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

1. Introduction

In many database applications it is desirable that the database system be time-shared among multiple users who access the database in an interactive way. In such a system the arriving requests for the execution of steps in different transactions from different users may by interleaved in any order. Assume that each transaction is correct in the sense that it preserves the consistency of the database when executed alone. The execution of many correct transactions in an interleaved order may, however, bring a consistent database state into an inconsistent one (see, e.g., [Eswaran et al. 76]). It is the task of the concurrency control mechanism of the database system, which is also called scheduler in this paper, to safeguard the database consistency by properly granting or rejecting the execution of arriving requests. A rejected request is scheduled for execution after some requests which arrive later have been scheduled for execution. That is, the concurrency control enforces database consistency by delaying the execution of some requests when this is necessary.

Although system consistency is the primary objective of concurrency control, there are certain other important considerations that must be taken into account in its design. For instance, one sure way to secure consistency would be to delay all other user requests until the first user logs out, then let the second user go, and so on. Since each individual transaction is correct, the execution of requests in this order will preserve consistency. Obviously, this straight-forward mechanism has a major deficiency: it may cause unnecessary delays for all but one user, and thus degrade the throughput and response time of the system. This scheduler, however, does have one important advantage. Namely, it requires no information about the transactions except for a user identification for each request. We see therefore that it is necessary to consider the performance of a scheduler and the information that it uses, in addition to its correctness.

Performance. We measure the performance of a scheduler by the set of request sequences which the scheduler can pass without any delay. We call this set 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 to the user requests. In fact, if the fixpoint set of a scheduler strictly includes that of another scheduler, then it can be argued that the former scheduler performs strictly better than the latter one as far as average delays are concerned. Further justification of this measure, as well as a discussion of its limitations appears in Section 6.

Information. The information used by a scheduler is the minimum knowledge about the database and the transactions that it requires in order to function correctly. Typical information that would be useful to the scheduler is syntactic information about the transactions (i.e., a flowchart with the names of the variables 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 data must satisfy. It should be intuitively obvious that the more information the scheduler has, the better job it can do in scheduling the transactions. There are, however, sound reasons why it is sometimes advantageous to "keep

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, NR 044-422, and a Miller Fellowship.

efficiency: we would like our scheduler to be reasonably efficient in reaching its decision about each arriving request, and excessive information may be distracting. These issues are examined in [Papadimitriou 78]. Another reason is that some information may not be available to the scheduler. For example, the integrity constraints may only be implicit. If the semantics of the operations are given in some powerful enough logical language, then the scheduler may even be faced with undecidable problems. Finally, it may be appropriate to leave the scheduler in some imperfect level of information because of other considerations, such as recovery [Gray 78].

There is a growing body of literature on various solutions to the concurrency control problem. This paper gives a uniform framework for evaluating these solutions, and, in many cases, for establishing their optimality. We point out a trade-off between the performance of a scheduler and the information that it uses. We show that most of the existing work on concurrency control is concerned with specific points of this fundamental trade-off. For example, our framework allows us to formally show that the popular approach of serialization (see, e.g., [Bernstein et al. 78], [Eswaran et al. 76], [Papadimitriou et al. 77], [Papadimitriou 78], [Stearns et al. 76], [Silberschatz and Kedem 78]) is the best one can hope for when only syntactic information is available. If the scheduler also has some semantic information, then non-serializable approaches such as those proposed by [Kung and Lehman 79] and [Lamport 76] are possible.

In Section 2 we introduce our model of transaction systems, carefully distinguishing among the syntactic, semantic, and integrity constraint components. In Section 3 we define schedulers, and develop the basic tools for studying the information-performance trade-off. In Section 4 we show several examples of schedulers, most of them already existing in the literature, that can be proven optimal with respect to the information that they use.

In Section 5 we examine the concept of locking from a similar viewpoint. We show that the locking approach amounts to first transforming a transaction system by a locking policy, and then entrusting its concurrency control to a very simplistic scheduler, the lock manager. We examine the question of optimality of the two-phase locking policy of [Eswaran et al. 76], and we outline a geometric methodology that is very useful for understanding locking. A full account of our investigations in locking appears in a forthcoming paper [Kung and Papadimitriou 79]. Finally, in Section 6 we discuss our results, the limitations of our model, as well as directions of future work.

2. Transaction Systems: Definition

By a transaction system we mean intuitively a database (basically data and integrity constraints) together with a set of prespecified transaction programs. Through these fixed transaction programs multiple users can access the database from different terminals in an interactive way. In the following we give both syntactic and semantic definitions of a transaction system. The definitions will be illustrated by an example in the end of the section.

Syntax

A transaction system T is a finite set of transactions, $\{T_1, ..., T_n\}$, 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 transformed, based on knowledge available to the transaction T_i at this time, namely, the values of all "declared" local variables $t_{i1}, ..., t_{ij}$. In keeping this transformation as general as possible, we do not assign specific meaning to f_{ij} at this point; f_{ij} may be open to arbitrary interpretations. For example, it could be the identity function on t_{ij}, in which case T_{ij} is simply a read step. Similarly, if f_{ij} is independent of t_{ij}, then T_{ij} is a write step. In this case, $t_{ij} \leftarrow x_{ij}$ need not be performed in an actual implementation.

Thus, our transactions are straight-line programs. In this simplified model of computation, results of this paper can be made easy to understand. In Section 6, we shall discuss how the results can be extended to transactions defined by more general programs.

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 set {0,1}, 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 $\Pi_{1\le i\le n}(\Pi_{1\le j\le j_i} D(x_{ij}))$ 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 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.

Finally, the semantics of T is associated with the function symbol f_{ij} at each step T_{ij} is a function $\varphi_{ij} : \Pi_{1\le k\le j} D(x_{ik}) \to D(x_{ij})$ which is the interpretation of f_{ij}. 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 $j_i \le m_i$ and $j_i = j$, then its execution modifies the three components of the state as follows:

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

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

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

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:

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

φ_{21} = if $t_{21} \ge 50$ then $t_{21} - 50$ else t_{21}
φ_{22} = if $t_{21} \ge 50$ then $t_{22} + 1$ 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 $A \ge 0$, $B \ge 0$, and $A + B = S - 50C$.

3. An Information-Based Model for Schedulers

3.1. Schedules

A schedule (a log or a history) of a transaction system T is a permutation π of the set of steps in T such that $\pi(T_{ij}) < \pi(T_{ik})$ for $1 \le j < k \le m_i$. A schedule corresponds to a possible stream of arriving execution requests for steps in T, or the order in which these requests are granted for 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, all serial schedules, i.e., all permutations π such that $\pi(T_{i,j+1}) = \pi(T_{ij}) + 1$ for $j \leq m_i - 1$.

3.2. Schedulers: Performance vs. Information

The primary goal of a scheduler or concurrency control is to transform a log of execution requests into a correct schedule, whose execution will preserve database consistency. Formally, a scheduler for a transaction system T is a mapping S from H to C(T). A scheduler S is said to be correct if all schedules produced by S are correct, i.e., if $S(H) \subseteq C(T)$. In this paper, schedulers under consideration are always assumed to be correct. As mentioned in Section 1, we measure the performance of a scheduler S by its fixpoint set P, which is defined to be the largest subset of H satisfying the following property:

$S(h) = h$ for all $h \in P$.

Hence, P must be a subset of C(T). For sequences of execution requests in P, the scheduler grants the requests in the same order as they arrive. Thus, the larger P is the less chance that the scheduler will have to ask a user to wait for other users. Further justifications of this measure will be given in Section 6.

While considering the performance of a scheduler, we must also look at its cost. A high performance scheduler that has a large cost is not necessarily useful. The cost of a scheduler refers to either the information or the time that the scheduler requires to make its decision. This paper studies the information component of the cost of schedulers. We derive upper bounds on the performance of schedulers based solely on the information they use, and we do not address the problem of how long it takes for schedulers to reach their decisions. The latter problem has been examined in great detail in [Papadimitriou 78], where sufficient and necessary conditions for the existence of efficient schedulers with prescribed fixpoint sets are given.

Given that the fixpoint set of any scheduler must always be a subset of C(T), ideally we wish to have a scheduler that can recognize all correct schedules in C(T) so as to maximize performance. For several reasons that we mentioned in Section 1, however, this is not always possible, nor desirable. The maximum possible information that a scheduler can have is, of course, the complete syntactic and semantic information about the transaction system in question. The minimum information is the format $(m_1, \ldots, m_n)$. The more information available to the scheduler, the "better" scheduling results may be expected. We would like to capture this in a formal theorem (Theorem 1 below). What is, therefore, a formal model for the information available to a scheduler S?

3.3. A Formal Theory

A level of information available to a scheduler about a transaction system T is a set I of transaction systems $\{T, T', T'', \ldots\}$ that contains T. Intuitively, if S is kept at this level of information, it knows that the transaction system it handles is among the transaction systems in I, but does not know exactly which. 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 define I as a projection that maps any transaction system T to an object I(T). Intuitively, I(T) is the information extracted from T by the projection operator I; for example, I(T) could be the syntax of T for all 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, $I = \{T' : I(T') = I(T)\}$.

Theorem 1: For any scheduler using information I, its fixpoint set P must satisfy:

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

The proof of this theorem uses a very general adversary argument, instances of which we shall see many times in the sequence. The proof goes as follows: If there is a schedule $h \in P$ and a transaction system $T' \in I$ such that S when fed by h is not correct for T' i.e., $S(h) = h \notin C(T')$, 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 1, the maximum-performance scheduler that is correct 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 [Papadimitriou 78] - 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 included in the level of information I' of S', i.e., if $I \subseteq I'$. On the other hand, schedulers are also partially ordered with respect to their performance: we say that S performs better than S' if $P' \supseteq P$, where P' and P are fixpoint sets of S and S', respectively. Then the mapping from any level of information I to the fixpoint set of the optimal scheduler for I,

$$I \to P \ (= \bigcap_{T' \in I} C(T')),$$

is a natural isomorphism between these two partially ordered sets. This captures the fundamental trade-off between scheduler information and performance: if $I \subseteq I'$ then $P \supseteq P'$ for the optimal schedulers.

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 Extrema 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, $I = \{T\}$. We can therefore define the scheduler S, in principle at least, such that $P = C(T)$. This is the optimal scheduler for the ultimate 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 the serial scheduler S which is defined to be a scheduler satisfying the following property:

$P = \{$all serial schedules in H$\}$ and $S(H) = P$.

By our basic assumption that each transaction is correct, S is correct.

Theorem 2: 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 $\{T_{ik}: x \leftarrow x+1, T_{i,k+1}: x \leftarrow x-1\}$ and T_j is $\{T_{j1}: x \leftarrow 2x\}$. 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 is a contradiction. Hence, for the minimum information case, the only correct schedules that a scheduler can produce are serial schedules, i.e., 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 [Manna 74] 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. By SR(T) we denote the set of all serializable histories of T. A serialization scheduler is defined to be a scheduler S satisfying the following property:

$$P = SR(T) \text{ and } S(H) = P,$$

for any T.

Theorem 3: The serialization scheduler is correct, and is optimal among all schedulers using complete syntactic information.

Proof: To prove that $SR(T') \subseteq C(T')$ for any $T' \in I$, we use Herbrand's Theorem [Manna 74], which essentially states that if two sequences of steps are equivalent under the

Herbrand interpretation, then they are equivalent under any interpretation. Thus if $h \in SR(T')$ then the execution results of h are the same as those of some serial schedule for T'. This implies that for any $h \in SR(T')$, the execution of h preserves the consistency of T'

To prove optimality, take a history $h \notin SR(T)$, 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 sequence S (possibly empty) of steps that is a concatenation of serial executions of transactions such that the initial values $(v_1, ..., v_k)$ are transformed by S 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 history, 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 the scheduler can hope to produce. After all syntactic information is the information one can quite easily extract from 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 [Eswaran et al. 76, Stearns et al. 76, Silberschatz and Kedem 78, Bernstein et al. 78, Papadimitriou 78]. In [Papadimitriou 78], it is shown that for some transaction systems of restricted syntax, although serialization is algorithmically intractable, it can nevertheless be approximated by more restrictive schedulers (see also [Papadimitriou et al. 77]).

4.3. Optimal Schedules for Complete Semantic Information but Integrity Constraints

Consider the transaction system of Fig. 1.

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

Figure 1: A transaction system.

The history $h = (T_{11}, T_{21}, T_{12})$ 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 achievable 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 some concatenation of transaction S, possibly with repetitions and omissions of transactions, also starting from state E. Denote by WSR(T) the set of all weakly serializable schedules of T. It is clear that $SR(T) \subseteq WSR(T)$. The weak serialization scheduler is defined to be a scheduler S satisfying the property:

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

for any T.

Theorem 4: The 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 3, and is omitted.

5. Some Comments on Locking

Almost all concurrency control methods that appear in the literature, with the notable exception of the SDD-1 system ([Bernstein et al. 78]), are implemented by locking, that is, by mechanisms ensuring exclusive access to certain resources, such as data. Locking-based concurrency control mechanisms are certainly special cases of schedulers, and hence our previous formalism applies to them. As we shall see, they are in fact very restricted special cases of schedulers, and possess an interesting mathematical structure of their own that is susceptible to a theoretical study parallel to the one developed in the previous sections. A full account of our results on locking will appear elsewhere [Kung and Papadimitriou 79]. We shall allude here to only the main important ideas. As a result, this section is quite dense.

5.1. Locking Policies

A locking-based concurrency control mechanism is implemented via a locking policy. A locking policy, L, takes an ordinary transaction system T, as defined in Section 2, and maps it into another transaction system, L(T), called the locked transaction system. Locked transaction systems have the following characteristics:

- Besides the set of variable names V of T, L(T) has also a set of new variable names LV, the locking variables. If $X \in LV$, then the domain of X, D(X), contains only three elements: 0 (for unlocked), 1 (for locked) and -1 (for error). In usual implementations, there is an isomorphism between LV and V, and a locking variable $X \in LV$ can always be thought of as the lock-bit of some ordinary variable $x \in V$. There is no reason, however, to impose this restriction to LV at this point.

- The steps of L(T) are the same as the steps of T, except that there are some additional steps of the form "lock X", "unlock X" inserted. These steps are well-nested in the obvious sense. They have a fixed interpretation: lock X means X: = if X = 0 then 1 else -1; unlock X means X: = if X = 1 then 0 else -1.

- The integrity constraints of L(T) correspond just to the assertion that $\Lambda_{X \in LV}(X = 0)$. In other words, all one has to do in order to safeguard the execution of L(T) is to manage locks properly.

Thus all the cleverness of concurrency control is incorporated into the locking policy L. After a locking policy L is designed, all we have to do is entrust L(T) to a very simple scheduler, the lock respecting scheduler LRS, which can only "see" the locking-unlocking steps, the integrity constraints, and nothing else. Obviously, LRS is optimal with respect to this level of information.

5.2. The Two-Phase Locking Policy - An Example

The most well-known paradigm of locking policies is the two-phase locking policy 2PL [Eswaran et al. 76]. 2PL transforms a transaction system into a locked one as follows:

1. Associate a locking variable X with every $x \in V$. (One can think that X is the lock-bit of x.)
2. If a step T_{ij} accesses x_j, then there is a step "lock X_{ij}" before T_{ij} and a step "unlock X_{ij}" after T_{ij} subject to the following rules:
   a) In no transaction is there a lock step after the first unlock step.
   b) Lock steps are as late and unlock steps as early as possible subject to condition a) above. Note that this does not uniquelly define the positions of locks, but we shall disregard this point.

For example, 2PL transforms the transaction of Figure 2(a) to that of Figure 2(b).

Original Transaction	Locked Transaction
T_{11}: x ← ...	lock X
	T_{11}: x ← ...
T_{12}: y ← ...	lock Y
	T_{12}: y ← ...
T_{13}: x ← ...	T_{13}: x ← ...
T_{14}: z ← ...	lock Z
	unlock X
	unlock Y
	T_{14}: z ← ...
	unlock Z
(a)	(b)

Figure 2: Locked transaction using 2PL.

Notice that one can talk about the information used by a locking policy exactly as with schedulers (Section 3). For example, 2PL uses only syntactic information. We shall return to discuss the question of its optimality. What is a performance measure for a locking policy L? Following our approach for general schedulers, we consider the set of schedules that are possible outputs of LRS to schedules of L(T). To compare with ordinary schedulers for T, we simply remove the lock-unlock steps from these schedules.

5.3. The Geometry of Locking

Much insight into locking can be gained by a simple geometric method. Suppose that we have two transactions T_1 and T_2. Then any state of progress towards the completion of T_1 and T_2 can be viewed as a point in the two-dimensional "progress space", as shown in Figure 3.

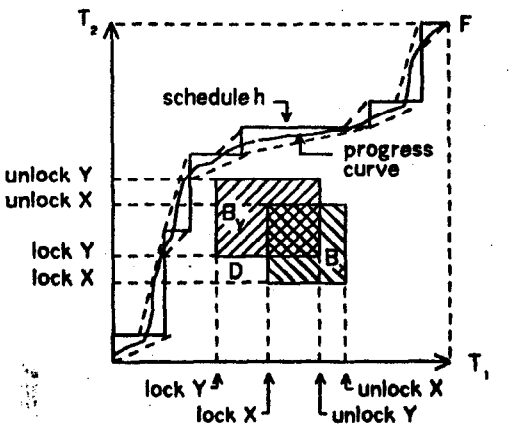

Figure 3: The "progress space" for T_1 and T_2.

Locking has the effect of imposing restrictions in the form of forbidden rectangular regions (blocks B_x and B_y Figure 3). The joint progress of T_1 and T_2 is represented by a nondecreasing curve from the origin to the point F that avoids all blocks. Such a curve, called a progress curve, is shown is Figure 3. The simultaneous increasing of the progress curve in two coordinates corresponds to the simultaneous progress the users make at their terminals. A schedule produced by a scheduler, however, corresponds to a nondecreasing step function, reflecting the fact that the scheduler grants only one request at a time. The step functions h in Figure 3 represents the schedule that could result in the particular progress curve shown in the figure. In fact, any nondecreasing function lying entirely in the indicated triangular regions surrounding the step function h can be a progress curve resulting from the schedule h. Region D in Figure 3 is a deadlock region, in the sense that any progress curve trapped in the region will not be able to reach F. In fact, this geometric method was used for the study of deadlocks by Dijkstra [Coffman et al. 71]. Here, we use it in a quite different way for studying several consistency related problems.

First, how good is locking as a concurrency control primitive? In other words, how general are the schedulers that can be implemented by locks? The answer is, not very. Note that any lock-implemented scheduler is memoryless in the following sense. Consider Figure 4(a). When the execution has reached point g, it has essentially "forgotten how it got there". We cannot distinguish among histories leading to the same point just by locking. Thus, if a class of schedules is the output set of a locking policy, it must be oblivious in this sense. Unfortunately, most sophisticated serialization principles (see, e.g., [Papadimitriou 78]) require that the scheduler remembers which transaction read data first from which, and thus they cannot be implemented by locks alone - although they may be implementable by queues ([Bernstein et al. 78]). In fact, the above statement has a converse that characterizes classes of schedules that can be the output sets of locking policies. In contrast, recall that, at least in principle, all classes of schedules are possible output sets of some scheduler.

Secondly, let us consider consistency - in fact, serializability, by assuming only syntactic information. Assume that the locking variables are locking bits, and that the transactions are well-formed, in that any access of x is surrounded by a (lock X, unlock X) pair. Then it can be shown that a schedule h is serializable if it can be transformed by elementary transformations (see Figure 4(b)) to one of the serial schedules without passing through any of the forbidden blocks. (The two serial schedules are the two nondecreasing functions lying on the boundaries of the square, OP_1F and OP_2F.) Such a elementary transformation corresponds to "interchanges" of the neighboring steps such as T_{1l} and T_{2r}. In the classic mathematical terminology, a serializable schedule is homotopic to some serial schedule. So non-serializable schedules are schedules that separate blocks (Figure 4(c)). An incorrect locking policy means a policy that may leave the blocks disconnected. The exact condition for a correct locking policy is somewhat less trivial for high dimensional cases, which correspond to transaction systems consisting of more than two transactions. The two-phase locking is now extremely easy to explain. It simply keeps all blocks connected by letting them have a point u in common. (Figure 4(d)). The coordinates u_1, u_2 of u are the phase-shift points, at which all locks have been granted, and none has been released. It is easy to check that u is contained by all blocks. This implies that 2PL is correct.

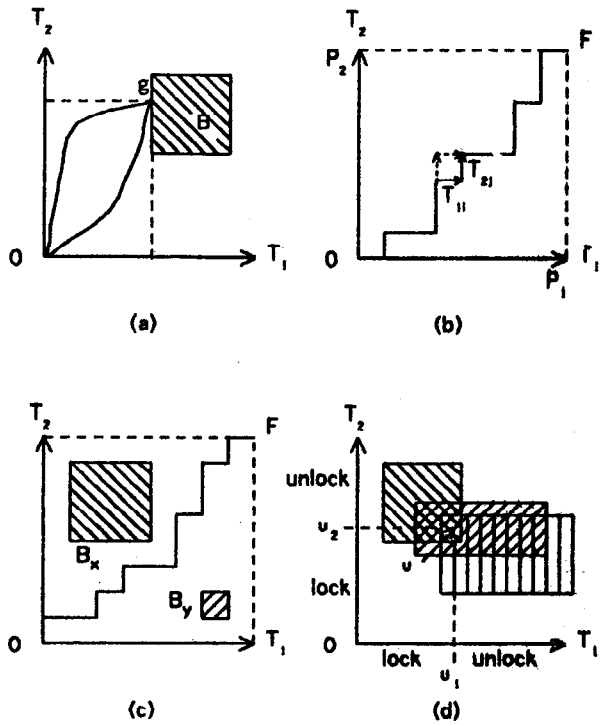

Figure 4: The geometries of locking.

5.4. Is Two-Phase Locking Optimal?

By the previous discussion 2PL cannot be optimal as a scheduler, since there will always be a scheduler that performs strictly better than any locking policy. But is 2PL optimal as a locking policy? The answer is no for a trivial reason. Suppose that there is a variable x that is

only accessed by just one transaction. Then a locking policy that two-phase locks all variables but x may be strictly better than 2PL, and still it is correct. This counter example shows just one of the ways that one can take advantage of global knowledge of all the transactions. However, 2PL has an important property, which is also a significant practical advantage: it is separable, in that it transforms the transaction system one transaction at a time, without using information on other transactions.

Is, therefore, 2PL at least optimal among separable locking policies? The following variant of 2PL can be shown to be both correct and strictly better than 2PL in performance.

2PL′:

1. Apply 2PL to all variables except to a distinguished one, x.
2. After the first usage of x insert a pair of steps lock X′ - unlock X′.
3. After the last usage of x insert the steps lock X′, unlock X.
4. After the last lock step insert unlock X′.

For example, 2PL′ would transform the transaction of Figure 2(a) into the one of Figure 5 (b). 2PL′ is correct, separable, and better than 2PL in performance, but is not the two-phase locking policy.

2PL, however, is optimal in the following important sense.[1] It is the best among all separable locking policies with syntactic information on unstructured variables. In other words, it is optimal among all policies that remain correct under arbitrary, local to the transactions, renamings of the variables. The tree-locking scheme of [Silberschatz and Kedem 78] violates this by assuming a hierarchical database, and our 2PL′ by making the variable x distinguished.

5.5. Conclusions about Locking

Locking is a simple primitive for implementing concurrency control techniques. Unfortunately, its simplicity is payed for by a significant loss in performance. A simple geometric vehicle is especially helpful for

[1] This remark is due to Mihalis Yannakakis [Yannakakis 79].

Original Transaction	Locked Transaction
T_{11}: x ← ...	lock X
T_{12}: y ← ...	T_{11}: x ← ...
T_{13}: x ← ...	lock X′
T_{14}: z ← ...	unlock X′
	lock Y
	T_{12}: y ← ...
	T_{13}: x ← ...
	lock X′
	unlock X
	lock Z
	unlock Y
	unlock X′
	T_{14}: z ← ...
	unlock Z
(a)	(b)

Figure 5: Locked transaction using 2PL'.

studying locking and its limitations. Strictly better results should be expected by combining locks with other simple techniques, such as queues [Bernstein et al. 78]. Restricting ourselves to locking, 2PL is optimal only for unstructured data. More general locking policies can therefore be devised by taking advantage of structured data [Kung and Papadimitriou 79, Yannakakis 79].

6. Discussions

A typical environment to which results of this paper apply can be described as follows: There are multiple users at various terminals executing transactions which mainly involve local computations but occasionally have to access or update data shared by many users. This is the case for example when in each transaction step the computation of $f_{ij}(t_{i1}, ..., t_{ij})$ is much more time-consuming than the read and write on x_{ij} (cf. Section 2). To safeguard the consistency of the database, some centralized scheduler is employed to properly sequence the execution of transaction steps from different users. From a user's viewpoint the time for carrying out a transaction step is divided into the following three parts:

- Scheduling time: The execution of the transaction step has to be scheduled by the scheduler. This may involve the time spent in waiting for the scheduler to become available to do its job and the time for the scheduler to figure out its decision.
- Waiting time: The scheduler may decide that the transaction step can not be executed until the completion of some transaction steps from other users.

- Execution time: This is the time actually spent in executing the transaction step.

we are interested in choosing a scheduler that will minimize the sum of these three quantities. We assume that the execution time is a constant, since it is independent of the the scheduler. The waiting time is directly related tc the fixpoint set P of the scheduler for the following reasons:

- The probability that none of the transaction steps have to wait is $|P|/|H|$, if all request histories are assumed to be equally likely.
- The richer P is the easier (and hence less waiting required) to rearrange a history originally not in P into one in P.

Thus, in the paper we have used P to measure the performance of the scheduler. The scheduling time reflects the complexity of the scheduler. Scheduling times for different users can not be overlapped, since there is only one central scheduler for all users. Thus, the scheduling time of a transaction step is also affected by the number of users who are competing for the scheduler. In general it it a difficult task to characterize the complexity of a scheduler. This paper has addressed it only in the information-theoretic point of view. Results of this paper nevertheless can have practical significance as well, if the schedulers in question have relatively small scheduling times as compared with waiting and execution times. This is fortunately often the case in practice, since practical schedulers all tend to be simple.

Our assumption that all transactions are straight-line programs is not essential, and was made only because it tends to simplify somewhat the notation. It also simplifies concepts like that of a legal schedule, which would have been data-dependent otherwise. We can easily extend our results in this direction.

A more important issue is the assumption that underlies our model that all information available to the scheduler is known to it at the beginning of the session with the transactions. This includes our other assumption that all transactions are fixed beforehand. In practice, however, one expects the scheduler to acquire this knowledge progressively and interactively, by questioning the users and soliciting declarations. This issue of dynamic information (as opposed to our static model) is admittedly a very important one, and must be dealt with theoretically in future work in concurrency control. Our results of Section 4 are in effect negative results, showing the impossibility of the existence of schedulers better than given ones, so their validity does not depend on this static information assumption. What remains to be seen, however, is whether our static information model prevents us from proving similar optimality results for certain other levels of information. We shall next see that this is indeed the case.

We have not examined in any detail so far schedulers operating at a level of information that includes the integrity constraints. Examples of such schedulers do exist. One example is the concurrency control of binary search programs proposed by [Kung and Lehman 79]. Their programs allow constructs of the form "if no other program has modified x since the beginning of the present program then $x \leftarrow a$ else $x \leftarrow b$". It is not hard to argue that this construct is inherently non-serializable. This construct, however, can be used safely if it is known that the integrity constraints do not involve x at all.

A different way to use the integrity constraints (and some further semantic information as well) is through proofs of correctness. Correctness proofs must rely on and, more importantly, must also reflect the meanings of the transation and integrity constraints. Therefore, a natural way to capture semantic information is to examine proofs. Such an approach has been proposed by L. Lamport [Lamport 76]. We outline it in the following. Consider proofs using assertions [Floyd 67]. A transaction is represented as a flowchart of operations which manipulate the global variables. Executing the transaction is viewed as moving a token on the flowchart from the input arc to an output arc. An assertion, defined in terms of the variables, is attached to each arc of the flowchart; in particular, the assertions on the input and any output arcs are the integrity constraints. A correct proof of a serial transaction amounts to demonstrating that throughout the execution of the transaction the token will always be on an arc whose assertion is true at that time, and will eventually reach an output arc. The consistency of a database under the concurrent execution of several correct serial transactions can be insured by the following scheduling policy:

> The request to execute one step in a transaction is granted only if the execution will not invalidate any of the assertions attached to those arcs where the tokens of other transactions reside at that time.

It is possible that at some time none of the transactions can be granted to execute their next steps. The "deadlock" situation can be resolved, for example, by backing up some transactions. With this approach it is possible for a scheduler to generate correct schedules beyond serial, serializable, or weakly serializable schedulers. Using the methodology developed in this paper, we can establish the optimality of the above scheduler in a dymanic information model. We plan to pursue this in a later version of this paper.

References

[Bernstein et al. 78]
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.

[Coffman et al. 71] Coffman, E.G., Jr., Elphick, M.J. and Shoshani, A.
System Deadlocks.
Computing Surveys 3(2):67-78, June 1971.

[Eswaran et al. 76] 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.

[Floyd 67] Floyd, R.W.
Assigning Meanings to Programs.
In *Proc. Symposium in Applied* Mathematics, pages 19-32. American Mathematics Society, 1967.

[Gray 78] Gray, J.
Notes on Data Base Operating Systems.
Technical Report RJ2188, IBM Research Laboratory, San Jose, February 1978.

[Kung and Lehman 79]
Kung, H.T. and Lehman, P.L.
A Concurrent Database Problem: Binary Search Trees.
Technical Report, Carnegie-Mellon University, Department of Computer Science, May 1979.
An abstract appears in the *Proceedings of the Fourth International Conference on Very Large Databases.*
The full paper is to be published in *ACM Transactions on Database* Systems.

[Kung and Papadimitriou 79]
Kung, H. T. and Papadimitriou, C. H.

In Preparation.

[Lamport 76] Lamport, L.
Towards a Theory of Correctness for Multi-user Data Base Systems.
Technical Report CA-7610-0712, Massachusetts Computer Associates, Inc., October 1976.

[Manna 74] Manna, Z.
Mathematical Theory of Computation.
McGraw-Hill, New York, 1974.

[Papadimitriou 78] Papadimitriou, C.H.
Serializability of Concurrent Updates.
Harvard University.
To appear in JACM.

[Papadimitriou et al. 77]
Papadimitriou, C.H., Bernstein, P.A. and Rothnie, J.B.
Computational Problems Related to Database Concurrency Control.

In *Proc. Conf. on Theoretical Computer* Science, pages 275-282. University of Waterloo, 1977.

[Silberschatz and Kedem 78]
Silberschatz, A. and Kedem, Z.
Consistency in Hierarchical Database Systems.
Manuscript, University of Texas at Dallas.

[Stearns et al. 76] Stearns, R.E., Lewis , P.M. II and Rosenkrantz, D.J.
Concurrency Control for Database Systems.
In *Proceedings of the Seventh Annual Symposium on Foundations of Computer Science*, pages 19-32. , 1976.

[Yannakakis 79] Yannakakis, M.

Private Communication.