Theoretical Computer Science 76 (1990) 93-114
North-Holland

PARALLEL UPDATE TRANSACTIONS *

Dino KARABEG** and Victor VIANU**

Department of Computer Science and Engineering, Mail Code MC-014, University of California at San Diego, La Jolla, CA 92093, USA

1. Introduction

While query languages have been extensively studied in the framework of the relational model, database updates and transactions have only recently become the object of formal investigation. Indeed, most studies of transactions have focused on concurrency issues [2, 7]. In [1, 5], a formal model for sequential update transactions in relational databases was introduced, and several basic results on transaction equivalence and optimization were obtained. In the present paper we introduce a model for *parallel* update transactions, which is an extension of the model developed in [1] for sequential transactions. Our results focus on the problem of maximizing the degree of parallelism within parallel transactions, and producing optimal parallelizations of sequential transactions.

Parallel transactions are viewed here as partially ordered sets of atomic database updates forming a semantic unit. We consider a widely accepted class of atomic updates. These consist of insertions, deletions, and modifications, where the selection of tuples (to be deleted or modified) involves the inspection of individual attribute values for each tuple. We first look at the specification of parallel transactions. Since specifying a parallel transaction by a partially ordered set of updates can be awkward, we propose a more convenient syntax, called "in-line", and examine its relationship to partially ordered sets of updates. While the "in-line" syntax is more restrictive, we show that it is powerful enough to capture the same relevant information on the degree of parallelism as arbitrary partial orders of updates.

The main results of the paper focus on algorithms for maximizing the degree of parallelism within parallel transactions. The algorithms can be used to optimize given parallel transactions, or to parallelize given sequential transactions. In order to understand the nature of the questions involved, it is useful to consider the following intuitively suggestive problem, corresponding to a special case of the optimization problem. Suppose m boxes $B_1, \ldots, B_m$ are given. Initially, each box B_i is either empty or contains some balls. Balls can be moved among boxes by any

* The authors were supported in part by the National Science Foundation, under Grant numbers IST-8511538 and IRI 8816078.

** On leave from Rudjer Boskovic Institute, Zagreb, Yugoslavia.

0304-3975/90/$03.50 © 1990—Elsevier Science Publishers B.V. (North-Holland)

sequence of *moves*, $m(B_j, B_k)$ each of which consists of putting the entire content of box B_j into box B_k. Suppose that the balls must be redistributed among boxes according to a given mapping f from boxes to boxes ($f(B_j) = B_k$ means that the content of box B_j must wind up in box B_k after the re-distribution). The problem in this case is to find a parallel schedule of moves which accomplishes the re-distribution in minimal time (assuming that each move takes one unit of time). It is shown (Theorem 3.13) that this problem is NP-complete. However, it is also shown that the problem has a very good polynomial-time approximation algorithm, which produces a solution differing from the optimal at most by the absolute constant 1. Such results are of particular interest, since they provide new examples of NP-complete problems with very good polynomial approximations. Also, the type of problem described above is likely to occur in other contexts as well.

The approximation algorithm for the general case produces parallelizations within a constant factor of the optimal.

The paper consists of four sections. Section 2 summarizes the model for sequential transactions of [1]. In Section 3, parallel transactions are defined and the optimization problem is shown to be NP-complete. In Section 4 we present the polynomial-time approximate optimization algorithms.

2. Background on sequential transactions

In this section we review the model of sequential transactions and some results previously obtained in [1].

We assume knowledge of the basic concepts and notation of relational data-bases, as in [6, 8]. We only review here briefly some notation and terminology used in the paper. We assume the existence of an infinite set of symbols, called *attributes*, and for each attribute A, of an infinite set of *values*, denoted dom(A), called the *domain* of A. A *relational schema* is a finite set of attributes. Let U be a relational schema. A *tuple t over U* is a mapping from U such that, for each A in U, $t(A)$ is in dom(A). A *relation* over U is a finite set of tuples over U. A *data-base schema* is a finite set of relation schemas. We usually denote attributes by $A, B, \ldots$, tuples by $t, u, v, \ldots$, relation schemas by $P, Q, R, \ldots$ and database schemas by $\mathbf{P}, \mathbf{Q}, \mathbf{R}, \ldots$.

The sequential transactions we consider are finite sequences of insertions, deletions, and modifications. We focus on the large class of "domain based" transactions, where the selection of tuples to be deleted or modified involves the inspection of individual attribute values of a tuple, independently of other attribute values in the tuple and of other tuples in the relation.

The following is a simple example of a domain-based transaction in SQL [3].

Example 2.1. Suppose a relation EMP (employee) has been defined (its attributes are NAME, DEPT, RANK, and SALARY). The following transaction hires Moe as the new manager of the parts department, with a salary of 30K, then fires all

managers from the parts department other than Moe. Finally, all employees from the parts department who are not managers are transferred to the service department. The rank remains unchanged. The new salary is 20K:

insert into EMP values ('moe', 'parts', 'manager', 30K)

delete from EMP where NAME ≠ 'moe' and DEPT = 'parts'

and RANK = 'manager'

update EMP set DEPT = 'service', SALARY = 20K

where DEPT = 'parts' and RANK ≠ 'manager'.

We now define the notions of a "condition" and satisfaction of a condition by a tuple.

Definition 2.2. Let U be a set of attributes. A *condition* over U is an expression of the form $A = a$ or $A \neq a$, where $A \in U$ and $a \in \text{dom}(A)$. A tuple u over U *satisfies* a condition $A = a$ ($A \neq a$) iff $u(A) = a$ ($u(A) \neq a$). A tuple u *satisfies* a set C of conditions if it satisfies every condition in C.

We do not explicitly use logical connectors to build up complex conditions. It can be easily seen that this would not add power to our transactions. Note however that logical connectors may provide a more succint representation (at most by a factor exponential in the number of attributes and constants in the transaction). Nonetheless, we adopt the natural syntax described above for the sake of simplicity.

Although the conditions use only equality and inequality, this assumption is not central to the development (it is straightforward to extend the conditions so that comparisons of the form $A > a$, $A < a$ are allowed).

In the following, only *satisfiable* sets of conditions are considered, that is, sets of conditions with no mutually exclusive conditions. Two sets of conditions are *incompatible* if there is no tuple satisfying both sets of conditions.

A set of conditions over U is used to specify a set of tuples over U (those satisfying the conditions). Due to the form of our conditions, we use the intuitively suggestive term "hyperplane" to identify such sets of tuples.

Definition 2.3. The *hyerplane* $H(U, C)$ defined by a (satisfiable) set C of conditions over U is the set $\{t \in \text{Tup}(U) \mid t \text{ satisfies } C\}$.

For simplicity, we sometimes use the same notation for a set C of conditions over U and for the hyperplane $H(U, C)$ defined by C. Thus, we say "hyperplane C" instead of "hyperplane $H(U, C)$", whenever U is understood. The *support* of a hyperplane $H(U, C)$ is the set of attributes $\{A \mid A = a \text{ is in } C \text{ for some } a\}$. Thus the support of the hyperplane defined by $\{A = 0, B = 1, C \neq 5\}$ is AB.

We now define the updates used to build our transactions. An *insertion* over a database schema $\boldsymbol{R}$ is an expression $i_X(C)$ where X is a relation schema in $\boldsymbol{R}$ and C is a set of conditions specifying a complete tuple over X. A *deletion* over $\boldsymbol{R}$ is an expression $d_X(C)$, where X is a relation schema in $\boldsymbol{R}$ and C is a set of conditions over X. Finally, a modification over $\boldsymbol{R}$ is an expression $m_X(C_1; C_2)$, where X is a relation schema in $\boldsymbol{R}$, C_1 and C_2 are sets of conditions over X and, for each A in X, either[1] $C_1|_A = C_2|_A$ or $A = a \in C_2$. (The equalities present in C_2 but not in C_1 indicate how tuples in $H(X, C_1)$ are modified.) Note that, if $m(C_1, C_2)$ is a modification, then support $(C_1) \subseteq \text{support}(C_2)$. An *update* is an insertion, deletion, or modification. Following is an example of updates.

Example 2.4. Consider again the database of Example 2.1. The following are updates over U, corresponding to the SQL updates in Example 2.1:

(1) $i_U(\langle \text{moe, parts, manager, 30K} \rangle)$,

(2) $d_U(\text{NAME} \neq \text{Moe, DEPARTMENT} = \text{parts, RANK} = \text{manager})$ (this deletes all managers in the parts department whose names are not Moe),

(3) $m_U(\text{DEPARTMENT} = \text{parts, RANK} \neq \text{manager; DEPARTMENT} = \text{service, RANK} \neq \text{manager, SALARY} = \text{20K})$.

This transfers all employees who are not managers from the parts department to the service department. The rank remains unchanged. The new salary is 20K.

In the following we sometimes omit the subscripts in writing updates. For instance, we write $i(C)$ instead of $i_X(C)$, whenever X is understood.

A *transaction* over a database schema $\boldsymbol{R}$ is a finite sequence of updates over $\boldsymbol{R}$. The semantics of a transaction t is defined by a mapping associating old instances and new instances, called the *effect* of t and denoted by eff(t) (see [A1] for formal definition of effect). Transactions t_1 and t_2 are *equivalent* ($t_1 \equiv t_2$) if they have the same effects.

Since the effects of updates over different relation schemas are independent, we will consider from now on only transactions over uni-relational schemas.

We next introduce a non-procedural method for describing the effect of a transaction on a database. The effect is described at the tuple level using the notion of a "transition". Transitions can be specified in an intuitively appealing manner and are a useful tool. For each tuple, a transition indicates whether the tuple is deleted or, if not, how it is updated. In addition, a transition gives a finite set of inserted tuples. A transition will be specified by first partitioning the space of tuples into sufficiently many hyperplanes. It is assumed that the partition is sufficiently fine so that all tuples in each hyperplane of the partition are either deleted or updated to yield another hyperplane in the partition. This is specified using a *transition graph* whose vertices are the hyperplanes in the partition. If H_1 is modified to H_2, there is an edge from H_1 to H_2. If H_1 is deleted there is no edge leaving H_1. Note that

[1] If C is a set of conditions over U and $A \in U$, then $C|_A$ denotes the set of conditions in C involving attribute A.

each vertex in a transition graph has out-degree at most 1. Also, if there is an edge from H_1 to H_2, then support$(H_1) \subseteq$ support(H_2). In particular, all hyperplanes belonging to a cycle have the same support.

The set of inserted tuples cannot be conveniently specified using the graph, and is given separately. A ***transition specification*** (***spec***) is a pair $\langle G$, Insert$\rangle$, where G is a transition graph and Insert is a set of newly inserted tuples (called the insert set of the transition spec), such that there is no edge $(C1, C2)$ in G with $C2 \in$ Insert. Note that the latter condition is not restrictive, since modifications with inserted tuples as targets are redundant and can be replaced by deletions without changing the effect of the transaction. (In particular, note that this holds also for loops (C_1, C_1) with $C1 \in$ Insert, in which case $C1$ is deleted rather than left unchanged.) The condition will be useful in our parallelization algorithms.

We now give a simple example of a transition spec (see [1] for the formal definition of transition specs and for more elaborate examples).

Example 2.5. Let $U = AB$ and G be the transition graph represented in Fig. 1. Let Insert $= \{\langle 1, 1\rangle\}$. Then $(G$, Insert$)$ is a transition specification over AB. The transition specified by $(G$, Insert$)$ consists of inserting the tuple $\langle 1, 1\rangle$ and deleting all tuples t where $t(A) = 0$ and the tuple $\langle 1, 1\rangle$. All other tuples remain unchanged.

Fig. 1.

We next look at the relation between transactions and transitions. For each transaction there exists a corresponding transition which represents the final effect of the transaction. In order to construct the transition spec corresponding to a transaction, it is first necessary to perform some "preprocessing" of the transaction. Specifically, the transaction is modified so that all hyperplanes corresponding to distinct sets of conditions occurring in the transaction are disjoint. A transaction having this property is said to be in *First Normal Form* (1NF). The 1NF property simplifies considerably our algorithms and results. It is shown in [1] that every transaction can be transformed into an equivalent 1NF transaction by "splitting" every hyperplane occurring in it into sufficiently small hyperplanes. Note that the normalization may increase the length of the transaction by a factor exponential in the number of attributes and constants in the transaction.

Examples 2.6. (i) Consider the transaction over AB: $i(\langle 1, 1\rangle)\ m(\{A = 0\}; \{A = 1, B = 1\})$. The corresponding transition specification is the one of Example 2.5. Note that the hyperplanes $\{A = 0\}$ and $\{A = 1, B = 1\}$ are deleted in the transition spec, rather than modified to $\langle 1, 1\rangle$, because $\langle 1, 1\rangle$ is an inserted tuple.

(ii) The transition specification corresponding to the transaction over A:

$$d(\{A=2\})\ m(\{A=0\};\ \{A=2\})\ m(\{A=1\};\ \{A=0\})\ m(\{A=2\};\ \{A=1\})$$

is $\langle G, \emptyset\rangle$, where G is represented in Fig. 2.

Note that the transition graph in Fig. 2 has a cycle. A transaction whose transition graph has no cycles is called *acyclic*. Thus, the transaction (i) is acyclic. Transaction (ii) is cyclic. Note that transaction (ii) implements the cycle by using the hyperplane $\{A=2\}$ as "temporary storage". It is shown in [1] that a transaction must always use some temporary storage to implement a cycle. It follows that transition graphs where no hyperplanes are "available" for use as temporary storage cannot be realized by any transaction. For instance, if hyperplane $\{A=2\}$ in Fig. 2 is left unchanged rather than deleted, the transition graph is not realizable. A formal characterization of realizable transition specifications is given in [1]. A *syntactic cycle* is a transaction of the form $m(C_1;\ C_T)\ m(C_k;\ C_1)\ m(C_{k-1};\ C_k)\dots m(C_2;\ C_3)\ m(C_T;\ C_2)$. Note that a syntactic cycle implements a cycle involving hyperplanes $C_1,\dots,C_k$, using C_T as temporary storage.

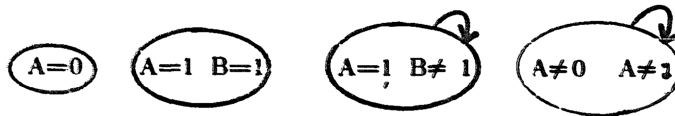

Fig. 2.

In [5], a sound and complete set of axioms for proving transaction equivalence is defined. Of these, we only need in this paper axioms indicating when two updates commute. For example, $d(C_1)\ i(C_2) \approx i(C_2)\ d(C_1)$ (for C_1 and C_2 incompatible) is a commutativity axiom.

3. Parallel transactions

In this section the notion of a parallel transaction is introduced. The "in-line" syntax for specifying parallel transactions is defined and compared to the specification using partial orders of updates. Some technical properties of parallel transactions are established, which help to understand the factors limiting parallelism within a transaction.

Definition 3.1. A *parallel transaction* is a partially ordered set of updates $(A, <)$ such that two updates e_i and e_j in A are incomparable with respect to the ordering relation "$<$" only if e_i and e_j commute.

We will represent a parallel transaction using the Hasse diagram of its partial order, defined as follows.

Definition 3.2. A *Hasse diagram* of a partially ordered set $(S, <)$ is a directed graph (S, E) where $(e_1, e_2) \in E$ if and only if

(i) $e_1 < e_2$ and

(ii) there is no e_3 in S such that $e_1 \neq e_3$, $e_2 \neq e_3$ and $e_1 < e_3 < e_2$.

The Hasse diagram representing the partial order of a parallel transaction $t = (A, <)$ is called the ***parallel transaction graph*** of t (PTG(t)). Note that PTG(t) is a DAG. The *effect* eff(t) of a parallel transaction t is the effect of any sequential transaction t' which is a linear extension of t. The *transition specification* of t is the transition specification of any linear extension of t. Two parallel transactions t_i and t_j are *equivalent* ($t_i \approx t_j$) if and only if $\text{eff}(t_i) = \text{eff}(t_j)$.

It is clear that parallel transactions include the sequential transactions as a special case. It follows from the definition of a parallel transaction that all the sequential transactions that are linear extensions of a given parallel transaction t have the same effect and the same transition specification. Hence, the effect and the transition specification of a parallel transaction are well defined.

We now introduce a measure of complexity of parallel transactions, called the "length". The length of a parallel transaction indicates the maximum number of updates of the transaction which must be performed sequentially. The length of a parallel transaction t is equal to the time required for the execution of t on a parallel computer assuming that the number of available parallel processors is sufficiently large and that each update is performed by a processor in one unit of time.

Definition 3.3. Let $t = (A, <)$ be a parallel transaction. The *parallel time* of update e in t, denoted ptime(e), is the depth of e in the DAG PTG(t) (i.e., the length of a longest sequence of updates $e_1, e_2, \ldots, e_k$ such that $e_i < e_{i+1}$ $(1 \leq i < k)$ and $e_k = e$). The *length* $|t|$ of t is the maximum of ptime(e) over all updates e in A (i.e., the depth of the PTG(t)). For each positive integer τ, the transaction $t^{(\tau)}$ is the parallel transaction defined as the restriction of $t = (A, <)$ to the updates e in t such that $\text{ptime}(e) \leq \tau$. The transaction $t^{(>\tau)}$ is the restriction of $(A, <)$ to the updates e in t such that $\text{ptime}(e) > \tau$. A parallel transaction t^* is *optimal* if there is no parallel transaction t such that $t \approx t^*$ and $|t| < |t^*|$.

The above definition of a parallel transaction has the disadvantage that specifying a parallel transaction involves specifying a parallel transaction graph, which is impractical. To remedy this difficulty we introduce a restricted class of parallel transactions, called the "in-line parallel transactions", which can be specified using a more pleasant syntax.

Definition 3.4. The set IPTrans(U) of *in-line parallel transactions* over a set of attributes U is the set of all expressions obtained by a finite number of applications of the following rules (note that the effect and length of in-line parallel transactions are defined concomitantly):

(i) Each sequential transaction over U is also in IPTrans(U).

(ii) If t_1 and t_2 are in IPTrans(U) then t_1; t_2 is in IPTrans(U), $\text{eff}(t_1; t_2) = \text{eff}(t_1) \circ \text{eff}(t_2)$, and $|t_1; t_2| = |t_1| + |t_2|$.

(iii) If $t_1, \ldots, t_n$ are in IPTrans(U) and for each i, j, $1 \leq i < j \leq n$, each update in t_i commutes with each update in t_j, then

$$(t_1 | t_2 | \cdots | t_n) \text{ is in IPTrans}(U),$$

$$\text{eff}((t_1 | t_2 | \cdots | t_n)) = \text{eff}(t_1 \ldots t_n) \quad \text{and}^2$$

$$|(t_1 | t_2 | \cdots | t_n)| = \max\{|t_i|: 1 \leq i \leq n\}.$$

An in-line transaction t' that is equivalent to the parallel transaction t given as input can be obtained by assigning to each vertex in PTG(t) its depth and outputing $(e_{11} | e_{12} | \cdots | e_{1n_1}); \cdots (e_{i1} | \cdots | e_{in_i})$, where i is the length of t, and $\{e_{k1}, e_{k2}, \ldots, e_{kn_k}\}$ is the set of vertices of depth k, $1 \leq k \leq i$. We refer to this procedure as "Algorithm IN-LINE". It is easy to verify that such an algorithm runs in polynomial time. It is also clear that the length of t' is equal to the length of the input parallel transaction t.

Remark 3.5. Although the output of Algorithm IN-LINE is equivalent to the input parallel transaction and has equal length, it is clear that some loss of information occurs when a parallel transaction is represented by a corresponding in-line transaction produced by the algorithm. For example, consider the transaction t whose parallel transaction graph is $(\{e_1, e_2, e_3\}, \{(e_1, e_3)\})$. The in-line parallel transaction corresponding to t produced by Algorithm IN-LINE is $t' = (e_1 | e_2); e_3$. In the transition from t to t' the information that e_2 and e_3 can be performed concurrently is lost. Note that, in this case, the complete information can be captured by a different in-line transaction: $(e_1; e_3 | e_2)$. However, there are parallel transactions for which there is no in-line transaction capturing precisely the same information. For instance, consider the parallel transaction graph $G = (\{e_1, e_2, e_3, e_4\}, \{(e_1, e_3), (e_1, e_4), (e_2, e_4)\})$. It is easily seen that, in every in-line transaction consistent with G, either e_3 precedes e_4, or e_2 precedes e_3. It is straightforward to modify Algorithm IN-LINE so that the output in-line transaction captures the same information as the input, if such an in-line transaction exists. We do not do this because the information lost when applying Algorithm IN-LINE does not affect the length of the output, which we use here as a measure of parallelism. The additional information may become relevant if a more refined measure of parallelism is used.

Given a sequential, or, more generally, a parallel transaction, it is desirable to find an equivalent parallel transaction of minimal length. This gives rise to an optimization problem that is the focus of this paper. We first present an example of a parallel transaction and its corresponding optimal parallel transaction.

[2] Note that, due to the condition required of the t_i's, $\text{eff}(t_1 \ldots t_n) = \text{eff}(t_{\sigma(1)} t_{\sigma(2)} \cdots t_{\sigma(n)})$ for every permutation σ of n.

Example 3.6. Figure 3 exhibits (a) a PTG of a transaction t, (b) the PTG of a corresponding optimal parallel transaction t^* and (c) the in-line parallel transaction that corresponds to t^*. Intuitively, t^* is obtained by "cutting" the long path in the PTG of t into segments. This is accomplished by implementing some of the updates on the long path by pairs of modifications, using the empty hyperplanes corresponding to the leaves of short paths as temporary storage.

We next present some technical concepts and results that will help understand the factors that limit parallelism within a parallel transaction. These results will then be used to evaluate the complexity of the optimization problem and, in the next section, to develop parallelization algorithms.

We use transition specifications to represent effects of transactions to be parallelized. Recall that, in the transition specification of a transaction t, edges (C_1, C_2) that end at hyperplanes inserted by t are omitted. Intuitively, this is useful since it eliminates redundant modifications while increasing the number of deleted hyperplanes that can subsequently be used as temporary storage to speed up the computation. On the other hand, increasing the number of deletions does not in general increase the length of a parallel transaction since all deletions can be performed in parallel.

We say that the hyperplane C is *deleted* by transaction t if C has no outgoing edges in the transition graph G of t. Hyperplane C is *emptied* by t if C has no ingoing edges in G. Hyperplane C_1 is *stored in* hyperplane C_2 by t if there is an edge (C_1, C_2) in G.

It is clear (see Example 3.6) that if many hyperplanes are deleted by a transaction t, then t will be likely to have an equivalent parallel transaction of a small length. Indeed, it is easy to see that if the number of hyperplanes that are deleted by a transaction t is as large as half of the total number of hyperplanes of t, and if all

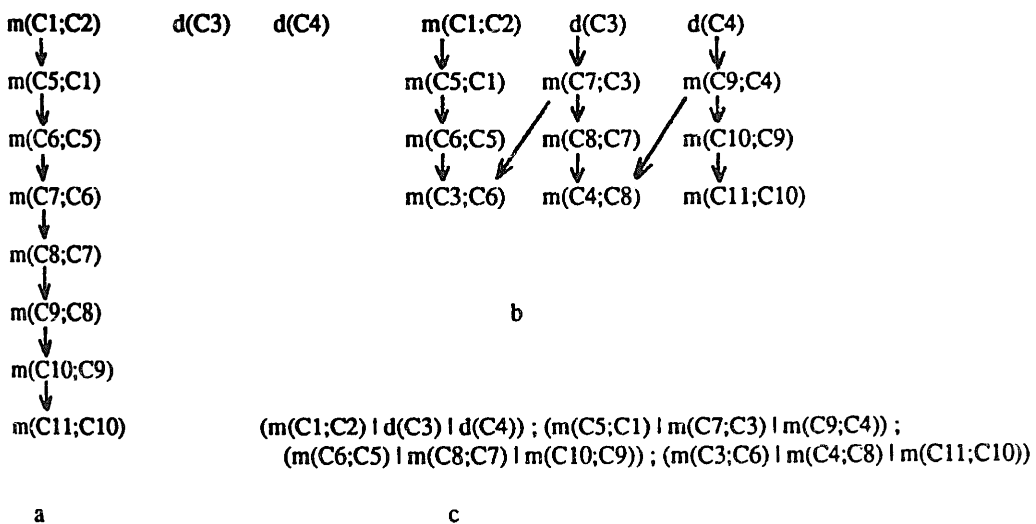

Fig. 3.

the hyperplanes have the same support, then there always exists a parallel transaction equivalent to t whose length does not exceed the constant 4 (the deletions are done in the first parallel step, all the modifications required to implement the edges in the transition graph of t are done in the second and third step, and the insertions are done in the last, fourth step). Of course, an arbitrary transaction t will in general not have such an efficient parallelization even if all its hyperplanes have the same support. We now discuss the factors that limit parallelism within a parallel transaction.

To begin with, it is clear that the number of hyperplanes that can be deleted and then used as temporary storage to speed up the computation is limited. Indeed, when a hyperplane is deleted, its contents are permanently lost. Hence only those hyperplanes that are deleted by t can be deleted. This gives rise to the following technical result (the proof is straightforward and is omitted).

Lemma 3.7. *Let $e = d(C)$ be a deletion in a parallel transaction t. Then each hyperplane that is stored in C by $t^{(\text{ptime}(e)-1)}$ is deleted by t.*

Intuitively, since all the deletions as well as all the insertions can be performed concurrently, the number of modifications will provide a measure of the essential part of "work" that needs to be done by any parallel transaction that is equivalent to some given transaction t. We focus on modifications that are likely to contribute to the effect of the transaction.

Definition 3.8. A modification $e = m(C_1; C_2)$ of a parallel transaction t is *active* if
(i) $C_1 \neq C_2$ and
(ii) if $t^{(\text{ptime}(e)-1)}$ stores some hyperplane C in C_1 then C is not deleted by t.

Note that a non-active modification can either be removed (i) or replaced by a deletion (ii). In particular, a non-active modification does not contribute towards implementing an edge in the transition graph.

The number of active modifications in a transaction t is related to the number of edges (excluding loops) in a transition graph of t, as those edges have to be implemented by modifications. Cycles require one additional modification for using the temporary storage, unless there is a "natural" storage for that cycle. This is defined as follows.

Definition 3.9. A *natural storage* for a cycle in a transition graph is a hyperplane adjacent to the cycle, with the same support as the hyperplanes in the cycle. The set of cycles in a transition graph G which have *no* natural storage is denoted Cyclesadj(G).

The following result formalizes the connection between active modifications in a parallel transaction and edges in its transition graph. It is a direct consequence of Lemma 5.1 in [1].

Lemma 3.10. *Let (G, I) be the transition specification of some parallel transaction t, where $G=(V(G), E(G))$, and M the number of active modifications in t. Then $M \geqslant |\{(C, C') \in E(G): C \neq C'\}| + |\mathrm{Cyclesadj}(G)|$.*

Modifications can not always be performed in parallel due to precedence constraints that exist between modifications. The next result follows from the observation that if two hyperplanes are stored in a common hyperplane, they can not be subsequently separated.

Lemma 3.11. *Let t be a parallel transaction, and let C_1, C_2, C be any hyperplanes such that (C_1, C) and $(C_2, C)$are edges in the transition graph of $t^{(\tau)}$ for some τ. Then either C_1 and C_2 are both deleted by t or C_1 and C_2 are stored in some common hyperplane by t.*

We are now ready to define and discuss the parallel transaction optimization problem.

Definition 3.12. An algorithm solves the *Parallel Transaction Optimization* (PTO) problem if, given as input a parallel transaction t, it produces as output an optimal parallel transaction t' such that $t' \approx t$.

The following shows that the PTO problem is intractable even in a simplified version.

Theorem 3.13. *The parallel transaction optimization problem is* NP*-complete even when the input transaction consists only of deletions and modifications, all the hyperplanes in the input transaction have the same support, and all vertices in the transition specification graph have indegrees no greater than* 1.

Proof. As customary, we prove NP completeness using the "language recognition" version of the PTO problem. We say that an algorithm solves the (language recognition) PTO problem if, given a parallel transaction t and an integer K as input, it decides whether there exists a parallel transaction t' such that $t' \approx t$ and $|t'| < K$.

It is easy to see that the language recognition PTO is in NP, since a nondeterministic Turing machine can "guess" a parallel transaction t', verify that $t' \approx t$ by comparing the corresponding transition specifications, and then verify if $|t'| < K$, all in polynomial time. The proof that the PTO is NP-hard is by reduction from 3-PARTITION.

An instance of 3-PARTITION consits of a finite set F of $3m$ elements, a bound $N \in Z^+$, and a "size" $s(a) \in Z^+$ for each $a \in F$, such that each $s(a)$ satisfies $N/4 < s(a) < N/2$ and the sum of the sizes of all elements in F is equal to mN. An algorithm solves the 3-PARTITION problem if it decides whether F can be partitioned into m disjoint sets $S_1, S_2, \ldots, S_m$ such that, for each set S_i, the sum of sizes of the elements of S_i is equal to N. It is known that the 3-PARTITION problem is

NP-complete even when all the element sizes are bounded by a polynomial function of the total number of elements (see the discussion in [4, p. 99]).

We now show that the PTO problem is NP-hard by exhibiting a polynomial-time reduction of the 3-PARTITION with element sizes bounded from above by a polynomial in the number of set elements, to the restricted version of the PTO. Given an instance of the 3-PARTITION problem, the corresponding instance (t, K) of the PTO is constructed as follows. The input parallel transaction t is actually sequential and consists of m deletions of the form $d(C1), \ldots, d(Cm)$, where $Ci = \{A = i\}$, followed by a sequence of $3m$ syntactic cycles, one for each element of the set F. The syntactic cycle for each element a_i of F consists of a sequence of $M = s(a_i)$ modifications of the form $m(C_{i1}; C_1)$ $m(C_{i2}; C_{i1})$ $\ldots m(C_{iM-1}; C_{iM-2})$ $m(C_1; C_{iM-1})$, where $C_{ij} = \{A = iN + m + j\}$ (this guarantees that the C_{ij} are incompatible). Note that each such syntactic cycle implements a cycle of length $s(a_i) - 1$, using C_1 as temporary storage. The constant K is set to $N + 1$.

It is clear that this transformation can be done in time that is polynomial in the size of the 3-PARTITION instance. It is left to show that the instance of the PTO has an affirmative solution if and only if the instance of 3-PARTITION has an affirmative solution. The idea of the proof is that the PTO has an affirmative solution iff the $3m$ syntactic cycles can be equally distributed among the m empty hyperplanes C_i in groups of three, such that the three syntactic cycles assigned to C_i use C_i as temporary storage. Clearly, this can be done iff the instance of 3-PARTITION has a solution. We now sketch the proof. Notice that the transition graph G of the transaction t consists only of cycles and vertices with no incident edges. Consider an arbitrary parallel transaction t_1 that is equivalent to t. By Lemma 3.10, t_1 contains at least mN active modifications. It is easy to verify, using the Lemmas 3.7 and 3.11 and the fact that all the indegrees in the transition graph G of t are at most 1, that for any τ there are at most m active modifications e in t_1 such that $\text{ptime}(e) = \tau$. There are exactly k active modifications in the parallel step τ only if at least k hyperplanes are empty at the beginning of τ. Since no hyperplanes are empty prior to the first parallel step, no modifications can be performed in the first parallel step. Hence if $|t_1| = N + 1$ then the first parallel step of t_1 consists of m deletions, and then mN modifications are performed in the following N parallel steps, m of them at a time in parallel.

It is easy to see that this is possible if and only if the syntactic cycles can be partitioned into m subsets of three syntactic cycles each, such that the number of modifications in each subset is equal to N. (The three syntactic cycles in the same subset must use the same hyperplane C_i as temporary storage.) This is equivalent to existence of a solution to the 3-PARTITION instance. The idea of the proof is illustrated in Fig. 4. □

Although the above NP-completeness proof uses an instance of the PTO corresponding to a cyclic transaction, the NP-completeness result holds even if the input

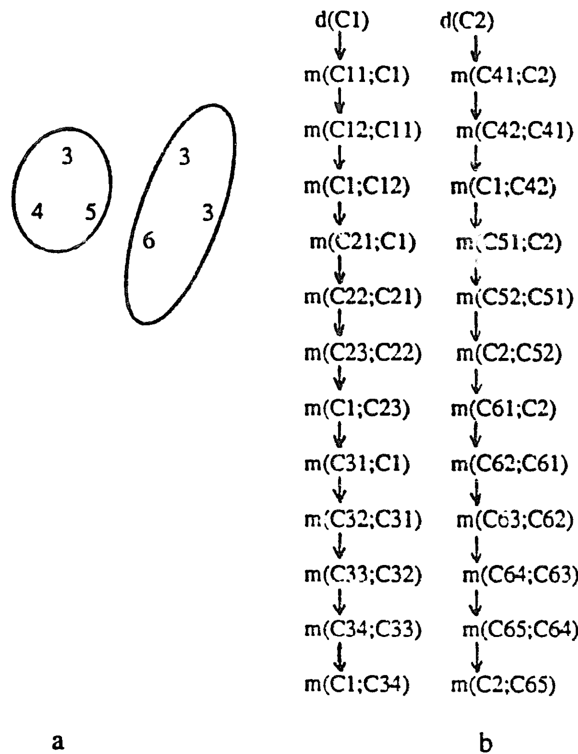

Fig. 4. A solution to an instance of 3-PARTITION (a), and the solution to the corresponding instance of PTO (b).

instances are restricted to contain only acyclic transactions. A proof similar to the above exists for this version of the PTO problem.

4. Approximate optimization algorithms

Since the PTO problem can be solved in polynomial time only in the highly unlikely case that P = NP, practically feasible approximation algorithms are needed for solving the problem. In this section we consider several such algorithms. We first introduce polynomial-time algorithms for two special cases, and show that they approximate the exact solutions within an absolute constant (1 and 2, respectively). Finally, we use these algorithms to develop a polynomial-time algorithm for the general case, which approximates the exact solution within a constant factor.

The two special cases we consider involve transactions using only hyperplanes with the same support. These cases correspond intuitively to the "box redistribution" problem described in the introduction.

We now exhibit our approximation algorithm for the restricted PTO where the transition graph of the input transaction contains hyperplanes with the same support and indegree at most 1. The algorithm first generates a parallel transaction graph that corresponds directly to the transition graph of the input transaction. Because of the indegree restriction, the parallel transaction graph consists only of paths and

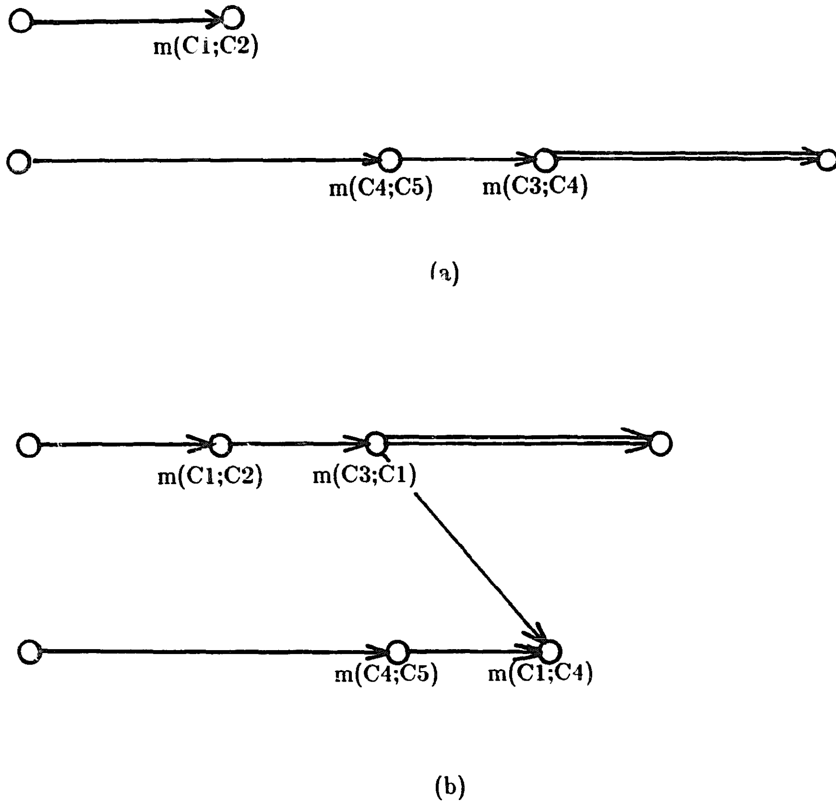

Fig. 5. (a) A short path and a long path. (b) The outcome of Procedure CUT-AND-PASTE.

isolated vertices. At the heart of the algorithm is procedure CUT-AND-PASTE, which equalizes the paths in the PTG by "cutting" the long paths and "pasting" them to short paths. We discuss this procedure informally, then exhibit the complete algorithm. Figure 5(a) exhibits a "short" path ending in $e_1 = m(C1;\ C2)$ and a long path where the updates $m(C4;\ C5)$ and $m(C3;\ C4)$ are adjacent and occur "after" e_1 (with respect to parallel time). In the long path, the updates following $m(C3;\ C4)$ (represented by the double arrow) are delayed in order for $C3$ to become empty. However, hyperplane $C1$ has been freed by now in the short path, and can be used to temporarily store $C3$ and thus allow execution of the double-arrow path. Thus, the double-arrow path is "cut" from the long path and "pasted" onto the short path. Eventually, $C3$ is forced to the intended destination (C4) using the modification $m(C_1;\ C4)$. The resulting PTG is represented in Fig. 5(b). Note the decrease in the length of the parallel transaction. The complete algorithm consists essentially of repeated applications of CUT-AND-PASTE until the paths are equalized as far as possible. The complete algorithm follows.

Algorithm FIRST APPROXIMATION

Input: A transaction t whose transition specification is $(G, \emptyset)$ such that all the hyperplanes in G have the same support and indegree no larger than 1.

Output: A PTG P of parallel transaction t' such that $t' \approx t$.

1. Compute the transition specification $(G, \emptyset)$ of t.
2. Set $P = \text{COMPUTE-PTG}(G)$.
3. Set $K = \lceil |P|/q \rceil + 1$, where $|P|$ is the number of updates in P and q is the number of connected components in P.
4. Repeat while there exists an update e in P such that ptime$(e) = K$ and outdegree of e is not equal to zero.
   begin
   CUT-AND-PASTE(e);
   end
6. Output P.

Procedure COMPUTE-PTG(G)

1. Set $P = (\emptyset, \emptyset)$.
2. For each hyperplane C in G that has no outgoing edges add a deletion $d(C)$ to P.
3. For each directed path in G that does not belong to a cycle add a corresponding sequence of modifications (directed path) to P, and an edge from the first modification to the appropriate deletion.
4. For each cycle in G add to P a syntactic cycle S that implements the cycle by using an arbitrary empty hyperplane C as temporary storage. Add an edge from the update that empties C to the first update of S.
5. Return P.

Procedure CUT-AND-PASTE(e)

1. Find an update e_1 in P with outdegree equal to zero such that ptime$(e_1) \leq K - 2$. Then either $e_1 = d(C1)$ or $e_1 = m(C1; C2)$ for some hyperplanes $C1, C2$. Let $e = m(C3; C4)$, for some hyperplanes $C3$ and $C4$.
2. Add a new vertex $e_2 = m(C1, C4)$ to P.
3. Replace the unique edge from the ancestor of e to e by an edge from the same ancestor to e_2.
4. Replace e by $e_3 = m(C3, C1)$ in both the vertex set and in the edge set of P.
5. Add edges (e_3, e_2) and (e_1, e_3) to P.

Remark 4.1. The empty hyperplane used for temporary storage in Step 4 of COMPUTE-PTG exists by Lemma 5.5 in [1].

Theorem 4.2. (i) *Algorithm FIRST APPROXIMATION runs in polynomial time.*

(ii) *Let t be a transaction that consists of only deletions and modifications, such that all the hyperplanes in t have the same support and the vertices in the transition graph of t have the indegrees no larger than 1. Let t' be the parallel transaction produced*

by FIRST APPROXIMATION under input t. Then $|t'| \leq |t^*|+1$, where t^* is the optimal parallel transaction corresponding to t.

Proof. We prove (ii) in two steps:

(*) $|t'| \leq K$;

(**) $|t^*| \geq K-1$.

Claim (i) will be proved within Step (*).

Proof of (*): Recall that $|t'| \leq K$ is the termination condition for the algorithm, and that otherwise CUT-AND-PASTE (CAP) is applied. We now show the following:

(a) Whenever there exists an update whose parallel time exceeds K, it is possible to apply CAP.

(b) CAP can be applied to P at most q times, where q is the number of connected components in P after Step 2 of the algorithm.

Clearly, (a) and (b) imply (*). Notice also that (b) implies (i), since it is easy to see that CAP as well as Steps 1-3 of FIRST APPROXIMATION require only polynomial time.

In order to prove (a) and (b) we define a partition of the updates in P into equivalence classes as follows. Before CAP is applied to P, each equivalence class consists of one connected component of P (a path or an isolated vertex). When CAP is applied on an update e, then e and its descendents are "cut off" and "pasted" to another update e_1 (since P is a directed acyclic graph, the set of descendents is well defined). At that point we change the partition so that e and its descendents are moved from their equivalence class R to the equivalence class to which e_1 belongs (e is at this time changed into e_3). The newly created update e_2 is added to R.

Notice that at any time each equivalence class R is a totally ordered subset of P, and that for each update e in R, the position of e in that total order corresponds to ptime(e). The unique update in R that has no outgoing edges in P (or, equivalently, that has the largest parallel time in R) will be called "the last" in R and denoted by $L(R)$.

Note that, if a certain equivalence class R contains exactly K updates at some time during the execution of the algorithm, CAP can not be subsequently applied to any update in R. Indeed, R neither contains an update whose outdegree is greater than 0 and whose parallel time is K, nor an update whose outdegree is 0 and whose parallel time is no larger than $K-2$. Recall from the algorithm that, when CAP is applied to an update e, the size of the equivalence class that previously contained e is reduced to K. Hence we have proved (b).

It follows by the above argument that every equivalence class of size K has at most one update created in Step 2 of CAP. Let P_n be the subgraph of P obtained by removing all the vertices that belong to equivalence classes of size K after n executions of CAP. It is easy to see by using the observation from the beginning of this paragraph that $K-1$ is no smaller than the mean value of ptime($L(P_i)$) over

all P_i contained in P_n. erefore if there is an update e in P_n whose parallel time exceeds K, then there ...ust exist another update e_1 such that e_1 is last in its equivalence class and ptime$(e_1) \leq K-2$. Hence, CAP can be applied. This completes the proof of (a) and of Claim (*).

Proof of (**): Consider an arbitrary active modification $e = m(C1; C2)$ in the optimal parallel transaction t^*. Since all the indegrees in the transition specification graph of the input transaction are no larger than 1, it follows by Lemma 3.11 that if ptime$(e) = \tau$ then $C2$ is empty at the beginning of parallel step τ and there is no active modification $m(C3, C2)$, for some hyperplane $C3$, that can be performed concurrently with e. Hence, at any time τ, t^* can perform at most as many modifications as there are emptied hyperplanes after parallel step $\tau - 1$. By Lemmas 3.7 and 3.11 there are at most as many empty hyperplanes in t^* at any time as there are zero-outdegree hyperplanes in G. By construction, the number of zero-outdegree hyperplanes in G equals the number q of connected components of P (Step 2). Thus, t^* can perform at most q active modifications in each parallel step. On the other hand, from Lemma 3.10 it follows that any parallel transaction equivalent to t must perform at least as many active modifications as in P, that is, $|P| - q$. Additionally, only deletions can be performed in the first parallel step of t^*. Thus,

$$|t^*| \geq 1 + \left\lceil \frac{|P| - q}{q} \right\rceil = K - 1. \qquad \square$$

The above result suggests that there may exist good approximation algorithms even for more general PTO problems. We now show that this is indeed the case. The following algorithm computes a nearly optimal parallel transaction for any transaction in which all the hyperplanes have the same support.

We first explain the intuitive idea behind the algorithm, and then define the algorithm in detail. It is clear that, if a certain hyperplane has several ingoing edges in the transition graph, then the modifications that correspond to those edges can be executed in parallel. Then not only multiple active modifications are done at a time, but also empty hyperplanes are created, which can subsequently be used as temporary storages to speed up the computation.

The algorithm has two phases. The first phase (Steps 1–10) divides the hyperplanes into equivalence classes of hyperplanes that are stored into the same hyperplane by the input transaction t. It then generates the first parallel step of the output parallel transaction in which all the hyperplanes that are deleted by t are deleted by deletions, and all the hyperplanes for each equivalence class are joined together into one member of the class by modifications.

The problem that remains to be solved after the first phase is exactly the one solved by Algorithm FIRST APPROXIMATION, hence FIRST APPROXIMATION is called to finish the job. All insertions are performed in the last parallel step. The algorithm is defined in detail below, and illustrated by an example in Fig. 6.

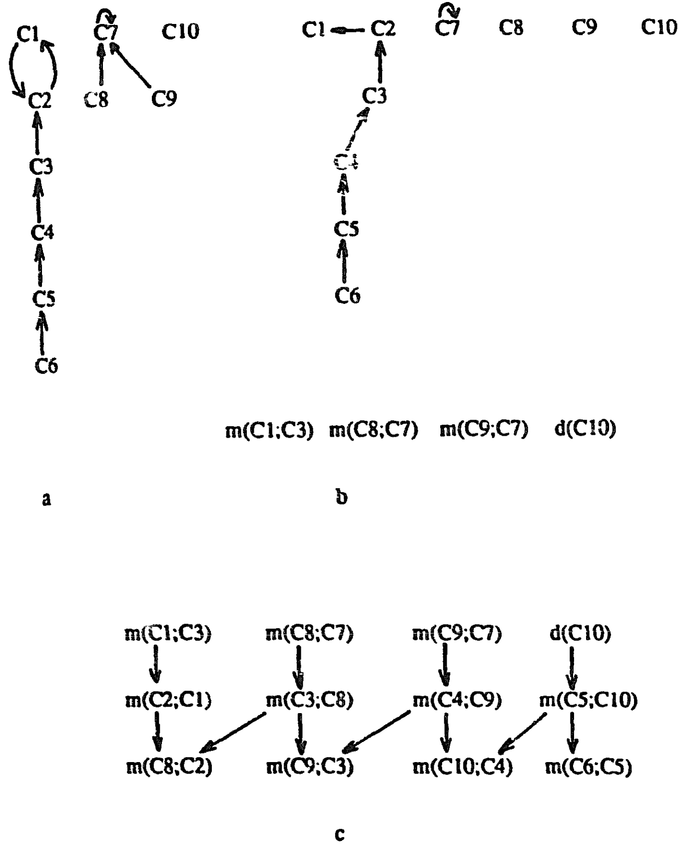

Fig. 6. (a) Transition graph G of the input transaction, (b) G and P before Step 11, and (c) the output PTG.

Algorithm SECOND APPROXIMATION

Input: A transaction t such that all the hyperplanes in t have the same support.

Output: A PTG P of a parallel transaction t' such that $t' \approx t$.

1. Compute the transition specification (G, I) of t.
2. Set $P = (\emptyset, \emptyset)$.
3. Repeat for each vertex C in G that has no outgoing edges:
   4. Add a vertex $d(C)$ to P.
5. Divide the vertices of G that have outdegrees greater than 0 into equivalence classes such that a vertex C_j is in the equivalence class $S(C_i)$ if (C_j, C_i) is an edge in G.
6. Repeat for each equivalence class $S(C_i)$
   begin
   if $S(C_i)$ has more than one element then
   begin
   7. Set $C_i^0 = \text{SELECT-ELEMENT}(S(C_i))$.
   8. Repeat for each $C_j \in S(C_i)$ such that $C_j \neq C_i^0$:
      9. Add vertex $m(C_j, C_i^0)$ to P.

10. Remove edge (C_j, C_i) from G.
end
end
11. Run FIRST APPROXIMATION on G and P omitting Steps 1 and 2.
12. Repeat for each vertex e in P created in Steps 4 and 9:
13. Repeat for each vertex e_1 created in Step 11:
if e_1 has no ingoing edges in P then
14. Add edge (e, e_1) to P.
13. Repeat for each hyperplane C such that $C \in I$
begin
14. Add vertex $e = i(C)$ to P.
15. Repeat for each vertex $e_1 \neq e$ in P:
if e_1 has no outgoing edges in P then
14. Add edge (e_1, e) to P.
end
16. Output P.

Procedure SELECT-ELEMENT($S(C_i)$)
if $C_i \in S(C_i)$ then
1. Set $C = C_i$.
else if C_i belongs to a cycle in G and there exists
a $C_j \in S(C_i)$ that is not on the cycle and that has the same support as C_i then
2. Set $C = C_j$.
else
3. Set $C = C_k$ where C_k is an arbitrary element of $S(C_i)$.
4. Return (C).

Theorem 4.3. (i) *Algorithm SECOND APPROXIMATION is polynomial-time in the number of updates of the input.*

(ii) *Let t' be the parallel transaction produced by SECOND APPROXIMATION on input t, let t^* be an optimal parallel transaction equivalent to t. Then* $|t'| \leq |t^*| + 2$. *If t has no insertions then* $|t'| \leq |t^*| + 1$.

Proof. It is easy to verify that SECOND APPROXIMATION is a polynomial-time algorithm. Consider (ii). Suppose first that t contains no insertions. By construction, $t' = t'^{(1)} t_0'^{(>1)}$, where $t_0'^{(>1)}$ is the output of FIRST APPROXIMATION on any parallel transaction whose transition graph is $G_0' = \mathrm{TG}(t) - \{\langle C_j, C_i \rangle \mid m(C_j, C_i^0)$ is in $t'^{(1)}\}$. Clearly, all nodes in G_0' have indegree at most 1. Also, note that the only cycles in G_0' are cycles in Cyclesadj(TG(t)) (cycles with no hyperplane adjacent to them). Next, consider $t^{*(1)}$ (the first parallel step of t^*). By Lemmas 3.7 and 3.11, $t^{*(1)}$ consists just of deletions or modifications of the form $m(C1, C2)$ where $C1$ and $C2$ are equivalent. Clearly, we may assume without loss of generality that PTG(t^*) contains no redundant edges (i.e., edges whose removal does not change

the effect of the transaction). Then $t^* = t^{*(1)} t_0^{*(>1)}$, where t_0^* is an optimal parallel transaction whose transition graph is $G_0^* = \mathrm{TG}(t) - \{\langle C1, C3\rangle \mid m(C1, C2)$ is in $t^{*(1)}$ and $\langle C1, C3\rangle$ is in $\mathrm{TG}(t)\}$ (otherwise, it is easy to see that t^* is not optimal). Next, note that the number of non-loop edges in G_0' is at most the number of non-loop edges in G_0^*, and

$$|\mathrm{Cycles}(G_0^*)| = |\mathrm{Cyclesadj}(G_0^*)| \geq |\mathrm{Cyclesadj}(G_0')| = |\mathrm{Cycles}(G_0')| = |\mathrm{Cyclesadj}(\mathrm{TG}(t))|.$$

From Lemma 3.10 it follows that $t_0^{*(>1)}$ has at least as many active modifications as $t_0'^{(>1)}$. By an argument similar to that of the proof of Theorem 4.2, $|t_0'^{(>1)}| \leq |t_0^{*(>1)}| + 1$. Hence, $|t'| \leq |t^*| + 1$. This concludes the proof of (ii) in the case when t does not contain insertions. Suppose now that t contains insertions. Clearly, if t_1' and t_1^* are obtained by eliminating the insertions from t' and t^*, then $|t_1'| \leq |t_1^*| + 1$, by the above. Next, note that t' is obtained from t_1' by adding a parallel step where all insertions are performed. Thus, $|t'| = |t_1'| + 1 \leq |t_1^*| + 2 \leq |t^*| + 2$. This completes the proof of (ii). □

So far we have considered a restricted version of the PTO problem, where all hyperplanes have the same support. We next consider the general case, where hyperplanes may have different supports. In that case we can divide the hyperplanes into equivalence classes of hyperplanes with the same support. Intuitively, since the number of different equivalence classes is fixed and relatively small (assuming a fixed schema), most of the computation still occurs within individual equivalence classes. This suggests that a reasonable approximation algorithm can be obtained simply by applying the SECOND APPROXIMATION algorithm to each equivalence class independently. This gives rise to the following algorithm.

Algorithm THIRD APPROXIMATION

Input: A transaction t.

Output: A PTG P of a parallel transaction t', $t' \approx t$.

1. Set $P = (\emptyset, \emptyset)$.
2. Compute the transition specification (G, I) of t.
3. Divide the vertices in G into equivalence classes of hyperplanes with equal support.
4. Repeat for each equivalence class S:
   5. Apply SECOND APPROXIMATION to P and the subgraph of G spanned by S, omitting Steps 1 and 2.
6. Repeat for each equivalence class S:
   7. Repeat for each edge (C_i, C_j) in G such that $C_i \in S$, $C_j \in S'$ and $S \neq S'$:
      begin
         8. Replace the vertex $d(C_i)$ in the subgraph of P that corresponds to S by $e = m(C_i; C_j)$.

if there exists a vertex e' in the subgraph of P
corresponding to S' such that $e' = d(C_j)$
or $e' = m(C_j; C_k)$ for some
hyperplane C_k and e' has no outgoing
edges in P then
9. Add edge (e', e) to P.
end
10. Output P.

Theorem 4.4. (i) *Algorithm THIRD APPROXIMATION runs in polynomial time.*
(ii) *Let t' be the transaction produced by THIRD APPROXIMATION under input t and let t^* be the optimal parallel transaction that corresponds to t. Then $|t'| \leq K|t^*| + K + 1$, where K is the number of attributes in t.*

Proof. It is easy to see that THIRD APPROXIMATION is a polynomial-time algorithm. To prove (ii), it is first shown that, intuitively, any optimal t^* must also be optimal within each equivalence class of updates using the same support. Next, consider any sequence $S = e_1, e_2, \ldots, e_m$ of updates in the PTG of t' such that $e_i < e_{i+1}$ and the number of modifications in S is equal to $|t'|$. Since the existence of $m(C1; C2)$ implies support $(C1) \subseteq$ support $(C2)$, the hyperplanes in S belong to at most K distinct equivalence classes of hyperplanes with the same support. Consider the segments $s_1, s_2, \ldots, s_m$ $(m \leq K)$ of S consisting of updates within the same equivalence class, in the order in which they appear in S. The length of segment s_1 is at most that of the output of SECOND APPROXIMATION on the equivalence class of s_1. The length of each s_i, $i > 1$, is at most the length of the output of SECOND APPROXIMATION on the equivalence class of s_i, minus 1 (it is easy to see that updates from the first parallel step produced by SECOND APPROXIMATION on the equivalence class of s_i, $i > 1$, do not occur in the path considered). Thus, $|s_1| \leq |t^*| + 1$ and $|s_i| \leq |t^*|$. Thus, the maximum number of updates in S involving hyperplanes within the same equivalence class is $K|t^*| + 1$. Additionally, S may contain at most $K - 1$ modifications $m(C_1; C_2)$ where support$(C_1) \subset$ support(C_2). Thus, at most $K - 1$ modifications in the sequence S involve hyperplanes from different equivalence classes. Finally, S contains at most one insertion. Hence, the length of S is at most $K|t^*| + K + 1$. □

In practice the number K of attributes in a relational database is a small constant, hence the worst case behavior of THIRD APPROXIMATION will differ at most by a constant factor from the optimum. However, it is clear that the worst case occurs only in very unusual circumstances, and that the practical or expected behavior of the algorithm will be much better.

Remark 4.5. The behavior of the algorithm in practice can be improved by using some global information about the PTG of the input, rather than just information

local to each equivalence class of hyperplanes. Specifically, a modified version of the procedure FIRST APPROXIMATION can be used, which takes into account information about the *total* length of chains in the PTG, rather than just the length of chains within a given equivalence class. However, this approach does not improve the worst-case behavior of the algorithm. Indeed, we conjecture that approximating the solution of the PTO in the general case within an absolute constant is NP-complete.

Acknowledgment

The authors wish to thank Richard Hull for useful discussions on this paper.

References

[1] S. Abiteboul and V. Vianu, Equivalence and optimization of relational transactions, *J. ACM* **35** (1988) 70–120.
[2] P. Bernstein and N. Goodman, Concurrency control in distributed database systems, *Comput. Surv.* **13** (1981) 185–222.
[3] C.J. Date, *A Guide to DB2* (Addison-Wesley, Reading, MA, 1984).
[4] M.R. Garey and D.S. Johnson, *Computers and Intractability* (Freeman, San Francisco, CA, 1979).
[5] A. Karabeg, D. Karabeg, K. Papakonstantinou and V. Vianu, Axiomatization and simplification rules for relational transactions, in: *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems* (1987) 254-259.
[6] D. Maier, *The Theory of Relational Databases* (Computer Science Press, Rockville, MD, 1983).
[7] C.H. Papadimitriou, B.A. Bernstein and J.B. Rothnie, Computational problems related to database concurrency control, in: *Proc. Conf. on Theoretical Computer Science*, Waterloo, Ontario, Canada (1977) 275–282.
[8] J. Ullman, *Principles of Database Systems* (Computer Science Press, Rockville, MD, 1980).