Safe Locking Policies for Dynamic Databases

Vinay K Chaudhri and Vassos Hadzilacos
Department of Computer Science
University of Toronto
Toronto, Ontario M5S 1A4

October 27, 1994

Abstract

It was shown by Yannakakis that a locking policy is not safe if and only if there exists a canonical non-serializable schedule of transactions running according to the rules of the policy in which all the 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

1 Introduction

A locking policy is called safe if any concurrent execution of a set of transactions while 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 non-serializable schedule in which all transactions except one are executed serially [Yan82]. The serializability graph of such a schedule consists of a simple path closed by a back edge. Such a characterization is useful because it provides a technique that can be used to prove the correctness of a wide range of locking policies. This result, however, was shown only for a static database, that is, a database that does not receive insertions and deletions of entities. In this paper, we generalize this result for the case of a dynamic database, that is, a database that receives insertions and deletions of entities. We consider three applications of this generalization.

In a dynamic database, all possible interleavings of a set of transactions may not make sense. This happens when transactions perform operations that depend on existence or non-existence of entities in the database. For example, if an entity does not exist in the database, its deletion by a transaction does not make sense. Similarly, in a graph-structured database in which transactions access the entities by following the edges in the graph, access of an entity B after another entity A is possible only if B can be reached from A by following certain edges that exist in the database. Later on, we define *proper schedules*, which formalize the sensible interleavings of transactions.

This implies that canonical non-serializable schedules for a dynamic database must be proper in addition to satisfying all the properties required of such a schedule in a static database. As a consequence of this requirement, the serializability graph of a canonical non-serializable schedule for a dynamic database possesses more general structure than its counterpart in a static database: its serializability graph is not just a simple path closed by a back edge, but instead, it can be a general graph.

This characterization leads to a convenient proof technique that can be used to obtain correctness proofs for locking policies for dynamic databases. Specifically, we consider three locking policies (1) Dynamic Directed Graph policy with shared and exclusive locks (DDG-SX) [CHM92, Cha94], which is an extension of the tree policy [SK80, KS83] and was designed by us 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 a changing set of partial orders over a database. It is interesting to note that these locking policies were developed in quite different contexts.

The organization of this paper is as follows. In Section 2, we introduce our model of a dynamic database and highlight the differences from the previous models. In Section 3, we state and prove the generalized version of the theorem on canonical schedules for dynamic databases. In Sections 4–5, we use the canonical schedules theorem to prove the correctness of three locking policies considered in this paper. The proof for the DDG-SX policy is presented in more detail than the proofs for the other two policies. More details are given in Appendix B. Finally, we conclude the paper in Section 7 with a summary of results and conclusions.

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 $O = \{\mathtt{R,W,I,D}\}$, where `R,W,I` and `D` are the abbreviations for `READ, WRITE, INSERT` and `DELETE` respectively. The `INSERT` and `DELETE` operations change a 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.

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. A *transaction* is a finite sequence of steps over $O \times U$.

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

A *transaction system* is a collection τ of transactions. A *schedule* S of a transaction system τ is an ordering of the steps of some transactions of τ that preserves the order of operations of each transaction. Some such orderings may not be possible. For example, with G_1, G_2, G_3 and G_4 as structural database states, consider the interleaving:

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

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

When each write is preceded by a read, correctness (serializability) of a schedule S is equivalent to acyclicity of the serializability graph $D(S)$ [Pap79]. If $D(S)$ contains no cycles, then the schedule S is serializable. This condition is necessary if every `WRITE` is preceded by a `READ` and that is the assumption we adopt for the rest of this paper.

Locked transactions are transactions with three additional operations, `LOCKS`, `LOCKX` and `UNLOCK` (abbreviated by `LS`, `LX` and `U` respectively) in their repertoire. `LS` and `LX` denote acquisition of lock in shared and exclusive mode respectively. We use `L` as a generic notation for both `LS` and `LX`. Two locks conflict if at least one of them is exclusive. Thus the set of allowed operations is now, $O_L =${`R,W,I,D,LS,LX,U`}. A locked transaction is a sequence over $O_L \times U$.

A locked transaction is *well-formed* if an operation can take place only if the corresponding entity is locked. A *legal schedule* of a set of locked transactions is a schedule in which a transaction cannot lock an entity that is already locked by some other transaction in a conflicting mode. A transaction system τ is *safe* if any legal and proper schedule of τ is correct.

A *locking policy* P is a relation such that $P(T,\overline{T})$ only if transaction T is a subsequence of a well-formed locked transaction $\overline{T}$. Intuitively, a locking policy tells us how to put locks in a given transaction. If $P(T,\overline{T})$ holds then $\overline{T}$ is one of the ways of locking T according to the locking policy. 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 $\overline{\tau} = \{\overline{T}_1, \ldots, \overline{T}_m\}$, where $P(T_i, \overline{T_i})$ for all $1 \leq i \leq m$, the locked transaction system $\overline{\tau}$ is safe.

The *interaction graph* $G(\tau)$ of a transaction system τ is an undirected graph with one node corresponding to each transaction T_i of τ and an edge between any two nodes whose corresponding transactions have conflicting steps. We assume that two steps conflict if both of them are not `READ` and operate on the same entity.

A *partial schedule* of a set of locked transactions $\{T_1, \ldots, T_m\}$ is a prefix of any schedule of these transactions. We say that an entity A is *mentioned* or *referenced* by a transaction T if T has a step involving A. We denote by $R(T)$ the set of entities referenced by a transaction T.

Let us summarize here the differences between our model and that of Yannakakis [Yan82]. First, we distinguish between the structural and value states of the database. In the model of a static database, such a distinction was not necessary, because the database is always in the same structural state. Second, we explicitly consider `INSERT` and `DELETE` operations in our model which leads to the definition of proper schedules. In case of a static database, every schedule is proper. Finally, we have defined locking policy as a relation in contrast to the static database where it was defined as a function. This is because for a given transaction, there can be more than one distinct mappings to the locked transactions depending on the structural state in which it is executed. The DDG-SX policy considered in this paper is an example of one such policy.

3 Theorem on Canonical Schedules

In this section, we state the theorem on canonical schedules. The proof is given in Appendix B.

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

(1) There is a subsequence $(\overline{T}_1, \overline{A}_1), \ldots, (\overline{T}_{\overline{k}}, \overline{A}_{\overline{k}})$ of $(T_1, A_1), \ldots, (T_k, A_k)$ such that $\overline{T}_1, \ldots, \overline{T}_{\overline{k}}$ form a cycle in the interaction graph $G(\tau)$.

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

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

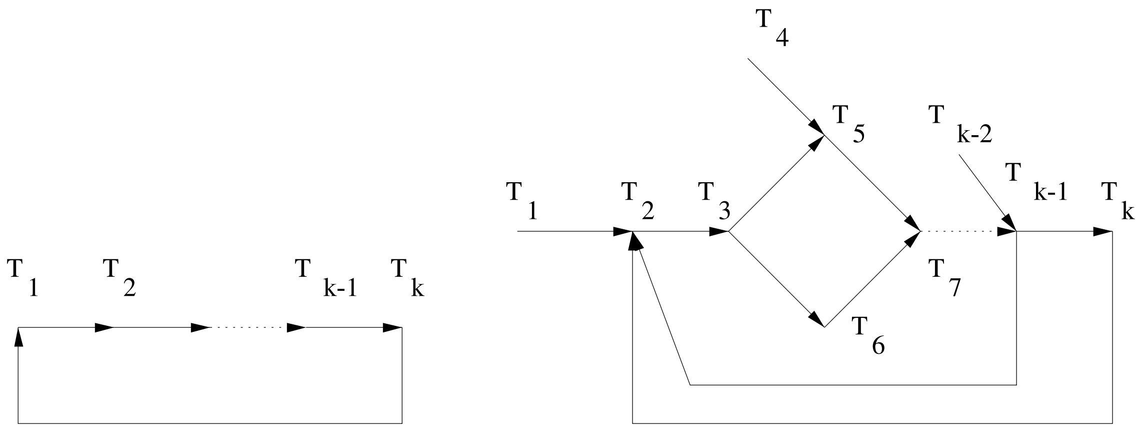
(a) $S^{'}$ is legal.

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

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

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

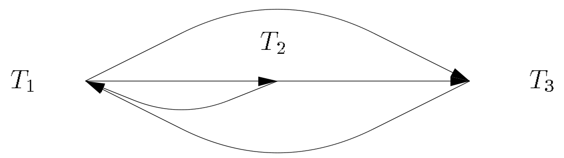
The key difference between the above theorem and its analogue for static databases [Yan82] derives from the fact that properness of schedules does not now come for "free": Special precautions must be taken to ensure that the canonical non-serializable schedule is, indeed, proper. This difference influences the theorem in three ways. First, all the transactions that are involved in the canonical schedule do not necessarily participate in a cycle in the interaction graph. Therefore, in condition 1 of the theorem, we need to consider a sub-sequence of the transactions $T_1, \ldots, T_k$. Second, the transaction T_c that locks A_k and creates the non-serializable schedule (condition (2) above) is not necessarily the first transaction in the sequence $T_1^{'}, \ldots, T_k^{'}$. This is because the properness of the transactions $T_c, \ldots, T_k$ may depend on the entities inserted/deleted by one or more of transactions $T_1, \ldots, T_{c-1}$. Finally, as specified in condition 3c, $D(S^{'})$ can have more than one source and more than one sink. For static databases, $D(S^{'})$ consists of a simple path which is closed by a back edge when T_c locks A_k (Figure 1(a)), whereas, for the dynamic databases the corresponding serialization graph is not necessarily a simple path (Figure 1(b)). The topological ordering of transactions (property 3(c)) becomes useful in the correctness proofs.



(a) Static system (b) Dynamic System

Figure 1: Example serializability graphs for canonical schedules

```
T1: (I a)                  (W b) (D c)
T2:    (W a) (I b)
T3:               (W a) (R b) (I c)
```



(a) Schedule S_p (b) Serializability Graph $D(S_p)$

Figure 2: A proper schedule S_p

As an example, consider the schedule S_p and its serializability graph shown in Figure 2. Suppose at the beginning of this schedule the database is empty. For this schedule to be proper, we need all three of the transactions. Even though T_1 and T_2 form a cycle in the interaction graph and the serializability graph, any complete schedule produced by their interleaving is not proper, because for T_1 to be able to complete, it must be able to delete the entity `c`, which will not exist in the database unless T_3 inserts it. We can see that $D(S_p)$ is not just a simple path closed by a back edge but has two parallel paths (from T_1 to T_3) and two back edges ((T_2, T_1) and (T_3, T_1)) that create a cycle.

4 Dynamic Directed 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 partof 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 Graph (DDG) policy which can deal with general dynamic graphs [CHM92]. DDG-SX is the variant of this policy that supports both shared and exclusive locks. We first define some properties of directed graphs that are necessary for specifying our algorithm.

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

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

We assume that we are given a directed graph representation G of the database. We assume that G is always connected and rooted. If G does not satisfy this property it can be converted into one that does by a simple construction [Cha94]. 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 [Cha94]. Let us now state the rules of the policy.

Locking Rules for a Transaction T

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

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

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

L4. The first lock obtained by a transaction T can be on any node. If the first node locked by T belongs to a non-trivial SCC, all nodes on that SCC are locked together in the first step.

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 $A_1, \ldots, A_p$.

The locking rules of the DDG policy are applied to the current state of the graph (rule L5). In contrast, the locking rules of the DAG policy are applied to the state of the graph when the transaction begins execution which is also the current graph. An example schedule produced by the DDG policy is given in Appendix A.

Let us now use Theorem 1 to prove the correctness of the DDG-SX policy. Due to the serial nature of the canonical schedule, we are able to analyze the execution of each transaction (as in Lemma 1) and its effect on the database in isolation (as in Lemmas 3–4), thus simplifying the correctness argument.

Before proceeding with the proof of the DDG-SX policy, we need the following definitions and lemmas. We define the *nearest common dominator* of two distinct nodes A and B in a rooted and connected graph G as the node C such that C does not lie on the same SCC as A and B, C dominates both A and B in G, and no proper descendant of C satisfies this property. We say that a set of nodes N *separates* nodes A and B in the structural database state G, if all the nodes in N are locked in exclusive mode and some ancestor of A or B in G that is also a descendant of their nearest common dominator in G belongs to N.

Lemma 1 *Suppose a transaction T_1 holds exclusive locks on a set of nodes N before another transaction T_2 begins execution in the database state G. If T_2 locks nodes A and B in G while T_1 has not performed any further lock or unlock operations then N does not separate A and B in G. (Proof Omitted.)*

Lemma 2 *Let N be the set of nodes and G be a rooted and a connected graph. If N does not separate A and B, but separates A and C then N also separates B and C. (Proof Omitted.)*

Theorem 2 *The DDG-SX policy is safe.*

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

For $1 \leq i \leq k$, let G_i be the state of the graph when T'_i begins its execution. Furthermore, let G_{k+1} be the structural database state when T'_k finishes its execution.

Let N be the set of nodes on which T_c holds an exclusive lock just before (but not including) the step (`L` A_k). Recall from Theorem 1 that T_c is a transaction in the canonical schedule with a special property: If T_c is the next transaction to execute a step after the schedule S' then we will have a non-serializable execution.

Lemma 3 *If N separates nodes A and B in some database state $G_i, c < i \leq k+1$, then N also separates nodes A and B in every database state G_i, $c < i \leq k+1$. (Proof Omitted.)*

Lemma 4 *If A is an entity unlocked by $T_c^{'}$ and $A \in G_{c+1}$ then N separates A and A_k in G_{c+1}. (Proof Omitted.)*

PROOF OF THEOREM 2 (CONTINUED). Let A be any entity unlocked by $T_c^{'}$. For $1 \leq i \leq k$, let B_i be the first entity locked by $T_i^{'}$. We prove that for each $i, c < i \leq k+1$, that if $A \in G_i$, N separates A from B_i in G_i. The proof is by induction on the length of the shortest path from $T_i^{'}$ to some $T_j^{'} \in \mathcal{K}$.

Base Case. The length of the shortest path in $D(S^{'})$ from $T_i^{'}$ to some $T_j^{'} \in \mathcal{K}$ is zero. Consider one such transaction $T_i^{'} \in \mathcal{K}$. Due to Lemma 4, N separates A and A_k in the database state G_{c+1}. Then due to Lemma 3, N must also separate A and A_k in G_i. Since $T_i^{'}$ locks B_i and A_k, due to Lemma 1, N does not separate B_i and A_k in G_i. Then, due to Lemma 2, N must separate A and B_i. This proves the base case.

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

Consider a transaction $T_i^{'}, i > c$, such that the length of the shortest path in $D(S^{'})$ from $T_i^{'}$ to some $T_j^{'} \in \mathcal{K}$ is $m, m > 0$. Let $T_j^{'}$ be a transaction such that $(T_i^{'}, T_j^{'}) \in D(S^{'})$ (such a transaction must exist due to condition 3c of Theorem 1 since $T_i^{'} \notin \mathcal{K}$). Therefore, there is an entity B that is locked by both $T_i^{'}$ and $T_j^{'}$. By induction hypothesis, N separates B_j and A in G_j. Since N does not separate B and B_j in G_j (due to Lemma 1), it separates B and A in G_j (due to Lemma 2). Then, due to Lemma 3, N separates B and A in G_i as well. Since N does not separate B_i and B in G_i (due to Lemma 1), and separates B and A in G_i, it separates B_i and A (due to Lemma 2).

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

Let $T_i^{'}, c < i \leq k$ be a transaction that locks A_c. Such a transaction must exist due to condition 3c of Theorem 1. A_c is unlocked by $T_c^{'}$ (by condition 2 of Theorem 1). As proven above, N separates A_c and B_i in G_i. But $T_i^{'}$ also locks A_c, and therefore, N cannot separate A_c and B_i in G_i (Lemma 1). This is a contradiction. Hence such a schedule $S^{'}$ does not exist and the DDG-SX policy must be safe. □

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. We assume that an `ACCESS` operation represents a `READ` immediately followed by a `WRITE`.

The last lock acquired by a transaction 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 locks an item which has been unlocked by T_j, and T_j has not reached its own locked point. The rules of altruistic locking are as follows:

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

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.

An example schedule produced by the altruistic locking policy is given in Appendix A. Unlike DDG-SX policy, altruistic locking does not impose any pre-specified order on the entities in the database, but on the other hand, the rule AL2 can be too restrictive, because to take advantage of the pre-release of locks a transaction has to be completely within the wake of another transaction. This may be relaxed, but only at the cost of several additional assumptions [SGMS94]. We now outline the correctness proof of the altruistic locking using Theorem 1. More details are given in Appendix B.

Theorem 3 *The altruistic locking policy is safe.*

Proof Outline. We assume the contrary and construct a a canonical non-serializable schedule as in Theorem 1. By induction we prove that in the schedule $S^{'}$, $T_i^{'}, i < c \leq k$ must be in the wake of T_c. Thus, $T_k^{'}$ is in the wake of T_c. Since $T_k^{'}$ locks A_k, A_k must have been previously locked and then unlocked by T_c. But T_c also locks A_k in the step immediately following $T_c^{'}$ which is a contradiction. Hence the assumed canonical non-serializable schedule does not exist and the 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. To distinguish the forest defined by the DTR policy from the directed graph that might exist in the database itself, we call the former a *database forest*. We first give some definitions and then define the rules of the DTR policy for the case when all locks are exclusive.

We say that a well-formed transaction T is *tree-locked* with respect to a forest G if except for the first lock step, each (`LX` A) step 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.

Recall that $R(T)$ is the set of entities referenced by a transaction. Let $A(T)$ be the set of entities for which there is an explicit `ACCESS`, `INSERT` or `DELETE` operation in T. These two sets can be different because a transaction locks some entities only to satisfy the rules of the locking policy. Let $G(e)$ be the database forest obtained by deleting e from 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. At the beginning of an active transaction T, join all the trees that contain some entity $v \in A(T)$ to form a single tree G. Add to G all the entities in $A(T)$ that are not already in it. Tree-lock T with respect to G.

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

An example schedule produced by the dynamic tree policy is given in Appendix A. Let us outline the correctness proof of this locking policy using Theorem 1. More details are given in Appendix B.

Theorem 4 *The Dynamic Tree policy is safe.*

PROOF OUTLINE. We assume the contrary and construct a canonical non-serializable schedule as in Theorem 1. For $1 \leq i \leq k$, let B_i be the first node locked by the transaction $T_i^{'}$. By induction, we prove that B_i is a descendant of A_k in G_i. Let $T_i^{'}, c < i \leq k$, be a transaction that locks A_c. Since B_i is a descendant of A_k, and T_c also locks A_c, B_c must be a descendant of A_c which is a contradiction. Hence the assumed canonical schedule cannot exist and the DTR policy must be safe. □

7 Summary and Conclusions

In this paper we studied the notion of canonical schedules for a dynamic database, that is, a database that undergoes insertions and deletions of entities. We showed that the shape of the serializability graph of a canonical non-serializable schedule in dynamic databases is more general than the shape of the corresponding serializability graph in static databases. We used the generalized canonical schedules to prove the correctness of three locking policies: DDG-SX policy, dynamic tree policy and altruistic locking policy. It is interesting to observe that these three locking policies were developed in different contexts and are based on quite different ideas. This work can be extended in several directions. The same technique could be used to analyze other locking policies [AA90, dFRH94]. Furthermore, generalization of these results to a dynamic hypergraph policy [Yan82] and incorporating recovery considerations in it [SWY93] remains an open problem and needs to be investigated in the future.

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.

[Cha94] Vinay K Chaudhri. *Transaction Synchronization in Knowledge Bases: Concepts, Realization and Quantitative Evaluation.* PhD thesis, University of Toronto, Toronto, 1994.

[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.

[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*, 1994. To Appear.

[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.

[Pap79] Christos Papadimitriou. Serializability of Concurrent Database Updates. *Journal of the Association for Computing Machinery*, 26(4):631–653, October 1979.

[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.

A Example Schedules

In this appendix we give examples of the application of the three locking policies considered in this paper. We use the notation `(U *)` to denote the release of all the locks by a transaction.

A.1 DDG-SX Policy

As an example application of the DDG-SX policy consider the database and the transactions shown in Figure 3. T_1 begins by locking node 1 (L4) in shared mode and then locks node 2 in exclusive mode (L5). It locks the nodes 3 and 4 which form a strongly connected component in shared mode in one step (L5). It is able to do so because the condition L5b is satisfied for each path from node 1 to node 3 and 4. It locks node 5 in exclusive mode and finishes execution. T_2 begins by locking both nodes 3 and 4 (L4) and then locks node 5 and finishes execution. If T_1 adds the edge $(2, 5)$ (L1), then T_2 will be unable to lock node 5 because in order to do that it must lock node 2 which is a predecessor of node 5 in the current state of the graph (L5a). T_2 must abort and start from node 2.

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

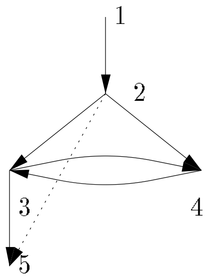


Figure 3: An Example Application of DDG-SX policy

A.2 Altruistic Locking

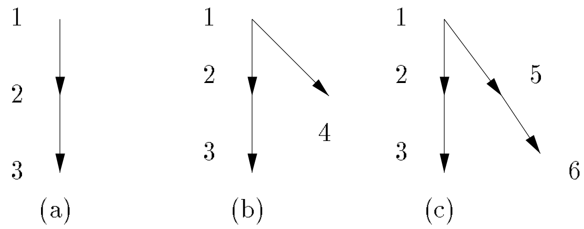
As an example of the application of the altruistic locking consider the schedule shown in Figure 4. Once T_1 releases lock on node 1, T_2 is able to lock node 1, thus entering the wake of T_1. From now on, T_2 can lock only those nodes that have been unlocked by T_1. Once, T_1 reaches its own locked point, when it locks node 3, T_2 is no longer in the wake of T_1 and can lock any item it needs.

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

Figure 4: An Example Application of Altruistic Locking

A.3 Dynamic Tree Policy

As an example of the application of the DTR policy, consider the schedule shown in Figure 5. When T_1 begins execution, the database forest is as shown in Figure 5(a) (DT0, DT2). Since T_2 accesses node 4, it is added to the database forest as shown in Figure 5 (DT1, DT2). Once T_2




```
T1: (LX 1,2) (U 1)        (LX 3) (U 2)                                        (U *)
T2:                 (LX 1,4) (U 4)        (LX 2) (U 1,2)
T3:                                                   (LX 1,5,6) (U 5,6) (LX 2) (U *)
```

Figure 5: An Example Application of Dynamic Tree Policy

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.

B Details of Proofs

Theorem 1:

Before we proceed with the proof of this theorem, we need the following lemma:

Lemma 5 *Let S be a legal and proper schedule of a transaction system τ. In S, let T_{ip} and T_{jq} be two contiguous subsequences of two distinct transactions T_i and T_j respectively such that T_{jq} appears immediately after T_{ip}. If $(T_i, T_j) \notin D(S)$, the schedule obtained by commuting T_{ip} and T_{jq} is legal and proper. (Proof Omitted.)*

Proof of Theorem 1.

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

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

Lemma 6 *τ contains a legal, proper, and non-serializable schedule S such that:*

(a) There exists a set of transactions $\mathcal{K} = \{T'_{\underline{k}+1}, \ldots, T'_k\}$ such that each of $T_1, \ldots, T_{\underline{k}}$ has a path to each $T_j \in \mathcal{K}$ and for all distinct transactions $T_i, T_j \in \mathcal{K}$ there is no path from T_i to T_j.

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

(Proof Omitted.)

Proof of Theorem 1 (Continued).

From now on we assume that the non-serializable schedule S of τ has the properties of Lemma 6. Let $RX(T_i)$ denote the set of entities locked by T_i in exclusive mode.

Without loss of generality, renumber the transactions in their topological order in $D(S^-)$. For $1 \leq i \leq k$, let $F_i = \big(RX(T_i) \cap [\cup_{(T_i,T_j)\in D(S)} R(T_j)]\big) \cup \big(R(T_i) \cap [\cup_{(T_i,T_j)\in D(S)} RX(T_j)]\big)$. Intuitively, F_i is the set of entities locked by T_i that are subsequently locked by some other transaction in conflicting mode in S. Each F_i is non-empty because the node corresponding to each of $T_1, \ldots, T_k$ has a successor in $D(S)$ (due to Lemma 6). By definition of S^+, A_k is the first entity of F_k locked by T_c and for $\underline{k} < i \leq k$, let $A_i = A_k$. For $1 \leq i \leq \underline{k}$, let A_i be the last item of F_i unlocked by T_i in S^+. ($T_i, 1 \leq i \leq \underline{k}$, unlocks at least one item in F_i, because each one of them has a successor in $D(S^+)$. $T_j, \underline{k} < j \leq k$ has unlocked at least A_k, because T_c locks A_k in conflicting mode in S^+, and thus, causes the edge $(T_j, T_c) \in D(S^+)$ (Lemma 6(b).)

We claim that the chosen T_is and A_is satisfy the conditions of the theorem. Condition (1a) is satisfied because there is a cycle in $D(S)$ implying a cycle in $G(\tau)$. Condition (2) is satisfied by the choice of k, and the fact that T'_c unlocks A_c (as argued in the previous paragraph, $c \leq k$), and because T_c locks A_k just after the schedule S^- (causing an edge (T_k, T_c) in $D(S^+)$ — due to Lemma 6(b)).

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

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

For $1 \leq i \leq k, 1 \leq j \leq l_i$, let $T_{i,j}^{'}$ be the sub-transactions of $T_i^{'}$ in S^-. For $1 \leq i \leq k, 1 \leq j \leq l_i$, inductively define $S_{i,j}$ as follows. Intuitively, in the sequence of schedules $S_{i,0}, S_{i,1}, \ldots, S_{i,l_i}$, we start with S_i and move the sub-transactions of $T_{i+1}^{'}$ one by one towards the beginning so as to follow the steps of $T_1^{'}, \ldots, T_i^{'}$. More formally, $S_{i,0} = S_i$. For $j > 0$, $S_{i,j}$ is obtained inductively from $S_{i,j-1}$ by moving the steps of $T_{i+1,j}^{'}$ backwards so that they follow directly after all the steps of $T_1^{'}, T_2^{'}, \ldots, T_i^{'}, T_{i+1,1}^{'}, T_{i+1,2}^{'}, \ldots, T_{i+1,j-1}^{'}$. For all $1 \leq i < k$, $S_{i,l_i} = S_{i+1}$.

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

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

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

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

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

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

We show that $S_{m,n+1}$ is legal and proper as well. To prove this, we need to show that $T_{m+1,n+1}^{'}$ can be commuted backward with any $T_{p,q}^{'}$, $m + 1 < p \leq k, 1 \leq q \leq l_p$ without violating legality and properness. Furthermore, we need to show that $D(S_{m,n+1}^-) = D(S^-)$.

Because the transaction indices are assigned in their topological order in $D(S^-)$, there is no path from T_p to T_{m+1} in $D(S^-)$ and consequently in $D(S_{m,n}^-)$. Therefore, by Lemma 5, $T_{m+1,n+1}^{'}$ can be commuted backward with any $T_{p,q}^{'}$, $m + 1 < p \leq k, 1 \leq q \leq l_p$ without violating properness or legality. When this series of commutations has occurred, the resulting schedule is by definition $S_{m,n+1}$. Thus, $S_{m,n+1}$ is legal and proper. Furthermore, since there is no edge from T_p to T_{m+1} in $D(S_{m,n})$, $T_{m+1,n+1}^{'}$ and $T_{p,q}^{'}$ do not access any conflicting entities. Therefore, $D(S_{m,n+1}^-)=D(S_{m,n}^-)=D(S^-)$.

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

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

Since $D(S_k^-) = D(S^{'}) = D(S^-)$, by Lemma 6(a), each of $T_1, \ldots, T_{\underline{k}}$ has a path in $D(S^{'})$ to each transaction in $\mathcal{K}$, and for all $T_i, T_j \in \mathcal{K}, i \neq j$, there is no path in $D(S^{'})$ from T_i to T_j. Also, the topological order of transactions in $D(S^{'})$ is the same as in $D(S^-)$. This proves condition 3c.

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

Theorem 3: The altruistic locking policy is safe.

PROOF.

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

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

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

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

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

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

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

Theorem 4: The Dynamic Tree policy is safe.

PROOF.

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

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

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

The following lemma states some simple properties of the DTR policy.

Lemma 7 *Let T be any transaction following the DTR policy, B be the first entity locked by T, and G be the state of the database forest 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. (b) For* *each A in $R(T)$, all nodes that are proper ancestors of A and descendants of B in G, have been referenced by T when it locks A. (Proof Omitted.)*

Lemma 8 *A_k is a successor of f in the database forests $G_{c+1}, \ldots, G_{k+1}$. (Proof Omitted.)*

Lemma 9 *If a node A is a proper descendant of A_k in some $G_i, c < i \leq k+1$, then for all $j, c \leq j < i$, if $A \in G_j$, A is a proper descendant of A_k in G_j. (Proof Omitted.)*

Proof of Theorem 4 (Continued).

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

Base case. $i = k$. Since f is a predecessor of A_k in G_k (due to Lemma 8), and T_k could not have locked f (because it is locked by T_c), and it locks A_k, A_k must be the first entity locked by T_k (because T_k is tree-locked). This proves the claim for the base case.

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

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

We first prove that $T_m^{'}$ did not lock any descendants of A_k in G_m. For contradiction, suppose $T_m^{'}$ locked a descendant A of A_k in G_m. Then $T_m^{'}$ must also lock f because of Lemma 7(b). But $T_m^{'}$ cannot lock f because it is already locked by T_c. Hence $T_m^{'}$ cannot lock any descendant of A_k in G_m.

Next, from $D(S^{'})$, pick a transaction $T_j^{'}$ such that $(T_m^{'}, T_j^{'}) \in D(S^{'})$ (such a transaction must exist due to condition 3c of Theorem 1 and because $m \neq k$). By induction hypothesis, B_j is a descendant of A_k in G_j. Therefore $T_j^{'}$ locked only descendants of B_j, and hence only descendants of A_k in G_j (by Lemma 7(a)). The descendants of A_k in G_j were also descendants of A_k in G_m if they existed in G_m (due to Lemma 9). Thus, $T_j^{'}$ locked only descendants of A_k in G_m.

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

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

Let $T_i^{'}, c < i \leq k$ be a transaction that locks A_c (such a transaction must exist because of condition 3c of Theorem 1). As proven above, B_i is a descendant of A_k in G_i. Therefore, A_c is also a descendant of A_k in G_i and also in G_c (due to Lemma 9). Since $T_c^{'}$ locks a parent of A_k and also locks A_c which is a descendant of A_k, it must also lock A_k (due to Lemma 7(b)). But we know that T_c locks A_k in the step immediately following $T_c^{'}$. This is a contradiction, because a tree-locked transaction may lock an entity only once. Hence DTR policy must be correct. □