MIT/LCS/TM-185

AN OPTIMALITY THEORY OF CONCURRENCY CONTROL FOR DATABASES

Hsing-Tsung Kung

Christos H. Papadimitriou

November 1980

An Optimality Theory of Concurrency Control for Databases

H. T. Kung

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

C. H. Papadimitriou

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

April 1979

[Last revised September 1980]

(This paper is issued simultaneously as a CMU and MIT Technical Memorandum)

KEYWORDS AND PHRASES: Database Concurrency Control, Schedulers, Performance vs. Information of Schedulers, Optimal Schedulers.

This research is supported in part by the National Science Foundation under Grants MCS 75-222-55, MCS 77-01193, MCS 77-05314, the Office of Naval Research under Contract N00014-76-C-0370, and a Miller Fellowship.

Abstract

A concurrency control mechanism (or a scheduler) is the component of a database system that safeguards the consistency of the database in the presence of interleaved accesses and update requests. We formally show that the performance of a scheduler, i.e., the amount of parallelism that it supports, depends explicitly upon the amount of information that is available to the scheduler. We point out that most previous work on concurrency control is simply concerned with specific points of this basic trade-off between performance and information. In fact, several of these approaches are shown to be optimal for the amount of information that they use.

1. Introduction

A database system may interact with many transactions in an interleaved manner. Even if we assume that each such individual transaction is *correct* (that is, it preserves the consistency of the databases when run by itself), the interleaved mode of operation may result in inconsistencies (see, for example, [2]). It is the task of the *concurrency control mechanism* of the database system, called *scheduler* in this paper, to safeguard the consistency of the database by granting or rejecting the execution of atomic steps of transactions, when requests for such executions are made.

The design of schedulers for databases has proved to be a non-trivial problem, and some theoretical work on the subject has appeared (see, for example, [2, 6, 7, 9]). Several solutions to this problem have been proposed under a variety of assumptions. In this paper, we give a uniform framework for evaluating these solutions, and, in some cases, for establishing their optimality. A scheduler is evaluated in terms of its *performance*, which is measured by the set of request sequences that the scheduler can authorize for execution without any delay. This set of request sequences is called the *fixpoint set* of the scheduler. The idea is that the richer this set is, the more likely that no delays will be imposed by the scheduler. In this sense the fixpoint set is a fair measure of the *parallelism* supported by the scheduler, and therefore of its performance.

We observe that there is a trade-off between scheduler performance and the *information* used by the scheduler. The latter is the minimum knowledge about the database and the transactions that the scheduler requires in order to function correctly. Typical information that could be useful to the scheduler is *syntactic information* about the transactions (that is, a flowchart with the names of the database entities accessed and updated at each step); or *semantic* information about the meaning of the data and the operations performed; or the *integrity constraints*, the consistency requirements that the database must satisfy. Ideally, a scheduler would like to have a perfect knowledge of all these three components of information. It is usually necessary, however, to have the scheduler operate at some imperfect level of information. There are many reasons for this. Some information (e.g., integrity constraints) may not be known explicitly even to the designer of the database. If semantic information is given in some powerful enough language (e.g., arithmetic) then it may not be possible to reason about it effectively. Finally, to utilize sophisticated information may render the scheduling problem combinatorially intractable -- see [6] for a case in which the ability of simply distinguishing between read and write operations makes the problem NP-complete. It should be intuitively clear that the more information the scheduler has, the better job it can do in enriching its fixpoint set, and therefore increasing its performance. We capture this intuitive trade-off in an equation (Theorems 3.1) and exhibit several specific instances for which well known concurrency principles correspond to optimal schedulers (optimal with respect to the information that they use). For example, in our framework we can formally show that serializability (which has been adapted in an ad hoc manner in virtually all the concurrency control literature) is indeed the right notion of correctness when only *syntactic* information is available (as is usually the case). If semantic or integrity information is available, then more liberal correctness criteria may be used (see, for example, [3, 4]). We also prove that some strict version of the two-phase locking technique of [2] is the best possible principle when syntactic information is acquired in an incremental manner.

The paper is organized as follows. In Section 2 we introduce our model for transaction systems, carefully distinguishing among the syntactic, semantic, and integrity constraint components. In Section 3 we formally introduce the notion of schedulers, and develop the basic tools for studying the information vs. performance trade-off. Specific examples of optimal schedulers are presented in Section 4.

2. Transaction Systems

2.1 Definition of a Transaction System

By a transaction system we mean a database (that is, data and integrity constraints) together with a set of statically prespecified transaction programs. A transaction system can be formally defined in terms of three components: syntax, semantics, and integrity constraints.

2.1.1. Syntax

A *transaction system* T is a finite set of *transactions*, $\{T_1, ..., T_n\}$, $n > 1$,where each transaction T_i is a finite sequence of *transaction steps*, $T_{i1}, ..., T_{im_i}$. The n-tuple of integers $(m_1, ..., m_n)$ is called the *format* of the transaction system. For simplicity, we assume that all transaction systems under consideration have the same, fixed format.

The transactions in a transaction system operate on a set of *variable names*. The variables are abstractions of data entities, whose granularity is not important for our development. The variables can represent bits, files or records, as long as they are individually accessible. The set of variable names is denoted by V. Besides the (global) variables in V, each transaction T_i is associated with local variables, $t_{i1}, ..., t_{im_i}$. A transaction step T_{ij} in T_i can be thought of as the *indivisible* execution of the following two instructions:

$$t_{ij} \leftarrow x_{ij},$$
$$x_{ij} \leftarrow f_{ij}(t_{i1}, ..., t_{ij}),$$

where f_{ij} is a j-place function symbol. That is to say, at step T_{ij} the current value of some global variable $x_{ij} \in V$ is stored at a local place t_{ij} and then x_{ij} is assigned a new value, based on function f_{ij} and knowledge available to the transaction T_i at this time, namely, the values of all "declared" local variables $t_{i1}, ..., t_{ij}$. The meaning of function f_{ij} is open to arbitrary interpretations at this point. For example, it could be the identity function on t_{ij}, in which case T_{ij} is simply a read step. Similarly, if all $f_{ik}(t_{i1}, ..., t_{ik})$ with $k \geq j$ are independent of t_{ij}, then T_{ij} is a write step.

2.1.2. Semantics

Associated with each variable name $v \in V$ we have an enumerable set $D(v)$, the domain of v, consisting of all possible values that the variable v can assume -- typically the integers, the Boolean values, or finite strings. A local variable t_{ij} has always the same domain as x_{ij}.

A *state* of a transaction system T is a triple (J, L, G), where

- J is an n-tuple of integers $(j_1, \ldots, j_n)$ with j_i, $(1 \le j_i \le m_i+1)$, specifying the next step of transaction T_i. The j_i's are thus program counters. If $j_i = m_i+1$, then transaction T_i has terminated.
- L is an element in $\Pi_{1 \le i \le n}(\Pi_{1 \le j < j_i} D(x_{ij}))$ representing the values of all declared local variables.
- G is an element in $\Pi_{v \in V} D(v)$ representing the current values of all global variables $v \in V$.

The semantics of T associate with the function symbol f_{ij} at each step T_{ij} a function $\varphi_{ij} : \Pi_{1 \le k \le j} D(x_{ik}) \rightarrow D(x_{ij})$, which is the *interpretation* of f_{ij}. Thus the execution of a transaction step maps one state of the transaction system into another one. More precisely, if transaction step T_{ij} is eligible for execution at state (J, L, G), that is, if $j_i \le m_i$, then its execution modifies the three components of the state as follows:

$$\begin{gathered} j_i \leftarrow j_i + 1, \\ t_{ij} \leftarrow x_{ij}, \\ x_{ij} \leftarrow \varphi_{ij}(t_{i1}, \ldots, t_{ij}). \end{gathered}$$

This view of single transaction steps can be extended to sequences of transaction steps in the obvious way.

2.1.3. Integrity Constraints

The *integrity constraints* of a transaction system T correspond to a subset IC of the product $\Pi_{v \in V} D(v)$. A state (J, L, G) of T is said to be *consistent* if G belongs to IC. A sequence of transaction steps is said to be *correct* if a serial execution of the steps in the sequence will map *any* consistent state of the transaction system into a consistent state.

The *basic assumption* throughout the paper is that all transactions in a transaction system are correct.

2.2 Example

Consider a transaction system consisting of three transactions T_1, T_2, and T_3, that access two banking accounts A and B in the following way:

- T_1 transfers \$100 from A to B if A has enough funds and the balance of B is below \$100.
- T_2 withdraws \$50 from B and increments a counter C, if B has enough funds.
- T_3 is an auditing transaction that computes the sum S of A and B, and sets the counter C back to 0.

Syntax. The set of global variable names is $V = \{A, B, S, C\}$. The x_{ij}'s are as follows:

$x_{11} = A, x_{12} = B, x_{13} = A$
$x_{21} = B, x_{22} = C,$
$x_{31} = A, x_{32} = B, x_{33} = S, x_{34} = C$

Thus the format of the transaction system is (3, 2, 4).

Semantics. For all $v \in V$, $D(v)$ is the set of natural numbers. Typical states would be as follows:

- $(J, L, G) = ((1, 1, 1), *, (150, 50, 200, 0))$. This is a possible state before any of the transactions has started execution. We have A = \$150, B = \$50, S = \$200, C = 0, and don't care about the values of local variables.
- $(J, L, G) = ((2, 2, 4), (150; 50; 150, 0, 200), (150, 0, 150, 0))$. In this state, A has not been decreased but B has. The new S has been computed but C has not.

As for the operations performed by each step:

$\varphi_{11} = t_{11}$
$\varphi_{12} = \textit{if}\ t_{11} \geq 100\ \textit{and}\ t_{12} < 100\ \textit{then}\ t_{12} + 100\ \textit{else}\ t_{12}$
$\varphi_{13} = \textit{if}\ t_{11} \geq 100\ \textit{and}\ t_{12} < 100\ \textit{then}\ t_{11} - 100\ \textit{else}\ t_{11}$

$\varphi_{21} = \textit{if}\ t_{21} \geq 50\ \textit{then}\ t_{21} - 50\ \textit{else}\ t_{21}$
$\varphi_{22} = \textit{if}\ t_{21} \geq 50\ \textit{then}\ t_{22} + 1\ \textit{else}\ t_{22}$

$\varphi_{31} = t_{31}$
$\varphi_{32} = t_{32}$
$\varphi_{33} = t_{31} + t_{32}$
$\varphi_{34} = 0$

The *integrity constraints* may very well be the set of states for which $A \geq 0$, $B \geq 0$, and $A + B = S - 50C$.

3. Schedules and Schedulers

3.1 Schedules

A *schedule* of a transaction system T is a permutation of the set $\{T_{ij}: 1 \leq i \leq n, 1 \leq j \leq m_i\}$ of steps in T such that in the permutation T_{ij} comes before T_{ik} for $j < k$. We may think of a schedule as a possible stream of arriving execution requests, or, in a different context, as a sequence of transaction steps that defines the order in which these execution requests are granted execution. The set of all schedules of T is denoted by H(T). Since this set depends only on the format of T and the format is assumed fixed, we shall write H for H(T). A schedule is said to be *correct* if its execution preserves the consistency of the database. The set of all correct schedules of T is denoted by C(T). The set C(T) is always nonempty, since it at least contains (by our basic assumption that all transactions are correct) all *serial schedules*, that is, all permutations π such that $\pi(T_{i,j+1}) = \pi(T_{ij}) + 1$ for $1 \leq i \leq n$ and $j \leq m_i-1$.

3.2 Schedulers

A *scheduler* (or concurrency control mechanism) transforms a stream of execution requests into a *correct* schedule. This is achieved by properly granting or rejecting the execution of arriving requests. (A rejected request is rescheduled for execution at some later time.) Thus, a scheduler for a transaction system T can be viewed as a mapping S from H to C(T).

We measure the performance of a scheduler S by its *fixpoint set* P_S, defined as

$$P_S = \{h \in H: S(h) = h\}.$$

Clearly P_S must be a subset of C(T). The larger P_S is, the more improbable it is that S will have to delay (or reject) the execution of a transaction step, after such an execution is requested. We therefore consider the inclusion-induced partial order on the sets P_S as a "qualitative" measure of scheduler performance.

3.3 Information

A *level of information* available to a scheduler S about a transaction system T is defined to be a set I of transaction systems $\{T, T', T'', ...\}$ that contains T. Intuitively, if S is kept at this level of information, it knows that the transaction system in question is among the transaction systems in I, but does not know exactly which. Thus, S has to be a scheduler for *all* transaction systems $T' \in I$. For example, the set I could be the set of all transaction systems that have the same syntax. This level of information corresponds to the case that a scheduler has complete syntactic information, but no other information.

Alternatively, we could view I as a *function* that maps any transaction system T to an object I(T) ($\in \{0, 1\}^*$). Intuitively, I(T) is the *information extracted from T by the operator I*; for example, I(T) could be an encoding of the syntax of T. The effect would be that T cannot be distinguished from the transaction systems T′ that have the same image I(T); in the notation of the previous paragraph, which we are going to follow henceforth, $I = \{T': I(T') = I(T)\}$.

The maximum possible information that a scheduler can have is, of course, the complete syntactic, semantic and integrity information about the transaction system in question; this corresponds to $I = \{T\}$. The minimum information is the format $(m_1, ..., m_n)$; this corresponds to I being the set of all transaction systems of the given format, with the single restriction that the transactions be correct -- by our basic assumption. The more information available to the scheduler, the "better" scheduling results may be expected. We formally capture this idea in the following theorem:

Theorem 3.1: For any scheduler S using information I, the fixpoint set P_S must satisfy:

$$P_S \subseteq \bigcap_{T' \in I} C(T').$$

The proof of this theorem uses a general adversary argument, instances of which we shall see many times in the rest of the paper. The proof goes as follows. If there is a schedule $h \in P_S$ and a transaction system $T' \in I$ such that h is not correct for T′ that is, $h\ (=S(h)) \notin C(T')$, then an adversary could "fool" the scheduler S by choosing T′ for S to handle, and giving h as the stream of execution requests. The resulting state after the execution can be inconsistent, since $S(h) \notin C(T')$. Thus, the scheduler is incorrect.

As a corollary of Theorem 3.1, the maximum-performance scheduler using information I is the one that has its fixpoint set $P = \bigcap_{T' \in I} C(T')$. We call this scheduler the *optimal scheduler* for the level of information I. (Notice that in practice there may be insurmountable difficulties — such as the negative complexity results in [6] — in realizing the optimal scheduler for a given level of information.) The concept of information introduced here partially orders schedulers with respect to their sophistication: we say that S is more sophisticated than S′ if S operates at a level of information I that is properly *included* in the level of information I′ of S′, that is, $I \subseteq I'$. On the other hand, schedulers are also partially ordered with respect to their performance: we say that S performs better than S′ if $P_S \supseteq P_{S'}$. Then the mapping from any level of information I to the fixpoint set of the optimal scheduler for I, $\bigcap_{T' \in I} C(T')$, is a natural *isomorphism* between these two partially ordered sets. This captures the fundamental trade-off between scheduler information and performance, that is, if $I \subseteq I'$ then $P_{S'} \supseteq P_S$ for the optimal schedulers S and S′ for I and I′, respectively.

In the next section, we present several examples of schedulers that are optimal for different levels of information.

4. Optimal Schedulers

4.1 Optimal Schedulers for Extreme of Information

Maximum Information

This is the case when complete information on the transaction system T in question is available to the scheduler. The information level I in this case is a singleton set, i.e., $I = \{T\}$. We can therefore define the scheduler S, in principle at least, such that $P_S = C(T)$. This is the optimal scheduler for the maximum level of information.

Minimum Information

If we only know the format of T, then we have the poorest possible level of information. What is the best possible scheduler in this case? Consider a *serial scheduler* S which is defined to be a scheduler satisfying the following property for any T:

$$S(H) = \{\text{all serial schedules of } T\} \text{ and } P_S = \{\text{all serial schedules of } T\},$$

where serial schedules are defined in Section 3.1. By our basic assumption that each transaction is correct, we see that each schedule in S(H) is correct.

Theorem 4.1: The serial scheduler S is optimal among all schedulers using the minimum information.

Proof: Suppose that S is not optimal. Then there must exist a non-serial schedule in C(T) in which some steps T_{ik}, T_{j1}, $T_{i,k+1}$ in T are executed in this order. Note that because of the minimum information assumption, I may contain transaction systems with any integrity constraints and interpretations for steps. We assume that the integrity constraints for some transaction system T′ in I correspond to "$x=0$", and that the interpretations of function symbols are such that T_i is $\{T_{ik}\colon\ x \leftarrow x+1,\ T_{i,k+1}\colon\ x \leftarrow x-1\}$ and T_j is $\{T_{j1}\colon\ x \leftarrow 2x\}$. We see that T_i and T_j are correct, but the sequence $\{T_{ik}, T_{j1}, T_{i,k+1}\}$ is not correct for it may transform a consistent state, $x=0$, into an inconsistent state, $x=1$. Thus, the schedule is not in C(T′). This contradiction implies that for the minimum information case, the only correct schedules that a scheduler can produce are serial schedules. Hence, the serial scheduler defined above is optimal. □

4.2 Optimal Schedulers for Complete Syntactic Information

Suppose now that all syntactic information is available; that is, the information level has the property that I is the set of all transaction systems with the same syntax. As in a similar situation in the theory of program schemata, one can supplement this syntax with canonical semantics called Herbrand semantics (see [5] for a detailed exposition). For all $v \in V$, the domain D(v) is the set of all strings from the alphabet $\Sigma = V \cup \{f_{ij}: i=1, \ldots, n; j=1, \ldots, m_i\}$ plus the symbols ")", "(", ",". If $a_1, \ldots, a_j$ are elements of D(v), then $\varphi_{ij}(a_1, \ldots, a_j)$, the interpretation of f_{ij}, is the string $f_{ij}(a_1, \ldots, a_j)$. In other words, the Herbrand interpretation captures all the history of the values of all global variables. We say that a schedule h is *serializable* if its execution results are the same as the execution results of some serial schedule under the Herbrand semantics. Since serial schedules are correct, so are serializable schedules. By SR(T) we denote the set of all serializable schedules of T. A *serialization scheduler* is defined to be a scheduler S satisfying the following property for any T:

$$S(H) = SR(T) \text{ and } P_S = SR(T).$$

Theorem 4.2: A serialization scheduler is optimal among all schedulers using complete syntactic information.

Proof: To prove the optimality, for any schedule $h \notin SR(T)$, we shall define a transaction system $T' \in I$ such that $h \notin C(T')$. The semantics of T' are the Herbrand interpretation. Now, for the integrity constraints, we define IC as follows. Assume that T is consistent initially. Let $(v_1, ..., v_k)$ be the initial values of global variables in V, where $k = |V|$. If $a_1, ..., a_k$ are in D(v), we say that $(a_1, ..., a_k) \in IC$ iff there exists a serial concatenation Q (possibly empty) of some transactions in T' such that the initial values $(v_1, ..., v_k)$ are transformed by Q to $(a_1, ... a_k)$. By this definition, all transactions are individually correct, and our basic assumption holds. Now, it is easy to see that, if h is any schedule, not in SR(T), then it transforms the initial values $(v_1, ..., v_k)$ to a set of values not in IC. Hence, $h \notin C(T')$. □

The theorem shows that even if complete syntactic information of a transaction system T is available to a scheduler, SR(T) is the maximum possible set of correct schedules a scheduler can hope to produce. After all syntactic information is the information one can easily extract in a transaction system, by having the users declare the files that they intend to open, say. It is therefore not at all surprising that most approaches to concurrency control have serialization as their goal [2, 8, 7, 1, 6].

4.3 Optimal Schedulers for Complete Semantic Information but Integrity Constraints

Consider the transaction system of Fig. 4-1.

T_1	T_2
T_{11}: $x \leftarrow x+1$	T_{21}: $x \leftarrow x+1$
T_{12}: $x \leftarrow 2*x$	

Figure 4-1: A transaction system.

The schedule $h = (T_{11}, T_{21}, T_{12})$ is not serializable since the Herbrand values for x of the two serial histories are $f_{12}(f_{11}(f_{21}(x)))$ and $f_{21}(f_{12}(f_{11}(x)))$, whereas that of h is $f_{12}(f_{21}(f_{11}(x)))$. But with the given interpretations of the f_{ij}'s, h is seen to produce the same state as the serial history (T_{21}, T_{11}, T_{12}). Hence, our knowledge of the interpretations allows us to expand the set of correct schedules. It is not hard to see, however, that the gains are delimited by a generalized notion of serialization, defined as follows. A schedule h is said to be *weakly serializable*, if starting from *any* state E the execution of the schedule will end with a state which is achievable by the execution of some concatenation of transactions in T, possibly with repetitions and omissions of transactions, also starting from state E. Since transactions are assumed to be correct, a weakly serializable schedule is correct. Denote by WSR(T) the set of all weakly serializable schedules of T. It is clear that $SR(T) \subseteq WSR(T)$. A *weak serialization scheduler* is defined to be a scheduler S satisfying the following property for any T:

$$S(H) = WSR(T) \text{ and } P_S = WSR(T).$$

Theorem 4.3: A weak serialization scheduler is optimal among all schedulers using all information but the integrity constraints.

The proof is quite similar to the proof of Theorem 4.2, and is omitted.

4.4 Optimal Schedulers for Dynamic Syntactic Information

So far we have implicitly assumed that the information of a scheduler about a transaction system is *static* in nature, that is, prespecified and fixed. We now consider the case that information is *dynamic*, that is, the amount of information available to a scheduler increases as the scheduler proceeds. We restrict ourselves mainly to the important case of *dynamic syntactic information.*

At a given state (J, L, G) of a transaction system T, the dynamic syntactic information available to a scheduler is the complete syntactic information on all transaction steps T_{ij}'s with $1 \leq i \leq n$, $1 \leq j \leq j_i$ and on those T_{ij_i+1}, $1 \leq i \leq n$, which are pending for execution. Thus, the set I corresponding to this level of information consists of all transaction systems of the given format that are syntactically identical to the one at hand up to the specified points. We can define by a straightforward generalization of the definition of P_S, the fixpoint set P_D of an optimal scheduler that uses dynamic syntactic information. By Theorem 4.2, we know that P_D must be contained in the set SR(T) of serializable schedules of T. Theorem 4.4 below characterizes P_D exactly.

Optimal schedulers for dynamic syntactic information are closely related to schedulers that are implemented by the well-known two-phase locking policy [2], which is defined informally as follows. (a) If a transaction accesses $x \in V$, then there is a *lock* x step before the first access of x and an *unlock* x step after the last, and (b) no *lock* step appears in any transaction after the first *unlock* step. Thus each transaction has two phases: the locking phase, during which no locks can be released, and the unlocking phase, during which no locks may be requested. Notice that rules (a) and (b) do not uniquely specify the positions of *lock-unlock* steps.

A *two-phase locking scheduler* is simply a scheduler that treats transactions as though they were locked according to some version of the two-phase locking policy. The fact that schedules output by a two-phase locking scheduler are all correct follows from a proof in [2]. The following version of the two-phase locking policy can be implemented by a scheduler using dynamic syntactic information.

A strong two-phase locking policy. For any $x \in V$, *lock* x is always inserted immediately before the first access of x, and *unlock* x occurs only after the last step of a transaction (or immediately before, in case that the last step does not update x.)

Theorem 4.4: A two-phase locking scheduler corresponding to the strong two-phase locking policy is optimal among all schedulers using dynamic syntactic information.

Proof: Suppose that h is a schedule not belonging to the fixpoint set of the two-phase locking scheduler defined in the theorem. Then there must exist transaction steps in h, say T_{1j_1} and T_{2j_2}, such that

- T_{1j_1} is not the last step of transaction T_1, i.e., $j_1 < m_1$,
- T_{1j_1} and T_{2j_2} access the same variable x, and
- these steps are scheduled in h in the order $T_{1j_1}, ..., T_{2j_2}, ..., T_{1m_1}$, and either T_{1m_1} updates x, or T_{1m_1} was not pending when T_{2j_2} was scheduled for execution.

We can construct a transaction system $T = \{T_1, T_2\}$, compatible with the syntactic information that was available at the moment when T_{2j_2} was scheduled, such that $h \notin C(T)$. Transaction system T is defined as follows:

T_{1j_1}: $x \leftarrow x + 1$,
T_{2j_2}: $x \leftarrow 2 * x$,
T_{1m_1}: $x \leftarrow x - 1$,

all other steps are read steps, i.e., T_{ij}: $x_{ij} \leftarrow x_{ij}$, and the integrity constraints is "$x = 0$." It is readily seen that schedule h is not a correct one for T, and thus the theorem follows. □

Note that the scheduler in Theorem 4.4 need not really insert *lock*'s and *unlock*'s into transactions, as it can just keep track the first occurrence of each variable in each transaction.

If a scheduler is given additional dynamic information, i.e., (a) the *read-completion information* -- indicating the earliest point that a transaction has read all the global variables that it ever wants to access, and (b) the *last-use information* -- indicating for each global variable in V the point in a transaction that the variable is used (read or written) for the last time, then the scheduler may enjoy higher performance. Using the read-completion and the last-use information the following version of the two-phase locking policy can be implemented by a scheduler.

A weak two-phase locking policy. For any $x \in V$, *lock* x is always inserted immediately before the first access of x and *unlock* x occurs as early as possible, as long as the two-phase locking requirement is still maintained.

Theorem 4.5: A two-phase locking scheduler corresponding to the weak two-phase locking policy is optimal among schedulers using dynamic syntactic information plus the read-completion and the last-use information.

Proof: Suppose that h is a schedule not belonging to the fixpoint set of the two-phase locking scheduler defined in the theorem. Then there must exist transaction steps, say T_{1j_1} and T_{2j_2} in h, such that (a) these steps are scheduled for execution in the above order, (b) T_{1j_1} and T_{2j_2} both access the same variable x, and (c) T_{1j_1} is not the last step in transaction T_1 that uses x, or (c') T_{1j_1} is not after the read-completion point for T_1. For the case of (c) we define the transaction system T to be the same as the one used in the proof of Theorem 4.4. For the case of (c'), we define the transaction system $T = \{T_1, T_2\}$ to be such that

T_{1j_1}: $x \leftarrow x + 1$,
T_{2j_2}: $y \leftarrow 2 * y$,
T_{1m_1}: $y \leftarrow y + 1$,
T_{2m_2}: $x \leftarrow 2 * x$,

all other steps are read steps, i.e., T_{ij}: $x_{ij} \leftarrow x_{ij}$, and the integrity constraints is "$x = y$." We see that the transaction system T defined in either case is compatible to the syntactic information available at the moment when T_{2j_2} is scheduled for execution while T_{1m_1} is not yet pending, and that schedule h is not a correct one for T. □

References

[1] Berstein, P.A., Goodman, N., Rothnie, J.B. and Papadimitriou, C.H.
A System of Distributed Databases (the Fully Redundant Case).
IEEE Transactions on Software Engineering SE-4:154-168, March, 1978.

[2] Eswaran, K.P., Gray, J.N., Lorie, R.A. and Traiger, I.L.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-633, November, 1976.

[3] Kung, H.T. and Lehman, P.L.
Concurrent Manipulation of Binary Search Trees.
ACM Transactions on Database Systems 5(3):354-382, September, 1980.
An extended abstract appears in the *Proc. of the Fourth International Conference on Very Large Databases.*, September 1978.

[4] Lamport, L.
Towards a Theory of Correctness for Multi-user Data Base Systems.
Technical Report CA-7610-0712, Massachusetts Computer Associates, Inc., October, 1976.

[5] Manna, Z.
Mathematical Theory of Computation.
McGraw-Hill, New York, 1974.

[6] Papadimitriou, C.H.
The Serializability of Concurrent Updates.
Journal of the ACM 26(4):631-653, October, 1979.

[7] Silberschatz, A. and Kedem, Z.
Consistency in Hierarchical Database Systems.
Journal of the ACM 27(1):72-80, January, 1980.

[8] Stearns, R.E., Lewis, P.M. II and Rosenkrantz, D.J.
Concurrency Control for Database Systems.
In *Proc . Seventh Annual Symposium on Foundations of Computer Science*, pages 19-32. IEEE, 1976.

[9] Yannakakis, M., Papadimitriou, C.H. and Kung, H.T.
Locking Policies: Safety and Freedom from Deadlock.
In *Proc. Twentieth Annual Symposium on Foundations of Computer Science*, pages 286-297. IEEE, 1979.