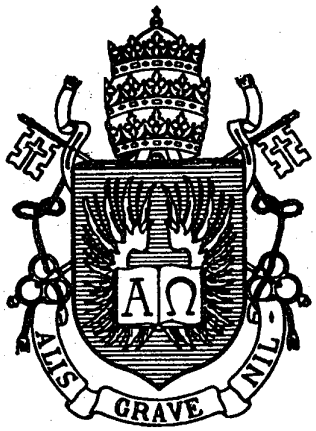

PUC

Series: Monografias em Ciência da Computação

Nº 2/81

ON THE CONSTRUCTION OF DATABASE SCHEDULERS BASED ON CONFLICT-PRESERVING SERIALIZABILITY

by

Marco A. Casanova

Philip A. Bernstein

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP-22453

RIO DE JANEIRO - BRASIL

Series: Monografias em Ciência da Computação

Nº 2/81

Series Editor: Marco A. Casanova Janeiro - 1981

ON THE CONSTRUCTION OF DATABASE SCHEDULERS BASED ON CONFLICT-PRESERVING SERIALIZABILITY*

by

Marco A. Casanova
Philip A. Bernstein**

* Research supported by the Brazilian Government Agencies CNPq and FINEP and the U.S. National Science Foundation.

** Center for Research in Computing Technology, Harvard University.

ABSTRACT:

This paper discusses solutions to the concurrency control problem for databases systems using a method called conflict-preserving serializability, an approach significantly different from conventional locking methods. The major results of the paper spell out how to implement such a method within practical space bounds. They are applied to the construction of schedulers for centralized, as well as distributed data-base systems.

Key Words: concurrency control, serializability, conflict graphs

RESUMO:

Esta monografia discute soluções para o problema de controle de concorrência em sistemas de banco de dados usando um método chamado de serialização por preservação de conflitos, que difere consideravelmente dos métodos de bloqueio convencionais. Os resultados principais desta monografia indicam como implementar tal método usando uma quantidade prática de memória. Estes resultados são então aplicados a construção de algoritmos para controle de concorrência em sistemas centralizados ou distribuídos.

Palavras-chave: controle de concorrência, serialização, grafos de conflito.

1. INTRODUCTION

If several users concurrently access a database, synchronization anomalies may arise leading, for example, to the loss of database consistency or to the loss of updates (see Figure 1). Thus, concurrency control subsystems, or *schedulers*, must be interposed between users and the database to arbitrate access to data. This paper addresses the question of constructing schedulers using a much less restrictive strategy than currently available schedulers, thereby possibly increasing concurrency. The major results of the paper spell out how to implement such a strategy within practical space bounds, a problem that so far has remained unsolved. The scheduling strategy guarantees serializability [BE1], a widely accepted correctness criterion avoiding both synchronization anomalies previously mentioned.

The paper is organized as follows. Section 2 introduces an informal model of schedulers that capture the major characteristics of a large class of real schedulers. Section 3 describes the scheduling strategy adopted and presents the major results of the paper. Finally, Sections 4 and 5 discuss implementations in a centralized and in a distributed environment, respectively.

Schedulers for database systems have been extensively studied before (e.g. [BE1, BE2, ES, GA, KU, PA1, PA2, RO, SC, ST, TH]). Our model of schedulers differs from previous work (e.g. [PA2]) insofar as we are concerned with the dynamic acquisition of information about the user's programs that characterize systems like IMS [IB], where user's programs become known to the system as they are submitted. Our scheduling strategy is derived from [BE1, PA1, PA2]. Although its potential has been realized

FIGURE 1.1

(a) Let U1 and U2 be two users of a database that decrease x by 1 by executing the following program: $a_i := x$; $a_i := a_i - 1$; $x := a_i$. Assume that the two users run concurrently as follows:

USER \ TIME	1	2	3	4	5
U1	$a_1 := x$		$a_1 := a_1 - 1$		$x := a_1$
U2		$a_2 := x$	$a_2 := a_2 - 1$	$x := a_2$	

Then, the update performed by U2 is overwritten by U1 and, hence, is lost.

(b) Consider that the database is consistent iff $x \geq 0$. Let U1 and U2 be two users that decrease x by 1 by executing the following program: if $x > 0$ then $x := x-1$. Assume that the two users run concurrently, starting with $x = 1$, as follows:

USER \ TIME	1	2	3	4
U1	$x > 0$?		$x := x-1$	
U2		$x > 0$?		$x := x-1$

Then, $x = -1$ in the final state and, hence, consistency is lost. □

before, previous suggestions [PA1] required keeping information about all user's programs processed thus far and, hence, were not practical. The results in Section 3 lead to a new approach that summarizes information about terminated transactions within practical space bounds, and constitute the major contribution of the paper. It can be shown [BE2] that several currently available schedulers, notably two-phase locking [ES], follow strategies that are, in a precise sense, more restrictive than ours (see also [ST, TH]).

We leave untouched two important aspects, though. First, a theoretical and empirical framework for comparing the effect of different schedulers on transactions' response time is badly needed to guide the choice of a scheduling strategy vis-à-vis a particular application. Second, more detailed implementation descriptions using the proposed scheduling strategy must be worked out before the strategy's potential can be fully evaluated.

2. MODEL OF A CENTRALIZED DATABASE SYSTEM

This section sets the context for the results in Sections 3 and 4. It introduces the notions of database schema, transaction, database system and general purpose scheduler and describes strategies for constructing schedulers. To avoid a heavy conceptual burden, we proceed as informally as clarity permits, referring the reader to [CA1, CA2] for a formal account of these notions. We concentrate exclusively on centralized database systems, leaving for Section 5 considerations about distributed systems.

2.1 Basic Notions

A database schema is a pair DB = (V,A), where V is a set of *database variables* taken from a *universe of variables* U, and A is a set of *consistency criteria*. A *database state* is an assignment of values to the variables in V from a certain *domain of variables* D. A state is *consistent* iff all consistency criteria are satisfied. The language used to write schemas and the method adopted to decide satisfiability need not concern us here.

A program accesses a database by executing *read* or *write* operations, that retrieve or modify, respectively, the value of a set of database variables (the operation's *readset* or *writeset*). We assume that read and write operations are uniquely numbered within each program so that we can refer to the program's i-th read/write operation. For purposes we explain later, we force each program to signal when it starts/terminates by executing a *begin/end* operation. Finally, we assume that each program preserves consistency of the database, that is, each program is a *transaction*, and that it terminates when executed by itself.

A *centralized database system* DBS has four components, as summarized in Figure 2.1: a database DB, a set of transactions $T = \{T_1, \ldots, T_n\}$, a *scheduler* SC and a *data manager* DM. Each operation of T_i actually sends a message to SC containing:

- begin or end operation: just the transaction number;
- read operation: transaction number and the number and readset of the operation;
- write operation: same as a read operation, plus new data values.

SC may delay the message, but eventually passes it to DM. DM accesses the database on behalf of the transactions, replying to a read message with the data values requested and to all other messages with just an ACK (acknowledgement). We assume that SC only outputs messages if received and that all messages of each transaction T_i are output in the same order as received. Moreover, we assume that messages sent by SC to DM are *pipelined* (i.e., received by DM in the same order in which SC sends them).

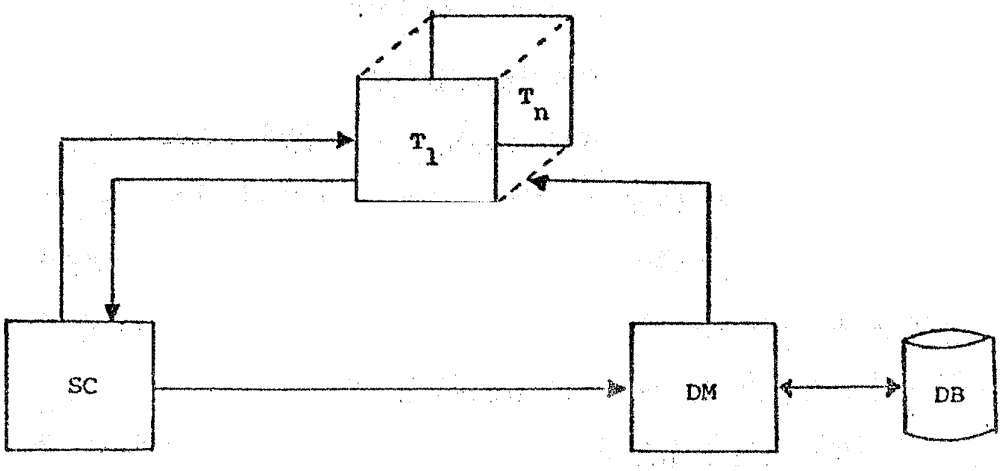

FIGURE 2.1

We assume a limited restart facility operating as follows. The scheduler may restart a transaction T_i by sending to T_i a *restart message*. Before sending any message to the scheduler, T_i checks if it was restarted and, if so, resumes execution at the beginning. However, the scheduler can only restart T_i after it outputs the begin message of T_i and before it outputs any write message of T_i. Hence, restarts do not cascade and no database rollback is necessary, since T_i will never be restarted after writing on the database. We show in Section 4 that, under certain assumptions about transactions, strongly motivated by reliability considerations, this limited restart facility suffices. We do not state these assumptions here, however, because the results in Section 3 are insensitive to them.

It is clear from the previous discussion that the scheduler controls the concurrent execution of the transactions by relaying the messages it receives to the data manager in a different order. Thus, to discuss concurrency control strategies, it is convenient to introduce a special notation for streams of messages. We then define a *log* as a pair $h = (L,S)$, where L is a string of symbols from the alphabet $\Sigma = \{R_{ij}, W_{ij}, B_i, E_i / i,j \in \mathbb{N}\}$ and S is a function assigning to each symbol R_{ij} or W_{ij} occurring in L a subset of the universe of variables U. We use $\underline{elem}(L)$ and $\underline{last}(L)$ to denote the set of symbols and the last symbol occurring in L, respectively. If $x \in \Sigma$ precedes $y \in \Sigma$ in L, we write $x <_L y$. We call a *log predicate* any restriction on logs.

We interpret a log $h = (L,S)$ as follows. The symbols B_i, E_i, R_{ij} and W_{ik} correspond to the messages generated by the begin, end, the j-th read and k-th write operations of T_i; L records the order of the messages

in the stream; $S(R_{ij})$ gives the readset of the j-th read operation of T_i, and similarly for $S(W_{ik})$.

An *output log* of a database system DBS is defined as the log $h = (L,S)$ corresponding to the stream of messages sent by the scheduler during a computation of DBS. To take into account restarts, we define the *reduced output log* corresponding to $h = (L,S)$ to be the log $h' = (L',S')$ where L' is obtained from L by deleting all symbols corresponding to operations performed by restarted executions of a transaction, and S' is the restriction of S to L'. Since the messages sent by the scheduler to the data manager are pipelined, the reduced output log records the order in which operations performed by the final execution of each transaction were applied to the database.

In view of the above discussion and since, by assumption, the scheduler outputs messages of T_i in the same order as received, for any transaction T_i, we may restrict ourselves to logs satisfying the following log predicate:

$h = (L,S)$ is a *well-formed log* ($h \in$ WLOG) iff

(1) each symbol in Σ occurs at most once in L;

(2) each R_{ij} or W_{ik} occurring in L must succeed B_i in L;

(3) if E_i occurs in L, then E_i must succeed B_i in L and all symbols R_{ij} and W_{ik} occurring in L.

Ultimately, we want to design database systems free of synchronization anomalies. The first step is to define a *system correctness criterion* Q restricting the behavior of database systems so that anomalies do not arise. Given two criteria Q and R, we say that Q *implies* R iff any database

system satisfying Q also satisfies R. A log predicate P does not qualify directly as a system correctness criterion, but as an abuse of language we classify P as one in the sense that a database system satisfies P iff all restricted output logs of the system satisfy P.

We will be primarily interested in the following correctness criterion. We say that a database system DBS is *serializable* iff any computation of DBS produces the same changes on the database variables as some serial execution of the transactions of DBS, one after the other. Hence, if DBS is serializable, we can transfer properties of serial executions of the transactions to DBS. For example, since each transaction preserves consistency, so does any serial execution of the transactions and, hence, DBS. This was the original argument in favor of serializability. But equally important, since in a serial execution of the transactions no update is lost, DBS also enjoys this property. In fact, if we consider the output of each transaction as a database variable, we can go further and assert that serializability guarantees that each transaction executes without interference from the others. Thus, serializability is central to concurrency control because it precludes the basic synchronization anomalies.

2.2 General Purpose Schedulers

The design of a database system satisfying a correctness criterion Q depends on Q itself, the scheduler and the set of transactions, assuming a data manager operating as intended. Although the dependency on Q is intrinsic, we can factor out the set of transactions (representing the application in question) by considering general purpose schedulers. A scheduler SC is a *general purpose scheduler* for Q iff:

- any database system whose scheduler is SC satisfies Q. Hence, SC is not application-dependent;
- the response of SC at a given point in time depends only on the readsets and writesets contained in the messages received, and on the stream of messages output thus far. Thus, the response of SC does not depend on the meaning of the transactions, the consistency criteria of the database, and on the operations not yet executed.

Even without fixing the correctness criterion Q, there are certain general remarks about the construction of a general purpose scheduler SC for Q worth stating. They will also help put into proper perspective the results of Sections 3 and 4. We first observe that Q offers little help in the construction of SC, since Q, in general, imposes restrictions on the database systems using SC. However, this is not the case for log predicates, because they restrict directly the scheduler output. Therefore, we propose the following problem-reduction strategy: first find a log predicate P implying Q, then construct a general purpose scheduler SC' for P'. It is trivial to see that SC' will also be a general purpose scheduler for Q, but as pointed out before, it is much simpler to construct SC' than SC.

Once an appropriate log predicate P is found, the next step is to choose a scheduling strategy. We have found it useful to distinguish between *conservative* and *aggressive* schedulers. A conservative scheduler tests if each message received can be added to the current output log without violating P; if not, the message is delayed, otherwise it is output. If the scheduler does not know in advance which operations a transaction will perform, scheduling mistakes can be made, which are corrected by restarting operations. On the other hand, if the readsets and writesets of

each operation are known when the transaction starts, mistakes and, thus, restarts can be avoided [CA1]. An aggressive scheduler tests if the output log satisfies P only when a transaction T_i requests to terminate by executing an *end* operation. If the current output log does not satisfy P, operations must be restarted to correct scheduling mistakes. Note that the scheduler cannot allow T_i to terminate without testing the output log, since the mistakes may only be correctable by restarting operations of T_i.

Both strategies, as outlined, only guarantee that the output log satisfies the log predicate, but not that all transactions terminate. In a conservative scheduler an operation may be delayed indefinitely or restarted repeatedly, while only the last problem occurs in an aggressive scheduler. Thus, rescheduling operations must be carefully planned if all transactions must terminate.

Conservative schedulers should be used when the probability of creating logs not satisfying the log predicate is high, since they tend to make fewer scheduling mistakes and, hence, require fewer restarts. Symmetrically, aggressive schedulers should be used when such probability is low, since they do not delay operations.

To conclude, we suggest that the construction of a general purpose scheduler should start with the choice of an appropriate log predicate restricting the possible output logs. This step depends only on the kind of synchronization anomalies that must be avoided. Next, a scheduling strategy should be fixed, taking into account the expected transaction population. Sections 3 and 4 explore these remarks in detail.

3. CONFLICT-PRESERVING SERIALIZABILITY

We now concentrate on the construction of general purpose schedulers for serializability, following the remarks in Section 2.2. This section introduces a log predicate, called *conflict-preserving serializability* (CPSR) that implies serializability, and explains how to test if the reduced output log satisfies it. Section 3.1 defines CPSR and relates it to serializability. Section 3.2 describes a straightforward method of testing for CPSR that unfortunately requires too much memory. This objection is removed in Section 3.3 via a refined method, which constitutes the central result of the paper. Section 4 uses the refined method to construct an aggressive scheduler, illustrating its usefulness.

3.1 Basic Results

Serializability is universally accepted as the appropriate correctness criterion for eliminating synchronization anomalies, but the methods used to construct what we called general purpose schedulers for serializability vary considerably. A general consensus is regained if we observe the scheduler output only, because it can be shown [BE1] that several of these methods guarantee that the restricted output log satisfies the two-phase locking condition of [ES], viewed here as a log predicate. We depart from this consensus by exploring a different log predicate, first described in [PA1], called *conflict-preserving serializability* (CPSR).

Our reason for choosing CPSR is two-fold. First, there is a fast method for testing if a log is CPSR, which is certainly necessary for construction of efficient schedulers. Second, any log satisfying the

two-phase locking condition also satisfies CPSR, but the converse is not true. Hence, CPSR allows more freedom for a scheduler to choose its output, thereby possibly increasing concurrency.

CPSR is defined as follows.

DEFINITION 3.1: Let $h = (L,S) \in WLOG$.

h is *conflict-preserving serializable* ($h \in CPSR$) iff

$PRECEDES[h] \subset \mathbb{N}^2$ is a partial order

where i PRECEDES$[h]j$ iff $i \neq j \wedge$

$$((\exists R_{ik}, W_{j\ell} \in \underline{elem}(L))(S(R_{ik}) \cap S(W_{j\ell}) \neq \emptyset \wedge R_{ik} <_L W_{j\ell}) \vee$$

$$(\exists W_{ik}, W_{j\ell} \in \underline{elem}(L))(S(W_{ik}) \cap S(W_{j\ell}) \neq \emptyset \wedge W_{ik} <_L W_{j\ell}) \vee$$

$$(\exists W_{ik}, R_{j\ell} \in \underline{elem}(L))(S(W_{ik}) \cap S(R_{j\ell}) \neq \emptyset \wedge W_{ik} <_L R_{j\ell}))$$. □

We abbreviate PRECEDES$[h]$ as PR$[h]$. Intuitively, transaction T_i precedes transaction T_j iff T_i reads or writes on a variable T_j writes later on, or T_i writes on a variable T_j reads later on. Thus, if h is in CPSR, we guarantee that there is a partial order among transactions, or, equivalently, that no transaction transitively precedes itself.

EXAMPLE 3.1: Let a log $h = (L_1 \ldots L_k, S)$ be denoted by $L_1[S(L_1)] \ldots L_k[S(L_k)]$. For simplicity, when a transaction T_i issues just one read or write, we represent it by R_i or W_i, respectively. The following logs are in CPSR:

$$h_1 = B_1 B_2 R_1[x] R_2[x] W_2[x] W_1[y] E_2 E_1$$

$$h_2 = B_1 B_3 R_3[x] W_1[x] E_1 B_2 R_2[y] E_2 W_3[y] E_3$$

Readers familiar with [ES] will note that h_2 does not satisfy the two-phase locking condition. The following log is not in CPSR because T_1 precedes T_2 and vice-versa:

$$h_3 = B_1B_2R_1[x]R_2[x]W_2[x]W_1[x]E_2E_1 \quad . \qquad \square$$

CPSR guarantees serializability in the following sense.

THEOREM 3.1: Let DBS be a database system.

If any reduced output log of DBS is in CPSR

and all transactions terminate,

then DBS is serializable.

Proof. (See Appendix.) $\square$

In view of Theorem 3.1, we propose to construct a general purpose scheduler SC for serializability by guaranteeing that any final reduced output log of any database system using SC is in CPSR and that all transactions always terminate. Irrespectively of the scheduling strategy used, we must then provide an efficient *CPSR strategy*, that is, a method for testing if the current reduced output log is in CPSR. Sections 3.2 and 3.3 describe two such strategies. We must also give a *termination strategy* guaranteeing that all transactions eventually terminate. Section 4 discusses one such strategy within the context of aggressive schedulers.

3.2 Basic CPSR Strategy

This section describes a straightforward CPSR strategy, which raises an interesting problem, though, solved in Section 3.3. Let $h = (L,S)$ be a log, to be interpreted as the current reduced output log. From now on, we will freely use graph-theoretic terms when referring to (the graph of) $PR[h]$. A straightforward CPSR strategy consists of checking the acyclicity of $PR[h]$ [PA2]. However, this strategy is not quite practical since it requires keeping the whole log h and computing $PR[h]$ when needed, or keeping $PR[h]$ (or even $PR[h]^+$) and S, which is needed to update $PR[h]$. That is, the scheduler would require time and space which is a function of the total number of transactions. Nonetheless, the following example suggests that S and a large subset of $PR[h]^+$ must indeed be kept (assuming the second alternative).

EXAMPLE 3.2:

(a) Consider the following reduced output log:

$$h = B_1R_1[x]B_2R_2[y_2,x]W_2[z_2,x]E_2B_3R_3[y_3,x]W_3[z_3,x]E_3$$
$$\ldots \quad B_iR_i[y_i,x]W_i[z_i,x]E_i \ \ldots \ B_nR_n[y_n,x]W_n[z_n,x]E_n$$

We first observe that all transactions have terminated, except T_1, and $1PR[h]^+i$, for each i in $[2,n]$.

We now argue that apparently S must be kept. For suppose that the scheduler tries to output W_1 and let h' be the log obtained by adding W_1 to h. If $S(R_i) \cap S(W_1) \neq \emptyset$ or $S(W_i) \cap S(W_1) \neq \emptyset$, then $iPR[h']^+1$. But since h is a prefix of h' and $1PR[h]^+i$, then $1PR[h']^+i$. Therefore, $1PR[h']^+1$ and,

hence, W_1 cannot be output. If $S(R_i)$ or $S(W_i)$ were discarded, this fact would have gone undetected. Likewise, the scheduler has to remember that $1PR[h]^+i$, for each i in $[2,n]$, to decide if W_1 can be output or not. Note that this subset of $PR[h]^+$ takes $O(n)$ space to store, while $PR[h]^+$ requires $O(n^2)$ space.

(b) Consider now a more elaborate version of (a):

$$g = B_1R_1[y_1,x]B_2R_2[y_2,x]W_2[z_2,x]E_2 \cdots$$
$$B_{2i-1}R_{2i-1}[y_{2i-1},x]B_{2i}R_{2i}[y_{2i},x]W_{2i}[z_{2i},x]E_{2i} \cdots$$
$$B_{2n-1}R_{2n-1}[y_{2n-1},x]B_{2n}R_{2n}[y_{2n},x]W_{2n}[z_{2n},x]E_{2n} .$$

Note that all even transactions have terminated, while all odd ones are still running. Moreover, $(2i-1)PR[g]^+j$, for each i in $[1,n]$ and j in $[2i,2n]$. Again, the scheduler has to remember $S(R_{2i-1})$, $S(R_{2i})$, $S(W_{2i})$ and $(2i-1)PR[g]^+j$, for each i in $[1,n]$ and j in $[2i,2n]$. However, unlike (a), the subset of $PR[g]^+$ that must be remembered and $PR[g]^+$ both require $O(n^2)$ space. □

Example 3.2 capitalizes on the fact that $iPR[h']1$ can only be detected after transaction T_i has terminated. We consider this intrinsic dependency on terminated transactions to be the central problem of CPSR scheduling, since it may lead to space and time bounds which are a function of the total number of transactions. While such time bounds are unavoidable in scheduling problems, the space bounds are clearly unacceptable and impractical.

3.3 Refined CPSR Strategy

In this section we present a refined CPSR strategy that circumvents the central problem of CPSR scheduling. It achieves this goal by taking into account the past behavior of the scheduler and summarizing information about transactions that terminated.

Let $h = (L,S)$ be a log, again interpreted as the current output log. Define the set of transactions *active* in h as follows:

$$AC[h] = \{i \in \mathbb{N}/B_i \in \underline{elem}(L) \wedge (E_i \notin \underline{elem}(L) \vee E_i = \underline{last}(L))\}$$

and denote by $PR[h]^+|AC[h]$ the restriction of $PR[h]^+$ to $AC[h]$. The refined CPSR strategy we propose replaces testing the acyclicity of $PR[h]$ by testing if $\neg\exists i(iPR[h]^+|AC[h]i)$. The correctness of this strategy, for both aggressive and conservative schedulers, is based on the following result.

Let the set of *prefixes* of h be defined as

$$PREFIX[h] = \{h' = (L',S') \in WLOG/L' \text{ is a prefix of } L\} .$$

Note: we consider L to be a prefix of itself.

THEOREM 3.2: Let $h = (L,S)$ be in WLOG.

$$(\exists i \in \mathbb{N})(iPR[h]^+i) \text{ iff } (\exists h' \in PREFIX[h])(\exists j \in \mathbb{N})(jPR[h']^+|AC[h']j) .$$

Proof. (See Appendix.)

We discuss the correctness of the refined CPSR strategy only for aggressive schedulers, since the case of conservative schedulers is entirely similar. We must show that if the aggressive scheduler guarantees that

$\neg\exists i(iPR[h]^+|AC[h]i)$ then it also guarantees that $h \in CPSR$, for each reduced output log h at the time a transaction terminates. Note that the converse is trivially true, since $h \in CPSR$ iff $\neg\exists i(iPR[h]^+i)$.

We argue by contradiction. Asume that the scheduler always guarantees that $\neg\exists i(iPR[g]^+|AC[g]i)$, for each reduced output log $g = (K,R)$ such that $\underline{last}(K) = E_j$, $j \in \mathbb{N}$, but $h \notin CPSR$, for some reduced output log $h = (L,S)$ such that $\underline{last}(L) = E_\ell$, $\ell \in \mathbb{N}$. Then, by Theorem 3.2, there is a prefix h' of h and a transaction T_k such that $k(PR[h']^+|AC[h'])k$. Take $h'' = (L'',S'')$ as the smallest prefix of h such that h' is a prefix of h'' and $\underline{last}(h'') = E_k$. Then, $k(PR[h'']^+|AC[h''])k$ holds when k terminates, which violates the scheduler behavior. Hence, any aggressive scheduler guaranteeing that $\neg\exists i(iPR[g]^+|AC[g]i)$ when a transaction terminates, also guarantees that $g \in CPSR$. Therefore, our CPSR strategy works correctly for aggressive schedulers and is, in fact, equivalent to testing if $h \in CPSR$.

The new strategy has a considerable advantage over the old one, though. We will prove that it requires storing basically a subset of $PR[h]^+|AC[h]$ and a new function $\bar{S}[h]: \{R,W\} \times AC[h] \to 2^V$, whereas the old one required keeping $PR[h]^+$ and S. Therefore, we contend that the new strategy is practical, since it uses information which is a function of the set of active transactions, not of the set of all transactions, as the old one.

The following example illustrates the new strategy.

EXAMPLE 3.3: Consider the following log g, which is not in CPSR:

$$g = R_1[w]R_2[y]W_2[w]R_3[z]W_3[y]R_4W_4[z,x]W_1[x]$$

Suppose that messages are sent to the scheduler so that the input log eventually equals g. The new strategy requires keeping, for the current reduced output log h, the set

$$\overline{PR}[h] = \{(i,j) \in PR[h]^+ \mid AC[h] \text{/there is a path } (i,i_1,\ldots,i_q,j) \text{ in } PR[h] \text{ such that } T_{i_p} \text{ has terminated, for all } p \text{ in } [1,q]\}$$

and a new function $\bar{S}[h]: \{R,W\} \times AC[h] \to 2^V$ such that $\bar{S}[h](R,i)$ contains the union of the readsets of all read messages of T_j occurring in h, for all T_j such that there is a path $(i,i_1,\ldots,i_q,j)$ in $PR[h]$ such that T_j and T_{i_p} have terminated, for all p in $[1,q]$; $\bar{S}[h](W,i)$ is similarly defined, but using writesets. Figure 3.1 illustrates how the scheduler would behave. We comment only on steps 5 and 8. In step 5, using $\bar{S}[h](R,1)$, the scheduler discovers that transaction T_1 precedes a transaction that has terminated, T_2, whose readset intersects $S(W_3)$. Hence, the scheduler deduces that $1 PR[h]^+ 3$. Likewise, in step 8 (parenthesized lines), the scheduler deduces that if W_1 is output, then $1 PR[h]^+ 1$. Hence, R_1 must be restarted (last two lines).

We now precisely characterize $\bar{S}[h]$.

DEFINITION 3.2: Let $h = (L,S)$ be in WLOG.

(a) $PRT[h] \subset \mathbb{N}^2$ is defined as follows:

$$i \; PRT[h] j \equiv i \; PR[h] j \wedge E_j \in \underline{elem}(L)$$

(b) $S_0[h]: \{R,W\} \times \mathbb{N} \to 2^V$ is defined as:

$$S_0[h](R,i) = \bigcup_{R_{ij} \in \underline{elem}(L)} S(R_{ij})$$

$$S_0[h](W,i) = \bigcup_{W_{ij} \in \underline{elem}(L)} S(W_{ij})$$

FIGURE 3.1

i	input log g	output log h	$\overline{PR}[h]$	$\bar{S}[h](R,1)/\bar{S}[h](W,1)$
1	R_1	R_1	ø	ø/ø
2	R_1R_2	R_1R_2	ø	ø/ø
3	$R_1R_2W_2$	$R_1R_2W_2$	{(1,2)}	{y}/{w}
4	$R_1R_2W_2R_3$	$R_1R_2W_2R_3$	ø	{y}/{w}
5	$R_1R_2W_2R_3W_3$	$R_1R_2W_2R_3W_3$	{(1,3)}	{y,z}/{w,y}
6	$R_1R_2W_2R_3W_3R_4$	$R_1R_2W_3R_3W_3R_4$	ø	{y,z}/{w,y}
7	$R_1R_2W_2R_3W_3R_4W_4$	$R_1R_2W_3R_3W_3R_4W_4$	{(1,4)}	{y,z}/{w,y,z}
8	$R_1R_2W_2R_3W_3R_4W_4W_1$	$R_1R_2W_3R_3W_3R_4W_4W_1$	ø	{y,z}/{w,y,z}
		$R_2W_3R_3W_3R_4W_4R_1$	ø	ø/ø

NOTE: to save space, we only represented $\bar{S}[h](R,1)$ and $\bar{S}[h](W,1)$.

(c) $\bar{S}[h]: \{R,W\} \times AC[h] \to 2^V$ is defined as:

$$\bar{S}[h](x,i) = \bigcup_{i\ PRT[h]^+ j} S_0[h](x,j), \quad i \in AC[h] \text{ and } x \in \{R,W\}.$$

□

Definition 3.2 is not appropriate, though, because it depends essentially on $PR[h]^+$ and S, which the scheduler is not supposed to keep. We avoid this problem in Theorem 3.3 below. Given a log $h = (L,S)$, let $h^- = (L^-,S^-)$ denote the log obtained by deleting the last symbol of L and the corresponding pair from S. If h represents the current output log, h^- stands for the output log just before the last message was output.

For each $h = (L,S) \in WLOG$, define $S'[h]: \{R,W\} \times AC[h] \to 2^V$ as follows:

(a) If $h = (\Lambda,\emptyset)$, then $S'[h] = \emptyset$

(b) If $\underline{last}(L) \notin \{E_k / k \in \mathbb{N}\}$,

then for each $i \in AC[h]$ and $x \in \{R,W\}$

$$S'[h](x,i) = \underline{if}\ i \notin AC[h^-]\ \underline{then}\ \emptyset\ \underline{else}\ S'[h^-](x,i)$$

(c) If $\underline{last}(L) = E_k$, for some $k \in \mathbb{N}$, then for each $x \in \{R,W\}$

(i) for each $i \in AC[h]$ such that $\neg i(PRT[h]^* \circ PR[h]|AC[h])k$

$$S'[h](x,i) = S'[h^-](x,i)$$

(ii) for each $i \in AC[h]$ such that $i(PRT[h]^* \circ PR[h]|AC[h])k$

$$S'[h](x,i) = S'[h^-](x,i) \cup S'[h](x,k) \cup S_0[h](x,k).$$

THEOREM 3.3: For any $h = (L,S) \in WLOG$, $S'[h] = \bar{S}[h]$.

Proof. (See Appendix.)

The recursive equations in Theorem 3.3 permit us to compute $\bar{S}[h]$ from $AC[h^-]$, $\bar{S}[h^-]$, $AC[h]$, $PRT[h]^* \circ PR[h]|AC[h]$ and $\bar{S}_0[h]$, the restriction

of $S_0[h]$ to the transactions active in h. We now prove that $PR[h]^+|AC[h]$ can be computed from $PRT[h]^* \circ PR[h]|AC[h]$, and $PRT[h]^* \circ PR[h]|AC[h]$ can in turn be computed from $\bar{S}[h^-]$, $\bar{S}_0[h]$, $AC[h]$ and $PRT[h^-]^* \circ PR[h^-]|AC[h]$.

THEOREM 3.4: Let $h = (L,S)$ be in WLOG.

Define $PRD[h]$, $PWR[h]$, $PTE[h] \subset \mathbb{N}^2$ as follows:

(i) $i\ PRD[h]j \equiv i \in AC[h] \wedge \exists k (R_{jk} = \underline{last}(L) \wedge$
$(S_0[h](W,i) \cap S(R_{jk}) \neq \emptyset \wedge i \neq j \vee \bar{S}[h^-](W,i) \cap S(R_{jk}) \neq \emptyset))$

(ii) $i\ PWRT[h]j \equiv i \in AC[h] \wedge \exists k (W_{jk} = last(L) \wedge$
$((S_0[h](W,i) \cup S_0[h](R,i)) \cap S(W_{jk}) \neq \emptyset \wedge i \neq j \vee$
$(\bar{S}[h^-](W,i) \cup \bar{S}[h^-](R,i)) \cap S(W_{jk}) \neq \emptyset))$

(iii) $i\ PTE[h]j \equiv E_j = \underline{last}(L) \wedge i(PRT[h^-]^* \circ PR[h^-]|AC[h])j$

Then

(a) $PRT[h]^* \circ PR[h]|AC[h] = PRT[h^-]^* \circ PR[h^-]|AC[h]$
$\cup\ PRD[h]$
$\cup\ PWR[h]$
$\cup\ PTE[h] \circ (PRT[h^-]^* \circ PR[h^-]|AC[h])$

(b) $PR[h]^+|AC[h] = (PRT[h]^* \circ PR[h]|AC[h])^+$.

We can now give a full description of the new CPSR strategy we propose. The scheduler will keep $AC[h]$, $\bar{S}[h]$, $PRT[h]^* \circ PR[h]|AC[h]$ and $\bar{S}_0[h]$. Initially $h = (\Lambda,\emptyset)$ and all these objects are empty. When the scheduler outputs a new message, a new output log h is created from the previous one h^-. The scheduler then updates its internal structures as follows:

- $AC[h]$ can be constructed directly, using $AC[h^-]$;
- $PRT[h]^* \circ PR[h]|AC[h]$ can be updated using $PRT[h^-]^* \circ PR[h^-]|AC[h]$, $\bar{S}[h^-]$ and $\bar{S}_0[h^-]$, by Theorem 3.4;
- $\tilde{S}_0[h]$ can be built from $\bar{S}_0[h^-]$ and the message just output;
- $\bar{S}[h]$ can be constructed from $AC[h^-]$, $\bar{S}[h^-]$, $AC[h]$, $PRT[h]^* \circ PR[h]|AC[h]$ and $\bar{S}_0[h^-]$, by Theorem 3.3.

A conservative scheduler will guarantee that the output log formed just after outputting any message satisfies $\neg\exists i(i\ PR[h]^+|AC[h]i)$, while an aggressive scheduler guarantees the same property just after each transaction terminates.

The space bounds of the two CPSR strategies can be derived as follows. Let T be the set of all transactions, T_A be the largest set of simultaneously active transactions, M be the set of all messages issued and V be the set of database variables. Then, the space required to store $PR[h]$, S, $AC[h]$, $\bar{S}[h]$, $PRT[h]^* \circ PR[h]|AC[h]$ and $\bar{S}_0[h]$ is $O(|T|^2)$, $O(|V||M|)$, $O(|T_A|)$, $O(|V||T_A|)$, $O(|T_A|^2)$ and $O(|V||T_A|)$, respectively. Hence, the basic CPSR strategy requires $O(|T|^2 + |V||M|)$ space, while the refined one needs $O(|T_A|^2 + |V||T_A|)$. The time bounds of the two strategies are also derived.

Although we have already discussed the advantages of the refined CPSR strategy, we stress the importance of our conclusion. We obtained, albeit in sketchy form, a practical CPSR test. This result was achieved by "summarizing" (via the union operations taken to define $\bar{S}[h]$) information about transactions that terminate into space proportional to the number of currently active transactions.

We close this section with some variations on the above strategy. Consider first the problem of implementing the refined CPSR strategy using limited memory. Since $\bar{S}[h]$ poses the biggest problem as far as space is concerned, we suggest the following. $\bar{S}[h](x,i)$ can now have either the usual value or *true*, indicating that $\bar{S}[h](x,i) = V$, where V is the set of database variables. When the scheduler runs out of memory, it selects the largest $\bar{S}[h](x,i)$ and sets it to *true*, thereby freeing some memory. Since V includes the previous value of $\bar{S}[h](x,i)$, the scheduler will now be more conservative, but will still produce an output log in CPSR.

Consider now a more realistic database model where the readsets and writesets are described by predicates, rather than lists of variables. All our results remain essentially unchanged, observing that $\bar{S}[h](x,i)$ is now a disjunct, rather than a union, of readsets or writesets. If $\bar{S}[h](x,i)$ becomes too long, we can set it to *true*, exactly as before. However, we now have the option of simplifying the disjunct, thereby obtaining a smaller description of $\bar{S}[h](x,i)$. The predicates allowed must be simple enough to permit the construction of fast simplication procedures, though.

We also note here that an implementation of the refined CPSR strategy depends on quite familiar algorithms--testing if the intersection of two sets is non-empty and detecting cycles in a graph--which require no further discussion. Interestingly enough, an implementation of two-phase locking [ES] also depends on exactly the same algorithms, since granting a lock depends on checking intersections of sets and deadlock detection reduces to cycle detection.

4. AN AGGRESSIVE CPSR SCHEDULER

In this section, we discuss how to construct an aggressive general purpose scheduler SC for serializability, using the results in Section 3. Briefly, SC will have two properties:

P1 if the last symbol of the current reduced output log h is E_k (transaction T_k was granted termination), $h \in$ CPSR;

P2 all transactions terminate.

By Theorem 3.1, P1 and P2 imply that any database system using SC is serializable. To achieve P1, we use the refined CPSR strategy of Section 3.3 and a *restart strategy* to correct scheduling mistakes by restarting transactions. Property P2 requires a *termination strategy*, guaranteeing that all transactions terminate.

Section 4.1 discussed how to use the refined CPSR strategy, and Sections 4.2 and 4.3 concentrate on restart and termination strategies, respectively.

The results in this section depend on a simpler transaction model, that we now describe. We assume here that each transaction executes only one write operation, which is also interpreted as an end operation (termination request). In terms of logs, the symbol E_i always immediately precedes W_i (dropping the second subscript). Our assumption abstracts a reliability subsystem operating as follows [LO]. All updates made by each transaction t are kept in a local workspace invisible to other transactions. When t terminates, the updates are then installed in the database (or *committed*) by a single atomic action, thereby becoming available

to other transactions. Therefore, the database state is always consistent and, as far as the database is concerned, each transaction either runs to completion, or not at all. Moreover, if a transaction t is restarted before terminating, no database rollback is necessary, and no other transaction t' needs to be restarted because it read t's output.

4.1 Using the Refined CPSR Strategy

Most of the problems regarding the refined CPSR strategy were already discussed in Section 3.3. We only have to analyze the effect of restarts. Let $h = (L,S)$ be the current reduced output log and $h' = (L',S')$ be the reduced output log after T_i is restarted (assuming T_i is active in h). Then, L' is L with all symbols $R_{ij} \in \Sigma$ deleted and S' is the restriction of S to the symbols occurring in L'. In view of Section 3.3, the scheduler must then construct $AC[h']$, $\bar{S}_0[h']$, $\bar{S}[h']$ and $PRT[h']^* \circ PR[h'] | AC[h']$ as follows:

- $AC[h'] = AC[h] - \{i\}$
- $\bar{S}_0[h'] = \bar{S}_0[h] | \{R,W\} \times AC[h']$
- $\bar{S}[h'] = \bar{S}[h] | \{R,W\} \times AC[h']$
- $PRT[h']^* \circ PR[h'] | AC[h'] = PRT[h]^* \circ PR[h] | AC[h']$.

Note that the situation would be considerably more complex if we allowed restarting transactions that had already written on the database.

4.2 Restart Strategies

We now discuss how scheduling mistakes can be corrected by restarting transactions. According to the aggressive strategy, the scheduler only tests the reduced output log when a transaction requests termination. So, let $h = (L,S)$ be the current reduced output log and suppose that the scheduler tries to output the end message of T_i. Since we adopted the refined CPSR strategy, a scheduling mistake was made if there is a j in $AC[h]$ such that $jPR[h]^+j$.

The important observation here is that all scheduling mistakes can be corrected by restarting only active transactions (before they actually wrote on the database). For suppose that there is j in $AC[h]$ such that $jPR[h]^+j$. Then, by definition of $AC[h]$ and assumption about transactions, either E_j is not in $\underline{elem}(L)$ or the last two symbols of L are W_jE_j. Since T_j issues no other write message, only a read message R_{jk} of T_j can force T_j to precede some other transaction Therefore, if T_j is restarted, all its read messages will be deleted from L and, in the new log h', $\neg jP\dot{R}[h']^+j$.

Finally, we note that if there is a choice of transactions to restart, it should try to minimize the amount of computation that must be redone. Any optimization must be based solely on the information kept by the scheduler (e.g. $PRT[h]^* \circ PR[h]|AC[h]$).

4.3 Termination Strategies

A termination strategy must guarantee that each transaction always terminates. In an aggressive scheduler, this reduces to guaranteeing that a transaction is not continually restarted. We first observe that cyclic

restarts [RO], a situation in which one transaction causes the restart of another and vice-versa, continually, cannot occur here since each transaction executes at most one write operation. We have to consider only the case of a transaction being repeatedly restarted by a set of different transactions (of course, we can beg the question by assuming a finite number of transactions). The following example illustrates this phenomenon:

EXAMPLE 3.4: Consider the following output log:

$$h = B_1R_1[x]B_2R_2W_2[x]E_2(W_1[x]E_1)R_1[x]\ldots$$
$$\ldots \quad B_iR_iW_i[x]E_i(W_1[x]E_1)R_1[x]\ldots$$

h should be understood as follows. When $W_1[x]$ and E_1 are received, the scheduler tries to output them (symbols within parenthesis), but discovers that the CPSR condition is violated. Then, to avoid the violation, T_1 is restarted and sends a read message again. This scenario is then repeated indefinitely. □

We can face continual restart from two diametrically opposed positions. First, we may optimistically assume that, for the application in question, transactions will be restarted with very low probability and that repeated restarts are independent events. Therefore, the probability of a transaction being repeatedly restarted is quite low. But certain systems clearly do not satisfy these assumptions.

EXAMPLE 3.5: Consider a banking system containing a transaction ASSETS summing up all balances, and a series of transactions TRANSFER[d,n,m] transferring d dollars from account n to account m. Consider now the following sequence of operations:

ASSETS reads the balance of n

TRANSFER reads and writes on the balance of n and m

ASSETS reads the balance of m.

Then, under CPSR, ASSETS precedes and succeeds TRANSFER[d,n,m] and, hence, must be restarted.

Assume that TRANSFER is very common. Then, since ASSETS takes a long time to execute and reads the entire database, the probability of restarting ASSETS repeatedly can be very high. A situation might arise where ASSETS is continually restarted until it runs alone. □

To handle systems like that of Example 3.5, conservative schedulers have to be built. If the scheduler has access to the readsets and the writesets of each transaction when the transaction starts, then the scheduler can avoid restarts entirely [CA1]. The situation here parallels deadlock avoidance. Otherwise, we may adopt the following conservative scheduling strategy. After a transaction T_i is restarted once, the scheduler assumes that T_i will read and write on the whole database. Then, to satisfy the CPSR condition, no transaction running concurrently with T_i is allowed to write on any data item T_i previously read. Any write operation violating this condition is delayed until T_i terminates. The final output log is in CPSR and each transaction is guaranteed to be restarted at most once, but the response time is likely to increase due to the introduction of extra delays.

5. EXTENDING THE RESULTS TO A DISTRIBUTED DATABASE SYSTEM

We now turn to the concurrency control problem for distributed database systems. We consider two extensions of the concepts of Section 2 to a distributed environment and, for each one, we discuss the problem of constructing general purpose schedulers for serializability.

5.1 A Distributed Database System with a Centralized Scheduler

Let $G = (N,E)$ be the graph representing a network, with $n = |N|$. A *distributed database* over G is a set $DDB = \{DB_1,\ldots,DB_n\}$ such that, for all i,j in N, $i \neq j$, $DB_i = (V_i,A_i)$ is a database, $A_i = A_j$ and $V_i \cap V_j \neq \emptyset$. A *distributed database system* over G (with centralized scheduler) consists of a set $T = \{T_1,\ldots,T_m\}$ of *transactions*, a *scheduler* SC, a set $TM = \{TM_1,\ldots,TM_n\}$ of *transaction managers*, a set $DM = \{DM_1,\ldots,DM_n\}$ of *data managers*, and a distributed database over G. We consider that DB_i, TM_i and DM_i *reside* in node i, i in N.

We assume from the beginning a transaction model such as that described in Section 4. Each transaction T_i runs at a single node $t_i \in M$ and starts and terminates by executing a begin and an end operation, respectively. T_i may execute several read operations, but at most one write operation, which must immediately precede the end operation.

Each operation executed by a transaction T_k (running on node ℓ) actually sends a message to the scheduler, as in a centralized database system. When the scheduler outputs a read message, with readset RS, it breaks the message into a set of read messages $RM_1,\ldots,RM_q$, with readsets $RS_1,\ldots,RS_q$, which are sent to a set of data managers $DM_{i_1},\ldots,DM_{i_q}$. The

messages are generated so that $RS = \bigcup_{i=1}^{q} RS_i$ and $RS_{i_j} \subset V_{i_j}$, j in $[1,q]$. DM_{i_j} processes RM_{i_j} as in the centralized case, sending to the transaction manager TM_ℓ of node ℓ the appropriate data values. TM_ℓ is responsible for assembling the final answer to the transaction's read message. A write message is processed likewise.

It should be clear that the scheduler operates exactly as in a centralized database system. The breaking of messages, forced by the distributed nature of the database, does not affect scheduling at all. Hence, the concept of log in Section 2.1 and all the results in Sections 2.2 and 3 hold here without change. Hence, constructing a general purpose scheduler for serializability presents no new problem.

Finally, we observe that an aggressive scheduler constructed along the lines of Section 4 will be functionally identical to a centralized locking scheduler for a distributed database system. Both require sending a message to the centralized scheduler (a lock request in the second case), before each local data manager can service a transaction operation.

5.2 A Distributed Database System with a Distributed Scheduler

Distributed database systems with distributed schedulers differ from the model in Section 5.1 in that they have a set $SC = \{SC_1,\ldots,SC_n\}$, called a *distributed scheduler*, instead of a centralized scheduler. The transaction model remains the same, but the processing of operations differs considerably. Let T_k be a transaction running on node ℓ. Each read operation of T_k is broken by TM_ℓ into a set of read messages $RM_1,\ldots,RM_q$, with readsets $RS_1,\ldots,RS_q$, which are sent to the schedulers $SC_{i_1},\ldots,SC_{i_q}$.

The messages are generated so that $RS = \cup_{i=1}^{q} RS_i$ and $RS_{i_j} \subset V_{i_j}$, j in $[1,q]$. SC_{i_j} and DM_{i_j} process RM_{i_j} as in the centralized case, except that DM_{i_j} sends back to TM_ℓ the appropriate data values. TM_ℓ then assembles the final answer to the read operation. A write operation is processed likewise. The begin operation of T_k is absorbed by TM_ℓ, which sends a begin message to a scheduler when T_k accesses that scheduler for the first time. The end operation and end messages follow a similar procedure.

For each node i, we define the *local reduced output log* h_i and the relation $PR[h_i]$ as in Section 2.1, except that we use the alphabet $\Sigma_i = \{B_j^i, E_j^i, R_{jk}^i, W_{jk}^i / j,k \in \mathbb{N}\}$. If $h_1,\ldots,h_n$ are the local reduced output logs of the nodes in N, a *global reduced output log* h of the distributed database system is any merge of $h_1,\ldots,h_n$ representing a possible execution of the transactions. Note that, irrespective of this last assumption, since $V_i \cap V_j = \emptyset$, we trivially have

$$PR[h] = \bigcup_{i \in N} PR[h_i]$$

Using the concept of a global log, we can define the notions of general purpose distributed scheduler and aggressive or conservative strategies.

We now briefly discuss the concurrency control problem for these systems, concentrating on the problems of using the results of Section 3 to construct an aggressive general purpose distributed scheduler.

The refined CPSR strategy remains the same, if we consider the global log. A way to implement it goes as follows. SC_i, the scheduler residing at node i, will keep $AC[h_i]$, $\bar{S}[h_i]$, $PRT[h_i]^* \circ PR[h_i]|AC[h_i]$ and $\bar{S}_0[h_i]$, exactly as discussed in Section 3. Note that SC_i can update its own information without consulting other schedulers.

However, granting termination to a transaction raises two difficult problems. Suppose that transaction T_j requests to terminate by writing on the database stored at nodes in the set J via messages W_j^k, $k \in J$. Let h_i be the current reduced output log at node i. Then, to grant termination to T_j, we have to apply the test (which is a correct improvement of the method discussed in Section 4):

$$\neg k \ (\bigcup_{i \in N} PRT[h_i'] *_\circ PR[h_i'] \,|\, AC[h_i])k$$

where $h_\ell' = h_\ell$, for $\ell \in N-J$, and h_k' is obtained by adding W_j^k to h_k, $k \in J$. Hence, we must face two problems:

- applying the test may require visiting several, if not all nodes of the network;
- all nodes visited must be properly synchronized so that we obtain a correct picture of the information about their local logs we need.

The following example illustrates the situation.

EXAMPLE 5.1: The following scenario illustrates the problem of testing if a transaction may terminate.

time / nodes		1	2	3
a	message	$R_i[x]$	$W_j[x]$	-
	precedes	$\emptyset$	$i<j$	$i<j$
b	message	$R_j[y]$	$W_k[y]$	-
	precedes	$\emptyset$	$j<k$	$j<k$
c	message	$W_k[z]$	-	$W_i[z]$?
	precedes	$\emptyset$	$\emptyset$	$k<i$?

At time 3, transaction T_i sends $W_i[z]$ to SC_c and requests to terminate. Although T_i had not visited node b, its termination request depends on information stored there to discover the cycle $i < j < k < i$. Moreover, the timeliness of such information is crucial. It is worth noting that the node b problem can be generalized to a cycle spanning any number of nodes. □

We can circumvent the first problem by choosing one of the local schedulers, say SC_k, as the *master scheduler*, which is responsible for keeping at all times $U_{i \in N} PRT[h_i]^* \circ PR[h_i] | AC[h_i]$. Each local scheduler SC_i must then send to SC_k any changes made to $PRT[h_i]^* \circ PR[h_i] | AC[h_i]$. However, we know of no straightforward way to solve the second problem, without introducing excessive inter-site communication. And, we doubt that a straightforward solution is possible. Therefore, the applicability of the refined CPSR strategy to the construction of distributed schedulers is left open pending further investigation.

6. CONCLUSIONS

The main result of this paper spells out how to implement CPSR schedulers within practical space bounds. We emphasize that, although CPSR was suggested as theoretically a way to synchronized database systems, no correct practical implementation existed.

We suggest that further work should be done in two directions. First, investigate under what conditions is CPSR superior to two-phase locking. Second, study the feasibility of constructing distributed CPSR schedulers without excessive intersite communication.

REFERENCES

[BE1] Bernstein, P.A., Shipman, D.W., Rothnie, J.B., Goodman, N. "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases." ACM Trans. on Database Sys. 5, 1 (March 1980), pp. 1-17.

[BE2] Bernstein, P.A., Shipman, D.W., Wong, W.S. "A Formal Model of Concurrency Control Mechanisms for Database Systems." IEEE Transactions on Software Engineering (May 1979).

[CA1] Casanova, M.A., Bernstein, P.A. "General Purpose Schedulers for Database Systems." Acta Informatica (to appear).

[CA2] Casanova, M.A. "The Concurrency Control Problem for Database Systems." Ph.D. Dissertation, Harvard University (Nov. 1979).

[ES] Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Database System." CACM 19, 11 (Nov. 1976), 624-633.

[GA] Gardarin, G. and Lebeaux, P. "Scheduling Algorithms for Avoiding Inconsistency in Large Databases." Proc. 1977 Int. Conf. on Very Large Data Bases, 501-516.

[IB] IBM "Information Management System Virtual Storage (IMS/VS) General Information Manual." IBM Form No. GH20-1260.

[KU] Kung, H.T. and Papadimitriou, C.H. "An Optimality Theory of Concurrency Control for Databases." Proc. 1979 ACM-SIGMOD Int. Conf. on Management of Data (June 1979).

[LO] Lorie, R.A. "Physical Integrity in a Large Segmented Database." ACM Trans. on Database Systems, Vol. 2, No. 1 (March 1977), pp. 91-104.

[PA1] Papadimitriou, C.H., Bernstein, P.A., Rothnie, J.B. "Some Computational Problems Related to Database Concurrency Control." Proc. of the Conf. on Theoretical Computer Science (Aug. 1977), 275-282.

[PA2] Papadimitriou, C.H. "Serializability of Concurrent Updates." JACM, Vol. 26, No. 4 (Oct. 1979), pp. 631-653.

[RO] Rosenkrantz, D.J., Stearns, R.E., Lewis, P.M. "System Level Concurrency Control for Distributed Database Systems." TODS 3, 2 (June 1978), 178-198.

[SC] Schlageter, G. "Process Synchronization in Database Systems." TODS 3, 3 (Sept. 1978), 248-271.

[ST] Stearns, R.E., Lewis, P.M., Rosenkrantz, D.J. "Concurrency Control for Database Systems." Proc. of the 17th IEEE Symp. on Foundations of Computer Science (Oct. 1976), 19-32.

[TH] Thomas, R.H. "A Solution to the Update Problem for Multiple Copy Databases which Uses Distributed Control." TODS 4, 2 (June 1979), 180-209.

APPENDIX

THEOREM 3.1: Let DBS be a database system.

If any reduced output log of DBS is in CPSR and all transactions terminate, then DBS preserves consistency.

Sketch of Proof.

Let C be a computation of DBS and $h = (L,S)$ be the corresponding reduced output log. We first observe that if two adjacent symbols in L, x and y, are such that none is a write or $S(x) \cap S(y) = \emptyset$, then we can commute their order in L without affecting the final database state produced by C. Now, since $h \in$ CPSR, PR[h] is a partial order. Let R be any total order compatible with PR[h]. Since, by assumption all transactions were executed to completion in C, R is a total order on T, the set of transactions of DBS. Using the previous observation about commutativity, it can be shown that there is a sequential computation C' of the transactions in T, one after the other in the order dictated by R, such that C and C' have the same initial and final database states. Therefore, we may conclude that DBS is serializable. □

THEOREM 3.2: Let $h = (L,S)$ be in WLOG.

$(\exists i \in \mathbb{N})(i\ PR[h]^{+} i)$ iff $(\exists h' \in PREFIX[h])(\exists j \in \mathbb{N})(j\ PR[h']^{+} j \wedge AC[h']j)$.

Proof.

($\Leftarrow$) The result follows trivially since $PR[h'] \subset PR[h]$, if h' is a prefix of h.

($\Rightarrow$) Assume $i\ PR[h]^+ i$. Let $C = (i_0, i_1, \ldots, i_k, i_0)$ be the smallest cycle in $PR[h]$ containing $i_0 = i$. Let $h' = (L', S')$ be the smallest prefix of h such that C is also a cycle in $PR[h']$. Then $\underline{last}(L') \in \{R_{i_j,m}, W_{i_j,m}\}$, for some m in $[1,\infty)$ and j in $[0,k]$, as otherwise h' is not the smallest such prefix. But then $i_j \in AC[h']$ and we are done. □

THEOREM 3.3: For each $h = (L,S) \in WLOG$, define $S'[h]: \{R,W\} \times AC]h] \to 2^V$ as follows:

(a) If $h = (\Lambda, \emptyset)$, then $S'[h] = \emptyset$.

(b) If $\underline{last}(L) \notin \{E_k / k \in \mathbb{N}\}$,

then for each $i \in AC[h]$ and $x \in \{R,W\}$

$S'[h](x,i) = \underline{if}\ i \in AC[h^-]\ \underline{then}\ \emptyset\ \underline{else}\ S'[h^-](x,i)$

(c) If $\underline{last}(L) = E_k$, for some $k \in \mathbb{N}$, then for each $x \in \{R,W\}$

(i) for each $i \in AC[h]$ such that $\neg i(PRT[h]^* \circ PR[h]|AC[h])k$

$S'[h](x,i) = S'[h^-](x,i)$

(ii) for each $i \in AC[h]$ such that $i(PRT[h]^* \circ PR[h]|AC[h])k$

$S'[h](x,i) = S'[h^-](x,i) \cup S'[h](x,k) \cup S_0[h](x,k)$.

Proof. We prove that $S'[h] = \bar{S}[h]$, for any log $h = (L,S)$, by induction on $|L|$.

basis: $|L| = 0$. Trivial.

induction step: Suppose that for all logs $g = (K,R)$, with $|K| = n$, $S'[g] = \bar{S}[g]$. Let $h = (L,S)$ be a log such that $|L| = n+1$. We prove that $S'[h] = \bar{S}[h]$. Let $i \in AC[h]$ and $x \in \{R,W\}$.

case 1: Suppose that $\underline{last}(L) \notin \{E_k / k \in \mathbb{N}\}$.

case 1.1: $i \notin AC[h^-]$. We trivially have

(1) $S'[h](x,i) = \bar{S}[h](x,i) = \emptyset$

case 1.2: $i \in AC[h^-]$. Since $PRT[h] = PR[h^-]$ and by the induction hypothesis, we have

(2) $S'[h](x,i) = \bar{S}[h^-](x,i) = \bar{S}[h](x,i)$.

case 2: Suppose that $\underline{last}(L) = E_k$, for $k \in \mathbb{N}$.

case 2.1: $\neg i(PRT[h]^* \circ PR[h]|AC[h])k$.

We first observe that $i = k$ is included here. We have to prove that:

(3) $S[h](x,i) = S'[h^-](x,i)$

By assumptions, we have

(4) $i\ PRT[h]^+ j$ iff $i\ PRT[h^-]j$

Then, by (4), definition of $\bar{S}[h]$ and the induction hypothesis, we have:

(5) $\bar{S}[h](x,i) = \bar{S}[h^-](x,i) = S'[h^-](x,i)$.

case 2.2: $i(PRT[h]^* \circ PR[h]|AC[h])k$

We have to prove that

(6) $\bar{S}[h](x,i) = S'[h^-](x,i) \cup S'[h](x,k) \cup S_0[h](x,k)$.

That the r.h.s. of (6) is a subset of the l.h.s. follows trivially.

We prove that the l.h.s. is a subset of the r.h.s. From the definition of $\bar{S}[h](x,i)$, it suffices to prove that $S_0[h](x,j)$ is a subset of the r.h.s. of (6), for each $j \in \mathbb{N}$ such that $i\ PRT[h]^+ j$.

case 2.2.1: $j = k$. Trivial.

case 2.2.2: $i\ PRT[h^-]^+ j \wedge j \neq k$.

By assumption of case 2, we have:

(7) $S_0[h](x,j) = S_0[h^-](x,j)$

Therefore, by definition of $\bar{S}[h^-]$ and induction hypothesis, we have:

(8) $S_0[h](x,j) \subset S'[h^-](x,i)$

case 2.2.3: $\neg i\ PRT[h^-]^+ j \wedge j \neq k$

By assumptions on i,j and k, we have:

(9) $\quad i\ PRT[h]^+ k \wedge k\ PRT[h]^+ j$

Therefore, by definition of $\bar{S}[h](x,k)$ and (9), we have:

(10) $\quad S_0[h](x,j) \subset \bar{S}[h](x,k)$

Now, by (10) and case 2.1, we have:

(11) $\quad S_0[h](x,j) \subset S'[h](x,k)$.

THEOREM 3.4: Let $h = (L,S)$ be in WLOG.

Define $PRD[h], PWR[h], PTE[h] \subset \mathbb{N}^2$ as follows:

(i) $i\ PRD[h]j \equiv i \in AC[h] \wedge \exists k(R_{j_k} = \underline{last}(L) \wedge$
$(S_0[h](W,i) \cap S(R_{j_k}) \neq \emptyset \wedge i \neq j \vee \bar{S}[h^-](W,i) \cap S(R_{j_k}) \neq \emptyset))$

(ii) $i\ PWR[h]j \equiv i \in AC[h] \wedge \exists k(W_{j_k} = \underline{last}(L) \wedge$
$((S_0[h](W,i) \cup S_0[h](R,i)) \cap S(W_{j_k}) \neq \emptyset \wedge i \neq j \vee$
$(S[h^-](W,i) \cup \bar{S}[h^-](R,i)) \cap S(W_{j_k}) \neq \emptyset))$

(iii) $i\ PTE[h]j \equiv E_j = \underline{last}(L) \wedge i(PRT[h^-]^* \circ PR[h^-]|AC[h])j$

Then

(a) $PRT[h]^* \circ PR[h]|AC[h] = PRT[h^-]^* \circ PR[h^-]|AC[h]$
$\cup\ PRD[h]$
$\cup\ PWR[h]$
$\cup\ PTE[h] \circ (PRT[h^-]^* \circ PR[h^-]|AC[h])$.

(b) $PR[h]^+|AC[h] = (PRT[h]^* \circ PR[h]|AC[h])^+$.

Proof.

(a) The r.h.s. is trivially a subset of the l.h.s., so we only prove the converse. Let p and q be such that

(1) $\quad p(PRT[h]^* \circ PR[h] | AC[h])q$

case 1: $p(PRT[h^-]^* \circ PR[h^-] | AC[h])q$. Trivial.

case 2: $\neg p(PRT[h^-]^* \circ PR[h^-] | AC[h])q$.

case 2.1: $\underline{last}(L) = R_{q_k}$, for some $k \in \mathbb{N}$.

By (1) and assumption, there is $r \in \mathbb{N}$ such that

(2) $\quad p\ PRT[h^-]^* r \wedge r\ PR[h] q \wedge \neg r\ PR[h^-] q$

By (2) and assumption of case 2.1, we must have:

(3) $\quad S_0[h^-](W,r) \cap S(R_{q_k}) \neq \emptyset$

If $p \neq r$, by definition of $\bar{S}[h^-]$, we have

(4) $\quad S_0[h^-](W,r) \subset \bar{S}[h^-](W,p)$

Hence, by (3), (4) and definition of PRD[h], we obtain

(5) $\quad p\ PRD[h] q$

case 2.2: $\underline{last}(L) = W_{q_k}$, for some $k \in \mathbb{N}$.

Similarly to case 2.1, we obtain

(6) $\quad p\ PWR[h] q$

case 2.3: $\underline{last}(L) \in \{B_q, E_q\}$.

(7) $\quad p\ PR[h] q \equiv p\ PR[h^-] q$

Therefore, by assumption of case 2, we have

(8) $\quad \neg p(PRT[h]^* \circ PR[h] | AC[h])q$

which contradicts (1).

case 2.4: $\underline{last}(L) \in \{R_{rk}, W_{rk}, B_r\}$, $r \neq q$, for some $k \in \mathbb{N}$.

Follows similarly to case 2.3.

case 2.5: $\underline{last}(L) = E_r$, $r \neq q$.

Then, by assumption of case 2, (1) can be rewritten as:

(9) $\quad p\ PRT[h]^* r \wedge r(PRT[h]^* \circ PR[h] \| AC[h])q$

By assumption, definition of PR[h] and PRT[h], we have:

(10) $\quad r(PRT[h^-]^* \circ PR[h^-] \| AC[h])q$

If $r = p$, we are done. So assume $r \neq p$. Then, by (9) we have:

(11) $\quad p\ PRT[h]^* \circ PR[h] r$

Again, by assumption and definition of PR[h] and PRT[h], we have:

(12) $\quad p\ PRT[h^-]^* \circ PR[h^-] r \wedge \underline{last}(L) = E_r$

Hence, from (10), (12) and definition of PTE[h], we finally obtain:

(13) $\quad p\ PTE[h] \circ (PRT[h^-]^* \circ PR[h^-] \| AC[h])q$

This completes part (a).

(b) The r.h.s. is trivially a subset of the l.h.s., so we only prove the converse. Let p and q be such that

(1) $\quad p\ PR[h]^+ \| AC[h] q$

Then, there is a path $P = (p_0, p_1, \ldots, p_m)$ in PR[h] such that $p = p_0$ and $q = p_m$, for some $m \in \mathbb{N}$. Let $p_{i_1}, \ldots, p_{i_k}$, with $0 \leq i_j \leq m$ and $1 \leq j \leq k$, be the set of all elements of P such that $p_{i_j} \in AC[h]$. Then, by definition of AC[h], we have:

(2) $P_{i_j}(PRT[h]^* \circ PR[h]|AC[h])P_{i_{j+1}}$ for each j in [1,k)

which implies that

(2) $p(PRT[h]^* \circ PR[h]|AC[h])^+q$.