(S, S', T')$ *is also legal and proper, and* $D(\mathbf{move}(S, S', T')) = D(S)$.

PROOF. Assume that S is legal and proper. Let $T' = t_1, t_2, \ldots, t_s$. Define inductively two sequences of schedules $S_0, S_1, \ldots, S_s$ and $S'_0, S'_1, \ldots, S'_s$, where S'_i is a prefix of S_i, as follows:

- $S_0 = S$, $S'_0 = S'$.
- For any i, $0 < i \leq s$, $S_i = \mathbf{move}(S_{i-1}, S'_{i-1}, t_{s-i+1})$, and S'_i is the longest prefix of S_i that does not contain t_{s-i+1}.

Let $T'_i = t_1, t_2, \ldots, t_{s-i}$. We claim that for any i, $0 \leq i \leq s$: (a) T'_i is a sink in S'_i; (b) S_i is legal and proper; and (c) $D(S_i) = D(S)$. This is shown by induction on i. The basis, $i = 0$, is trivial. Part (a) of the induction step is immediate from the definitions of T'_i and S'_i. Part (b) follows by repeated use of Lemma 1, noting that (by part (a) of the induction hypothesis), t_{s-i+1} does not conflict with any step that follows it in S'_{i-1}. For part (c), note that every pair of conflicting steps are in the same order in S_i as in S_{i-1}. Thus, $D(S_i) = D(S_{i-1})$. Combining this with part (c) of the induction hypothesis, $D(S_i) = D(S)$, as wanted. □

We now turn to the proof of Theorem 1.

If. Let S be a complete, legal and proper schedule of τ that has S' as a prefix (such a schedule exists by Condition 2b). $D(S')$ is a subgraph of $D(S)$. As noted earlier, T'_c is not a sink of $D(S')$. Thus, there is a nontrivial path from T'_c to some sink of $D(S')$, say T'_i, and hence a nontrivial path from from T_c to T_i in $D(S)$. In S, T_c locks A^* after the prefix S'. By Condition 2a, $D(S)$ contains the edge (T_i, T_c). Therefore, $D(S)$ contains a cycle and S is not serializable. Thus τ is not safe.

Only If. Suppose τ is not safe, and let S be a legal and proper nonserializable schedule of τ. Given such a schedule, we define schedules S^- and S^+, transaction T_c, and an entity A^* as follows: S^- is the longest prefix of S such that $D(S^-)$ is acyclic, and S^+ is the shortest prefix for which $D(S^+)$ has a cycle; thus, S^+ is S^- extended with one lock step. T_c is the transaction that performs this step and A^* is the entity locked in that step. Intuitively, T_c is the transaction that causes the earliest cycle(s) in $D(S)$ and does so by locking A^*.

We claim that τ contains a legal, proper and nonserializable schedule S with the following property:

> Every T_i that is a sink of $D(S^-)$ unlocks A^* having previously locked it in a mode that conflicts with the mode in which T_c locks A^*. $(*)$

To see this, let S be any legal, proper and nonserializable schedule of τ. Define $\mathcal{M}(S)$ to be the set of transactions that neither unlock A^* in S^- nor precede in $D(S^-)$ a transaction that unlocks A^* in S^-. Choose S so that the cardinality of $\mathcal{M}(S)$ is as small as possible. If $\mathcal{M}(S)$ is nonempty, then it contains at least one transaction that is a sink in $D(S^-)$. Let T' be such a transaction. Since T' is a sink in $D(S^-)$ and does not unlock A^* in S^-, it follows that T' is also a sink in $D(S^+)$. Let $\hat{S} = \mathbf{move}(S, S^+, T')$. By Lemma 2, $\hat{S}$ is legal, proper and nonserializable. Furthermore, it is easy to see that $|\mathcal{M}(\hat{S})| < |\mathcal{M}(S)|$, contradicting the definition of S. Thus, $\mathcal{M}(S)$ is empty. In that case, S satisfies $(*)$.

Thus, we may assume that S is a legal, proper and nonserializable schedule that satisfies $(*)$. Let $T_1', T_2', \ldots, T_k'$ be a topological sort of the transactions in $D(S^-)$. We shall show that the transaction T_c (that causes the earliest cycle(s) in S), the entity A^*(whose locking by T_c causes the earliest cycle(s) in S), and the partial schedule S' consisting of the serial execution of $T_1', T_2', \ldots, T_k'$ (in that order) satisfy the conditions of Theorem 1.

As argued earlier, T_c' is not a sink of $D(S^-)$, and thus, it must unlock some entity (subsequently locked by some other transaction) in S^-. Since (`L` A^*) is the next step of T_c after the steps it has executed in S^-, it follows that T_c unlocks some entity before it locks A^*, which proves that Condition 1 of the theorem holds.

To prove that Condition 2 holds, we proceed as follows: We progressively transform S by moving first the steps of T'_k to the end of S^-, then the steps of T'_{k-1} just before those of T'_k, then the steps of T'_{k-2} just before those of T'_{k-1}, and so forth. The schedule resulting at the end of this process satisfies Condition 2. More precisely, we define inductively two sequences of schedules $S_0, S_1, \ldots, S_s$ and $S'_0, S'_1, \ldots, S'_s$, where S'_i is a prefix of S_i, as follows:

- $S_0 = S$, $S'_0 = S^-$.
- For any i, $0 < i \leq k$, $S_i = \mathbf{move}(S_{i-1}, S'_{i-1}, T'_{k-i+1})$, and S'_i is the longest prefix of S_i that does not contain T'_{k-i+1}.

We claim that for any i, $0 \leq i \leq k$: (a) T'_{k-i} is a sink in $D(S'_i)$; (b) S_i is legal and proper; and (c) $D(S_i^-) = D(S^-)$. This is shown by induction on i. The basis, $i = 0$, is trivial. For the induction step, part (a) is immediate from the definition of S_i and S'_i. Part (b) follows by Lemma 2, using the facts that T'_{k-i+1} is a sink in $D(S'_{i-1})$ (by part (a) of the induction hypothesis) and S_{i-1} is proper and legal (by part (b) of the induction hypothesis). For (c) note that the order of any two conflicting steps in S_i is the same as in S_{i-1}. (This is because, by Lemma 2, $D(S_i) = D(S_{i-1})$, and S_i has the same steps as S_{i-1}.) Thus, S_i^- is the prefix of S_i up to and excluding the ($\mathtt{L}$ A^*) step of T_c, and $D(S_i^-) = D(S_{i-1}^-)$. By part (c) of the induction hypothesis, $D(S_{i-1}^-) = D(S^-)$, and therefore, $D(S_i^-) = D(S^-)$, as desired.

Let $S' = S_k^-$. By construction, S' is the partial schedule consisting of the serial execution of $T'_1, T'_2, \ldots, T'_k$ (in that order). Since $D(S') = D(S^-)$, the two graphs have the same sinks. Recall that S satisfies property $(*)$. Thus, all sinks of $D(S')$ unlock A^*, having previously locked it in a mode that conflicts with T_c's lock of A^*. This proves that S' satisfies Condition 2a of Theorem 1. S' can be extended to a complete legal and proper schedule because it is a prefix of S_k (which has these properties). Thus, S' also satisfies Condition 2b of Theorem 1.

This completetes the proof of Theorem 1.

3.3 Canonical Schedules Theorem with only Exclusive Locks

When only exclusive locks are permitted, Condition 2a of Theorem 1 can be simplified to:

(2a) $D(S')$ has a unique sink which unlocks A^*.

Now, $D(S')$ contains a unique sink as compared to a set of sinks in $D(S')$ of Theorem 1. The only way that multiple sinks can arise in Theorem 1 is when the sinks lock A^* in *shared* mode. When only exclusive locks are allowed, all lock operations on A^* by different transactions conflict, resulting in a unique sink in $D(S')$.

4 Dynamic Directed Acyclic Graph Policy

Many advanced database systems, such as object-oriented databases or knowledge based systems can be viewed as directed graphs. For example, in a system storing information about complex objects, the part-subpart relationship is a directed graph in which each node represents an object and an edge represents a part-of relationship between two objects. To take advantage of the graph-like structure of such systems, we extended the DAG policy [SK80, Yan82] to the Dynamic Directed Acyclic Graph (DDAG) policy which can deal with general Dynamic Acyclic Graphs. In the present paper, we consider the case when only exclusive locks are used. We assume that an `ACCESS` operation represents a `READ` immediately followed by a `WRITE`. A more general version of the policy that works for both shared and exclusive locks and for general graphs may be found elsewhere [Cha95]. We first define some properties of directed graphs that are necessary for specifying our algorithm.

A *root* of a directed graph is a node with no predecessors. A directed graph is *rooted* if it has a unique root and there is a path from the root to every node in the graph. A dominator D of a set of nodes W is a node such that every path from the root to a node in W passes through D. Thus, in a rooted graph, the root dominates all the nodes in the graph including itself.

We assume that we are given a rooted directed acyclic graph (DAG) representation G of the database. The set of entities now consists of all the nodes and edges in the DAG representation of the database. The transactions consist of `ACCESS`, `INSERT` and `DELETE` operations on the entities in the database. We assume that the transactions ensure that these properties are always maintained. We also assume that once an object has been deleted from the database, it may not be inserted into it again. More discussion on this assumption can be found elsewhere [Cha95].

Locking Rules for a Transaction T

L1. Before a transaction T performs any `INSERT`, `DELETE` or `ACCESS` operation on a node A (an edge (A, B)), T locks A (both A and B).

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

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

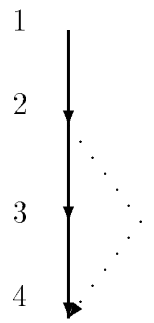
L4. A transaction may begin by locking any node.

L5. Other than the first node locked by T, a node in G can be locked by T only if all its predecessors in the *present* state of G have been locked by T in the past, and T is presently holding a lock on at least one of them.

Note that rule L5 referes to the state of the graph at the time the rule is applied — not the state of the graph when the transaction started execution.

As an example application of the DDAG policy, consider the database and the transactions shown in Figure 2. T_1 begins by locking node 2 (L4), then it locks nodes 3 and 4 (L5) and then unlocks node 3. T_2 begins by locking node 3 (L1). In the meantime, T_1 releases the lock on 4, and T_2 proceeds by locking node 4. If T_1 had added the edge $(2, 4)$ (L1), while it had locks on both nodes 2 and 4, then T_2 will be unable to lock node 4: to do so, it must lock node 2 which is a predecessor of node 4 in the current state of the graph (L5). T_2 must abort and start from node 2.

Let us now use Theorem 1 to prove the correctness of the DDAG policy. We divide the argument into two parts addressed in the next two subsections. We first consider properties implied by the locking rules of the DDAG policy. Then, we make the correctness argument based on Theorem 1.



T_1: `(LX 2) (LX 3) (LX 4) (U 3)     (U *)`

T_2: `                         (LX 3)            (LX 4) (U *)`

Figure 2: An Example Application of the DDAG policy

4.1 Properties of the DDAG policy

Lemma 3 *Let T be any transaction following the DDAG policy, B be the first entity locked by T, and G be the state of the graph when T begins execution. In the absence of any other concurrent transactions:*

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

(b) For each A in G that is locked by T, all nodes that are ancestors of A and descendants of B in G, are locked by T before A.

Proof of Lemma 3.

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

Base case, $i = 1$. T first locks B, which dominates itself.

Induction step. Suppose the lemma is true for the first m nodes locked by T, $m \geq 1$.

Let C be the $(m+1)$st node locked by T. If C is inserted by T then we are done. If C is not inserted by T, it must be in G, and T must have locked in the past all the predecessors $C_1, \ldots, C_l$ of C (by rule L5). Every path from the root to C must go through one of the predecessors $C_1, \ldots, C_l$. By induction hypothesis, every path from the root to any of $C_1, \ldots, C_l$ must go through B. Thus, any path from the root to C must go through B as well. Hence C is dominated by B.

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

Base case, $i = 1$. The lemma is trivially true for the first entity locked by T.

Induction step. Suppose the lemma holds for each of the first m nodes locked by $T, m \geq 1$.

Let C be the $(m+1)$st node locked by T. T must have locked all the predecessors of C (by rule L5), and the lemma is true for C by induction hypothesis. If C is being inserted it has no predecessors in G and the lemma is trivially satisfied. $\square$

4.2 Correctness of the DDAG Policy

Theorem 2 *The DDAG policy is safe.*

PROOF. Suppose the DDAG policy is not safe. Choose transactions $T_1, \ldots, T_k$, locked according to the policy, a distinguished transaction T_c among these, and an entity A^* that satisfy the two conditions of Theorem 1. Let $T_1^{'}, \ldots, T_k^{'}$ and $S^{'}$ be as in Condition 2 of Theorem 1. Without loss of generality, assume that $T_1^{'}, \ldots, T_k^{'}$ is a topological sort of $D(S^{'})$. Since we are dealing with only exclusive locks, $T_k^{'}$ is the unique sink of $D(S^{'})$.

For $1 \leq i \leq k$, let G_i be the structural state of the database when $T_i^{'}$ begins execution, and B_i be the first entity locked by $T_i^{'}$. Let G_{k+1} be the structural database state when $T_k^{'}$ finishes its execution.

Since A^* is not the first entity locked by T_c (see Condition 1 of Theorem 1), and T_c is about to lock A^* when the database is in structural state G_{k+1}, T_c must be holding a lock on at least one predecessor of A^* in G_{k+1}. This follows from the locking rules L5 and that T_c could not be inserting A^* using rule L2, because if that were the case it would mean that $T_k^{'}$ deleted A^*, and we assume that an entity that has been deleted cannot be re-inserted. Let F be the predecessor of A^* in G_{k+1} on which T_c is holding a lock at the time it locks A^*. T_c has held this lock since it executed its prefix $T_c^{'}$ and throughout the execution of $T_{c+1}^{'}, \ldots, T_k^{'}$ in $S^{'}$.

We first establish some facts used later in the proof.

Lemma 4 *A^* is a successor of F in all of $G_c, \ldots, G_{k+1}$.*

PROOF OF LEMMA 4.

By definition, F is a predecessor of A^* in G_{k+1}. T_c has held a lock on F since it executed its prefix $T_c^{'}$. Since T_c locks A^* in the step immediately following $T_c^{'}$, it could not have locked A^* in $T_c^{'}$ because L3 prevents a transaction from locking a node more than once. Therefore, $T_c^{'}$ did not insert or delete the edge (F, A^*) (because doing so would require locking of A^*, by rule L1). Since T_c has held a lock on F since it executed its prefix $T_c^{'}$, no transaction from $T_{c+1}^{'}, \ldots, T_k^{'}$ could have deleted or inserted the edge (F, A^*) (because doing so would require locking of F, by locking rule L1). Hence A^* must be a successor of F in $G_c, \ldots, G_{k+1}$.

END OF LEMMA 4. $\square$

Lemma 5 *If a node A is a proper descendant of A^* in some $G_i, c < i \leq k+1$, then for all $j, c \leq j < i$, if $A \in G_j$, A is a proper descendant of A^* in G_j.*

Proof of Lemma 5.

Suppose, for contradiction, that the lemma is false, and consider the smallest $j, c \leq j \leq k$, such that $A \in G_j$, A is not a descendant of A^* in G_j, but A is a descendant of A^* in G_{j+1}. To accomplish this, T_j' must insert an edge from a descendant B of A^* in G_j to an ancestor C of A in G_j. Thus, T_j' must lock B and C (by locking rule L1). By Lemma 3(a), B_j (the first node locked by T_j) must dominate, and thus be an ancestor of, B and C in G_j. Since C is not a descendant of A^* in G_j, B_j could not be a descendant of A^* in G_j. Since B_j dominates a descendant B of A^* in G_j, and B_j is not a descendant of A^* in G_j, B_j must be a proper ancestor of A^* in G_j (otherwise, there would be a path in G_j from the root to B that passes through A^*, but does not pass through B_j). Thus, A^* is a (proper) descendant of B_j and an ancestor of B in G_j. Since T_j' locks both B_j and B, by Lemma 3(b), it also locks A^*. We now show that it is impossible for T_j' to lock A^* yielding the desired contradiction.

If $j = c$, T_j' cannot lock A^*, because T_c locks A^* in the step immediately after T_c', and it cannot lock an item more than once (by rule L3). If $c < j \leq k$, T_j' cannot lock A^* for the following reason. Since $A^* \in G_j$ (by Lemma 4), T_j' could not be inserting A^*. A^*, being a proper descendant of B_j in G_j, is not the first entity locked by T_j'. Thus, to lock A^*, T_j' must have locked all predecessors of A^* in G_j (by rule L5); but F is a predecessor of A^* in G_j (by Lemma 4) and T_c' holds a lock on F. Thus, for all $c \leq j \leq k$, T_j' cannot lock A^*.

End of Lemma 5. □

Let $T_j, j > c$, be a transaction such that $(T_c', T_j') \in D(S')$ (T_j' exists since, as we argued before, T_c' is not a sink of $D(S')$.) Let A be an entity that is locked by both T_c' and T_j'. Thus, A must be a descendant of B_j in G_j (by Lemma 3(a)). As we shall show later, B_j is a descendant of A^* in G_j. Therefore, A is a descendant of A^* in G_j. By Lemma 5, A is a descendant of A^* in G_c as well. Since $T_c^{'}$ locks a predecessor F of A^* in G_c (by Lemma 4), B_c must be an ancestor of A^* in G_c (by Lemma 3(a)). Since $T_c^{'}$ locks A, it must also lock A^* which is an ancestor of A in G_c and a descendant of B_c in G_c (by Lemma 3(b)). But then T_c locks A^* twice: once in the prefix $T_c^{'}$ and once immediately after. This contradicts L3. Hence the assumed nonserializable schedule cannot exist and the DDAG policy is safe.

It remains to show that B_j is a descendant of A^* in G_j. Indeed, we prove by backward induction that B_i is a descendant of A^* in G_i for all $i, c < i \leq k$.

Base case: $i = k$.

Recall that F is a predecessor of A^* on which T_c holds a lock when it locks A^*. By Lemma 4, F is a predecessor of A^* in G_k. Since $S^{'}$ is a legal schedule (by virtue of being the prefix of a complete legal and proper schedule S — see Condition 2b of Theorem 1), T_k could not have locked F. By Condition 2a of Theorem 1, $T_k^{'}$ locks A^*. Thus, A^* must be the first entity locked by T_k (by rules L4, L5), i.e. $B_k = A^*$.

Induction step: Let $c+1 < m \leq k$, and suppose that $B_{m+1}, \ldots, B_k$ are descendants of A^* in $G_{m+1}, \ldots, G_k$ respectively. We will show that B_m is a descendant of A^* in G_m.

Suppose, for contradiction, that B_m is not a descendant of A^* in G_m. Let $T_i^{'}, i > m$, be a transaction such that $(T_m^{'}, T_i^{'}) \in D(S^{'})$ (such a transaction must exist because $T_k^{'}$ is the only sink of $D(S^{'})$ and $m \neq k$).

First we show that $T_m^{'}$ does not lock any descendant of A^* in G_m. For, suppose the contrary, and let A be a descendant of A^* in G_m that $T_m^{'}$ locks. B_m is a dominator of A in G_m (by Lemma 3(a)). Furthermore, B_m is an ancestor of A^* in G_m, because if it was incomparable to A^*, there would be a path from the root to A in G_m (via A^*) which does not pass through B_m. Since A^* is an ancestor of A and descendant of B_m, to lock A, $T_m^{'}$ must lock A^* (by Lemma 3(b)). Since A^* is not the first entity locked by $T_m^{'}$ (otherwise $B_m = A^*$ and B_m would be a descendant of A^*), $T_m^{'}$ must lock all predecessors of A^* in G_m. As F is a predecessor of A^* in G_m (by Lemma 4) and F is already locked by T_c, $T_m^{'}$ cannot lock F, and therefore, it cannot lock A^*. Thus, $T_m^{'}$ cannot lock any descendant of A^* in G_m.

Next we show that $T_i^{'}$ locked only descendants of A^* in G_m or the nodes inserted after $T_m^{'}$

T_1: `(LX 1,2) (UX 1)      (LX 3) (UX 2)            (UX *)`

T_2: `             (LX 1)              (LX 2) (LX 5) (UX *)`

Figure 3: An Example Application of Altruistic Locking

(i.e., by $T'_{m+1}, \ldots, T'_i$). Since T'_i began execution by locking B_i, T'_i locked only descendants of B_i in G_i and nodes it inserted. By induction hypothesis, B_i is a descendant of A^* in G_i, and therefore, T'_i locked only descendants of A^* in G_i and nodes it inserted. The nodes that are descendants of A^* in G_i were descendants of A^* in G_m, if they existed in G_m (by Lemma 5). Therefore, T'_i locked only descendants of A^* in G_m and nodes inserted after T'_m.

Thus, T'_m did not lock any descendants of A^* in G_m, and T'_i locked only descendants of A^* in G_m and nodes inserted after T'_m. The latter obviously cannot be locked by T'_m. Therefore, T'_m and T'_i do not lock anything in common. This contradicts the fact that $(T'_m, T'_i) \in D(S')$ (by the choice of T_i). Therefore, B_m is a descendant of A^* in G_m, completing the induction step. □

5 Altruistic Locking

Altruistic locking was developed to deal with long-lived transactions [SGMS94]. We first define the rules of the altruistic locking policy for a database that undergoes insertions and deletions of entities [SGMS94]. We consider the basic version of altruistic locking in which all locks are exclusive. As in the previous section we assume that an `ACCESS` operation represents a `READ` immediately followed by a `WRITE`.

The instant when a transaction acquires its last lock is known as its *locked point*. A transaction T_i is said to be in the *wake* of another transaction T_j if T_i has locked an item which has been unlocked by T_j, and T_j has not reached its own locked point. The rules of altruistic locking are as follows:

AL1. A transaction must acquire a lock on an item before performing an `INSERT, DELETE` or `ACCESS` operation on that item.

AL2. If a transaction T_i is in the wake of another active transaction T_j, then all items locked by T_i so far must have been unlocked by T_j in the past.

AL3. A transaction may lock an item only once.

As an example of altruistic locking, consider the schedule shown in Figure 3. Once T_1 releases the lock on entity 1, T_2 is able to lock entity 1, thus entering the wake of T_1. From now on, T_2 can lock only those entities that have been unlocked by T_1. Once T_1 reaches its locked point, when it locks entity 3, T_2 is no longer in the wake of T_1 and can lock any entity it needs.

The advantage of altruistic locking relative to the DDAG policy is that it does not impose any pre-specified order on the entities in the database. The disadvantage is that rule AL2 can be too restrictive, because to take advantage of the pre-release of locks a transaction has to be completely in the wake of another transaction. This may be relaxed, but only at the cost of several additional assumptions [SGMS94].

We now prove the correctness of altruistic locking using Theorem 1.

Theorem 3 *The altruistic locking policy is safe.*

Proof. Suppose, for contradiction, that the altruistic locking policy is not safe. Choose transactions $T_1, \ldots, T_k$, locked according to the policy, a distinguished transaction T_c among these, and an entity A^* that satisfy the two conditions of Theorem 1. Let $T_1', \ldots, T_k'$ and S' be as in Condition 2 of Theorem 1. Without loss of generality, assume that $T_1', \ldots, T_k'$ is a topological sort of $D(S')$. Since we are dealing with only exclusive locks, T_k' is the unique sink of $D(S')$ (Section 3.3).

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

<u>Base case</u>. The length of the shortest path from T_c' to T_i' in $D(S')$ is 1. Since T_c' and T_i' are adjacent in $D(S')$, there is an entity, say A, that is unlocked by T_c' and is locked by T_i'. In the prefix T_c', T_c has not reached its lock point, because it locks A^* after it has executed the prefix T_c' (Condition 2 of Theorem 1). Since T_i' locks A, which has been unlocked by T_c', and T_c has not reached its locked point in S', T_i' is in the wake of T_c in S'.

Induction step. Suppose that the claim is true for all transactions T_i' such that the length of the shortest path from T_c' to T_i' in $D(S')$ is less than m, $m \geq 2$.

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

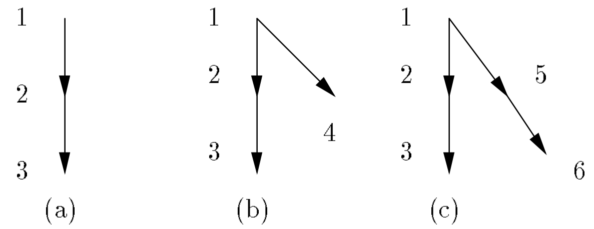
Hence by induction, all transactions $T_i', c < i \leq k\ T_i'$, are in the wake of T_c.

In particular, T_k' is in the wake of T_c. By Condition 2a of Theorem 1, T_k (the unique sink of $D(S')$) locks A^*. Hence, A^* must have been locked and then unlocked by T_c' (locking rule AL2). T_c also locks A^* in the step immediately following T_c'. This contradicts locking rule AL3. Hence the assumed canonical nonserializable schedule does not exist and the altruistic locking policy is safe. □

6 Dynamic Tree Policy

The Dynamic Tree (DTR) Policy was proposed to allow a changing set of partial orders to be defined over the objects in a database [CM86]. Thus, instead of assuming that we are given a directed graph corresponding to the database, the DTR policy defines a forest for itself depending, in part, on the entities that a transaction wishes to access. We call the forest defined by the DTR policy a *database forest*. We first give some definitions and then define the rules of the DTR policy for the case when all locks are exclusive.

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



T_1: `(LX 1,2) (U 1)` `(LX 3) (U 2)` `(U *)`

T_2: `(LX 1,4) (U 4)` `(LX 2) (U 1,2)`

T_3: `(LX 1,5,6) (U 5,6) (LX 2) (U *)`

Figure 4: An Example Application of Dynamic Tree Policy

Let $A(T)$ be the set of entities for which there is an explicit `ACCESS`, `INSERT` or `DELETE` step in T. T may lock some additional entities to satisfy the rules of the locking policy. Let $G(A)$ be the database forest obtained by deleting entity A from the forest G.

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

DT0. Initially the database forest G is empty.

DT1. To join two database trees $g_1, g_2 \in G$, draw an edge from the root of g_1 to the root of g_2. To add a set of entities to a tree g_1, first connect them to form a tree g_2 and then join g_1 and g_2.

DT2. When a transaction T starts, join all the trees that contain some entity $A \in A(T)$ to form a single tree g. Add to g all the entities in $A(T)$ that are not already in it. Tree-lock T with respect to g.

DT3. A node A may be deleted from the database forest G if it is not currently locked by any active transaction and for each active transaction T, T is tree-locked with respect to some $g \in G(A)$.

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

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

We observe that Lemma 3 applies to the DTR policy as well. Thus, if B is the first node locked by a transaction T in a database forest G, then in the absence of any other concurrent transactions, all nodes locked by T are dominated by B in G. (Since G is a forest, the nodes dominated by B are simply its descendents.) Similarly, in the absence of any other concurrent transaction, for each A in locked by T, all nodes that are ancestors of A and descendants of B in G, have been locked by T when it locks A. We now prove the correctness of this locking policy using Theorem 1. The structure of the argument is similar to that in the proof of Theorem 2, but some of the reasoning differs because of the difference between the nature of graphs in the DDAG and DTR policies.

Theorem 4 *The Dynamic Tree policy is safe.*

PROOF. Suppose the dynamic tree locking policy is not safe. Choose transactions $T_1, \ldots, T_k$, locked according to the policy, a distinguished transaction T_c among these, and an entity A^* that satisfy the two conditions of Theorem 1. Let $T_1', \ldots, T_k'$ and S' be as in Condition 2 of Theorem 1. Without loss of generality, assume that $T_1', \ldots, T_k'$ is a topological sort of $D(S')$. Since we are dealing with only exclusive locks, T_k' is the unique sink of $D(S')$ (Section 3.3).

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

Since A^* is not the first node locked by T_c (Condition 1 of Theorem 1), $T_c^{'}$ must be holding a lock on a predecessor F of A^* in G_{k+1}. This is because T_c is tree-locked with respect to G_c (by rule DT2) and is also tree-locked with respect to G_{k+1} (due to rule DT3).

We first establish some facts used later in the proof.

Lemma 6 *A^* is a successor of F in the database forests $G_{c+1}, \ldots, G_{k+1}$.*

Proof of Lemma 6.

First, we show that A^* and F belong to database forests $G_{c+1}, \ldots, G_{k+1}$. Since $T_c^{'}$ holds a lock on F, F cannot be deleted from the database forests $G_{c+1}, \ldots, G_{k+1}$ (rule DT3). Similarly, A^* cannot be deleted from the database forests $G_i, c < i \leq k+1$, because then T_c will not be tree-locked with respect to $G_i(A^*)$ (rule DT2, DT3). Hence, A^* and F belong to database forests $G_{c+1}, \ldots, G_{k+1}$.

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

Base case. $i = k+1$. Follows from the definition of F.

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

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

End of Lemma 6. □

Lemma 7 *If a node A is a proper descendant of A^* in some $G_i, c < i \leq k+1$, then for all $j, c \leq j < i$, if $A \in G_j$, A is a proper descendant of A^* in G_j.*

Proof of Lemma 7.

For contradiction suppose that the lemma is not true. Consider the smallest $j, c \leq j \leq k$, such that $A \in G_j$, A is not a descendant of A^* in G_j, but A is a descendant of A^* in G_{j+1}. The only way to change the position of A in G_j is to first delete A from G_j and then re-insert it as a child of the root (rule DT1). Even so, A cannot be a descendant of A^* in G_{j+1}, because A^*, being a descendant of F in G_{j+1} (by Lemma 6), is not a root in G_{j+1}. Hence the the lemma is true.

End of Lemma 7. □

We now proceed with the proof of Theorem 4. Let $T_j^{'}, c < j \leq k$, be a transaction that locks an entity A also locked by $T_c^{'}$ (such a transaction must exist because of Condition 2a of Theorem 1). Later we shall shown that B_j is a descendant of A^* in G_j. Therefore, A is also a descendant of A^* in G_j and also in G_c (by Lemma 7). Since $T_c^{'}$ locks a parent of A^* and also locks A which is a descendant of A^* it must also lock A^* (by Lemma 3(b)). But T_c also locks A^* in the step immediately following $T_c^{'}$. This is a contradiction, because a tree-locked transaction may lock a node only once. Hence the DTR policy must be correct.

It remains to show that B_j is a descendant of A^* in G_j. Indeed, we prove by backward induction that B_i is a descendant of A^* in G_i for all $i, c < i \leq k$.

Base case. $i = k$. T_k locks A^* and cannot lock its predecessor F in G_k (because F is locked by T_c). Since T_k is tree-locked with respect to G_k, A^* must be the first entity locked by T_k. Thus, $B_k = A^*$, and so B_k is a descendant of A^* in G_k.

Induction step. Suppose B_i is a descendant of A^* for all $i, c < m < i \leq k$. We show that B_m is a descendent of A^* in G_m. Suppose, for contradiction, that B_m is not a descendant of A^* in G_m.

We first prove that $T_m^{'}$ did not lock any descendants of A^* in G_m. For contradiction, suppose T_m' locked a descendant A of A^* in G_m. B_m must be an ancestor of A^* (it cannot be incomparable to A^* for, otherwise, there would be two distinct paths from the root to A — one via B_m and one via A^*, which is impossible in a tree.) Thus, T_m' locks both an ancestor and a descendant of A^*, and so it must lock A^*. A^* cannot be the first lock of T_m' (otherwise, $B_m{=}A^*$, and B_m is a descendant of A^*). So, T_m' must also lock F because F is the parent of T^* in G_m (by Lemma 6). But T_m' cannot lock F because F is already locked by T_c. Hence T_m' cannot lock any descendant of A^* in G_m.

Let T_j' be such that $(T_m', T_j') \in D(S')$ (such a transaction exists because $D(S')$ has a unique sink T_k' and $m \neq k$). By induction hypothesis, B_j is a descendant of A^* in G_j. Therefore T_j' locked only descendants of B_j, and hence only descendants of A^* in G_j (by Lemma 3(a)). The descendants of A^* in G_j were also descendants of A^* in G_m if they existed in G_m (by Lemma 7). Thus, T_j' locked only descendants of A^* in G_m.

Thus, T_m' did not lock any descendant of A^* in G_m and T_j' locked only descendants of A^* in G_m. Therefore, T_m' and T_j' do not lock any entity in common. This contradicts the fact that, by choice of T_j', $(T_m', T_j') \in D(S')$. Hence B_m is a descendant of A^* in G_m.

END OF THEOREM 4. $\square$

7 Discussion, Related Work and Conclusions

Some comments regarding the utility of Theorem 1 are now in order. A natural question is whether the correctness of the locking policies described in the previous three sections could be shown without Theorem 1. We invested significant effort to prove the correctness of the DDAG policy without the generalized version of the canonical schedules theorem (Theorem 1) but did not succeed. Even though we have not shown that the correctness cannot be proven otherwise, our experience suggests that the canonical schedules technique can make the correctness argument easier.

Correctness proofs for altruistic locking and dynamic tree locking had already been obtained by other researchers, without the use of canonical schedules. The only reason we considered them here was to see if the canonical schedules theorem has any relevance in a context other than the traversals on dynamic graphs. We found that Theorem 1 not only has general applicability but it leads to simpler proofs.

The results presented in this paper are based on the first author's doctoral dissertation [Cha95] and contribute towards a larger goal: to design concurrency control techniques for highly structured databases (or knowledge bases). The results on the performance evaluation of the DDAG policy have been published elsewhere [CHMS94]. Details of implementation considerations can be found in the dissertation itself [Cha95].

In summary, in this paper, we generalized Yannakakis' canonical nonserializable schedules theorem to dynamic databases. We showed that the shape of the serializability graph of a canonical nonserializable schedule in dynamic databases is more complex than in static databases. We used the generalized canonical schedules to prove the correctness of three locking policies: DDAG policy, dynamic tree policy and altruistic locking policy. This work can be extended in several directions. The same technique could be used to analyze other locking policies [AA90, dFRH94]. Furthermore, the generalization of these results to a dynamic hypergraph policy [Yan82] and incorporating recovery considerations in it [SWY93] remain problems for future research.

Acknowledgements

We thank the anonymous referees for their suggestions.

References

[AA90] D. Agrawal and A. El Abbadi. Locks with Constrained Sharing. In *9TH ACM Symposium on Principles of Database Systems*, pages 85–93, April 1990.

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

[Cha95] Vinay K Chaudhri. *Transaction Synchronization in Knowledge Bases: Concepts, Realization and Quantitative Evaluation.* PhD thesis, University of Toronto, Toronto, January 1995.

[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.

[CM86] Albert Croker and David Maier. A Dynamic Tree-Locking Protocol. In *Proceedings of the International Conference in Data Engineering*, pages 49–56, Los Angeles, CA, 1986.

[dFRH94] Fernando de Ferreira Rezende and Theo Härder. A Lock Method for KBMSs Using Abstraction Relationships' Semantics. In *Proceedings of the Third International Conference on Knowledge Management*, pages 112–121, 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.

[KS83] Z. M. Kedem and A. Silberschatz. Locking Protocols: From Exclusive to Shared Locks. *Journal of the Association for Computing Machinery*, 30(4):787–804, 1983.

[Pap86] Christos Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, Rockville, MD, 1986.

[SGMS94] Kenneth Salem, Hector Garcia-Molina, and Jeannie Shands. Altruistic Locking. *ACM Transactions on Database Systems*, 19(1):117–164, 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.

[SWY93] H.-J. Schek, G. Weikum, and H. Ye. Towards a Unified Theory of Concurrency Control and Recovery. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 300–311, June 1993.

[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.