ON THE CORRECT SCHEDULING
OF TRANSACTION SYSTEMS
FOR HIGHLY PARALLEL DATA BASE MACHINES

Ravindran Krishnamurthy
Umeshwar Dayal

Department of Computer Sciences,
The University of Texas at Austin.

TR-170            Feb., 1981
ABSTRACT

This paper proposes a two-step technique for producing correct and highly parallel schedules for MIMD database machines. A parallel program schema model for transaction systems is presented. The concept of correct (i.e., serializable) executions existing in concurrency control theory for the sequential model is extended to this parallel model. The model is used to derive minimally constrained schemas for optimal scheduling. This constitutes the first step of the two-step technique. In the second step, the transactions are partially interpreted to enhance parallelism.
# Table of Contents

1. INTRODUCTION .......................... 1
2. MODEL OF A TRANSACTION SYSTEM ...... 4
   2.1 Computation sequences and their properties 5
   2.2 Semantic equivalence ................. 6
   2.3 Syntactic equivalence ............... 7
   2.4 Relationship between syntactic and semantic equivalence 8
3. MODEL OF EXECUTION .................. 9
   3.1 Execution and correctness criterion 9
   3.2 Schema for a transaction system .... 10
4. TRANSACTION MODIFICATION .......... 13
   4.1 Environment ........................ 14
   4.2 Transformation of an operation pair in series 17
   4.3 Generalized transformation .......... 19
   4.4 Algorithm for transformation ...... 21
   4.5 Implementation considerations ...... 22
5. CONCLUSION ......................... 23
6. Appendix: PROOFS ..................... 26
1. INTRODUCTION

Parallel (i.e. multiple instruction-stream, multiple data-stream) data base machines such as DIRECT, have been proposed with the objective of enhancing processor utilization and achieving high transaction throughput [DeW78]. Improving processor utilization requires the efficient scheduling of transactions (for parallel execution) on available processors. But a parallel execution of transactions requiring access to shared data, can lead to race conditions and inconsistent states of the data base, unless some synchronization (concurrency control) mechanism is used[EGLT76]. This underscores the importance of both scheduling and synchronization to achieve correct (i.e. serializable) and maximally parallel executions.

The DIRECT machine uses locking in the front-end as its synchronization mechanism. However, this seems unduly restrictive and may even be prohibitive for very high throughput machines. No other synchronization mechanisms for parallel data base machines have been proposed in the literature. On the other hand, there is a wealth of concurrency control theory that has been developed for centralized and distributed data base systems, assuming a sequential model of execution [BSW79, Papa79, BSR80]. In this paper, we extend this theory to a parallel model of execution and use it to derive the minimal precedence constraints for scheduling. Although this paper develops a theoretical model of concurrency control for parallel processing environments in general, we believe that it can be directly applied to any MIMD machine such as DIRECT.

Assume that any data base system can be modelled as shown in figure 1-1. Users submit transactions (each consisting of several steps) to the system. A set of these transactions, called a transaction system, is input to the scheduler, which examines the transactions for potential conflicts and imposes
Fig. 1-1: Model of a data base management system.
a partial ordering on the transaction steps. The scheduler outputs a
precedence graph (called a schema) corresponding to this partial ordering, for
execution on the data base machine. The transaction system is then executed
by the machine using some low-level processor allocation policy. We are
concerned here with the problem of representing a transaction system using
minimal precedence constraints so that any execution satisfying these
constraints is correct. An execution is correct if and only if it is
serializable, i.e. equivalent to some serial execution. In the existing
concurrency control theory for the sequential model of execution, both in
centralized and distributed systems, a transaction is modelled as a sequence
of operations. The execution of a transaction system in the centralized case
is also modelled as a sequence of operations, perhaps with steps of different
transactions interleaved in it. This sequence is called a history. A given
history is serializable if and only if it is computationally equivalent to a
serial history, which defines a total ordering on the transactions in the
system [BSW79, Papa79]. In the distributed case, an execution is modelled as
a set of histories, one for each site. Since there is no global clock in the
system, no ordering can be imposed on the operations executing at different
sites. Consequently, distributed systems use protocols to ensure
serializability [BSR80]. A distributed execution is serializable if and only
if the history at each site is serializable and the equivalent serial
histories at all sites impose the same total ordering of transactions.

A sequential ordering of transaction steps does not exploit the full power
of a parallel machine. Gouda [Gou80] has extended this sequential model of a
transaction and a transaction system to a directed acyclic graph (DAG). The
edges of the DAG impose precedence constraints on conflicting pairs of
transaction steps. In the first part of this paper, we formalise this model using parallel program schemata theory [Kell73] and extend the notion of serialisability to this model. We then derive a minimal set of precedence constraints for a transaction system to satisfy this correctness criterion.

Most proposed concurrency control mechanisms use uninterpreted transactions, i.e. they use only syntactic information such as the read and write sets of the operations in the transaction steps. It has been shown in [KP79] that greater concurrency can be achieved if semantic information is utilized in addition to purely syntactic information. In the second part of this paper we show how to exploit semantic information to modify transaction steps and thus increase parallelism even further.

Thus, this paper proposes a two-step technique for producing correct and highly parallel schedules: first, construct a schema with minimal precedence constraints; then, modify it using semantic information to increase parallelism.

Section 2 presents a parallel program schema model of a transaction and defines the notion of syntactic and semantic equivalence of schemata. A model of execution and a correctness criterion are presented in section 3, together with a method for syntactically constructing a correct execution schedule with minimum precedence constraints. Section 4 extends the theory to incorporate semantics.
2. MODEL OF A TRANSACTION SYSTEM

In this section we adopt the general model of parallel program schemata developed in [Kel73] to database transaction systems. The database is viewed as a shared memory $M_0$ and a local memory $M_i$ for each user $i$. Each memory is viewed to be a countably infinite set of cells, and $M$ is defined to be the union of all the memories. Transactions and transaction systems are modelled as schemata, defined over a set of operations. Each transaction step is an operation (henceforth, we use these two terms synonymously). Associated with each operation $s_i$ are:

1. a unique symbol $\text{ter}_i$ called the terminator of $s_i$; and

2. two finite sets $D_{s_i} \subseteq M$, the domain of $s_i$ and $R(s_i) \subseteq M$, the range of $s_i$.

Intuitively, each operation reads the elements of its domain, performs some computation on them, and writes into the elements of its range. The terminator is an atomic event (i.e., indivisible and mutually exclusive) and represents the atomic commitment of the operation. We distinguish between read-only and other read-write operations. A read-only operation, $s_i = r_i$, reads from the main memory into the corresponding cells of the local memory, whereas a read-write operation, $s_i = w_i$, writes into the shared memory. A computation operation, $s_i = f_i$, calculates a new value from a set of values in the local memory. Informally, we shall use $R(\text{ter}_i)$ and $D(\text{ter}_i)$ are defined to mean $R(s_i)$ and $D(s_i)$ respectively. In addition for any $C \subseteq S$ define,

$$Q(C) = \{\text{ter}_i | s_i \in C\}, \text{ and informally } Q = Q(S),$$

$$Q = \{x | x \text{ is a permutation of } Q\}.$$

The domain and range of an operation provide purely syntactic information. To express the semantics of the operation (i.e., the actual computation performed by it), an interpretation is required. An interpretation for an
operation set defines a universe and an initial assignment of values to the memory cells, and for each operation \( s_i \), a set of functions which map \( D(s_i) \) into \( R(s_i) \).

As we are interested initially in developing a model based purely on syntactic information, a schema is defined independent of any specific interpretation. In section 4 we extend this concept to incorporate partially interpreted operations.

Definition 2-1: Let \( S \) be a set of operations. A parallel program schema, (or simply schema) over \( S \) is a directed acyclic graph \( G=(S,E) \), where the edges in \( E \) represent precedence constraints on the operations in \( S \).

The schema specifies those operations which may be executed concurrently. An operation can be enabled for execution only after all its predecessors in \( G \) have terminated. The statement 'G is a schema over S' will be abbreviated \( G \in \text{SCH}(S) \).

2.1 Computation sequences and their properties

Based on the control information specified by a schema \( G \in \text{SCH}(S) \), we identify two important properties of \( G \), viz. equivalence and determinacy. To do this, we must characterise the set of allowable computation sequences of a given schema \( G \).

A computation sequences (or comp) for \( G \) is a string \( z \in Q \) such that \( z \) is a topological sort sequence of the partial order defined by the schema \( G \). Intuitively, a comp for a schema \( G \) is a sequence of terminations representing a permissible order of committing the effects of the operations. The set of computation sequences for \( G \) is denoted COMP(G).

In general, we can characterise the properties of a schema \( G \in \text{SCH}(S) \) based on the properties of the set of comps allowed by \( G \). Given an equivalence relation \( E \) on \( Q \), \( G \in \text{SCH}(S) \) is called \( E \)-determinate if
\[ \forall x, y \in \text{COMP}(G) \quad [x = y \ (E)] \]

that is, if \( \text{COMP}(G) \) is contained in a single \( E \)-equivalence class. Intuitively, if we define an equivalence relation \( E = \) 'produces the same result under a given interpretation', then \( E \)-determinacy ensures that all comps produce the same result under a given interpretation of the operations in the schema.

Given an equivalence relation \( E \) on \( Q \) and two schemata \( G_1, G_2 \in \text{SCH}(S) \), \( G_1 \) and \( G_2 \) are said to be \( E \)-equivalent, (written \( G_1 \equiv G_2 \)), iff,

\[ \forall x \in \text{COMP}(G_1) \exists y \in \text{COMP}(G_2) \ [x = y \ (E)] \ \& \ \\ |\{E(x); x \in \text{COMP}(G_1)\}| = |\{E(y); y \in \text{COMP}(G_2)\}| \]

where \( E(x) \) denotes the equivalence class of \( x \). It is to be noted here that if the schemata \( G_1 \) and \( G_2 \) are determinate, then each has only one equivalence class. Therefore, equivalence of the two schemata implies that their COMPs represent the same equivalence classes. Intuitively, both the schemata represent sets of computations that are equivalent under \( E \).

These properties of a schema have been defined for any arbitrary equivalence relation. Two particular equivalence relations are of interest. These two relations, \( E_g \) and \( E_h \), which represent syntactic and semantic equivalence, are defined below.

2.2 Semantic equivalence

Intuitively, we expect two schemata \( G_1 \) and \( G_2 \) to be equivalent if for any interpretation, they behave identically; i.e., given any comp \( x \), of one schema there is a comp \( y \), of the other schema such that for every cell \( m \) in \( M \), \( x \) and \( y \) assign the same value to \( m \). Since this must be true for all interpretations, we can use the notion of an Herbrand interpretation [Mann74]. This is defined formally as follows.
Definition 2.2: Given $G \in \text{SCH}(S)$, and a string $x \in \text{COMP}(G)$ define the Herbrand interpretation, (denoted $H_m(x)$), for the $m$th cell where $m \in M$, as follows:

1) $H_m(\lambda) = \lambda$ (where $\lambda$ is the null string)
2) $\forall x \in \text{COMP}(G), \forall y < x$, and $\forall \text{ter}_1 \in G$,

$$H_m(y) = \begin{cases} H_m(y) & \text{if } m \not\in R(s_i) \\ <G(y \text{ ter}_i) & \text{if } m \in R(s_i) \end{cases}$$

$$H_m(\text{ter}_i) = \begin{cases} F \mid H_{l_1}(y), H_{l_2}(y), \ldots, H_{l_j}(y) & \text{where each } l_j \in D(s_i) \text{ and } j > k \Rightarrow l_j > l_k \text{ if } m \in R(s_i) \\ <G(y \text{ ter}_i) & \text{if } m \not\in R(s_i) \end{cases}$$

Undefined otherwise

Where $<G(a)>$ is true iff there exists a $b \in \text{COMP}(G)$ such that $a$ is a prefix of $b$.

Intuitively, $H_m(x)$ is an encoding of the final value of the $m$th cell after the termination of the comp $x$ under the Herbrand interpretation. Given $G \in \text{SCH}(S)$, two comps $x, y \in \text{COMP}(G)$ are related by the equivalence relation $E_H$ iff

$$\forall m \in M \quad [H_m(x) = H_m(y)] \text{ iff } x = y (E_H)$$

So, two comps are related by $E_H$ iff they assign the same value to every cell for all interpretations. $E_H$ formalizes our intuitive understanding of the semantic equivalence of two comps. It is to be noted here that an occurrence of an 'undefined' is by definition not equal to another occurrence of an 'undefined'. Thus, two schemata $G_1$ and $G_2$ are equivalent iff $G_1 \equiv G_2$ ($E_H$). The relation $E_H$ correctly captures our intuitive notion of semantic equivalence, but is very difficult to detect from the definition.

2.3 Syntactic equivalence

In this subsection, we define a syntactic equivalence relation for schemata. Since this equivalence is defined in terms of the graph properties of schemata, it can be algorithmically checked. Syntactic equivalence is based on a 'reads-from' relation and a property of liveness that we define for computation sequences. These concepts were defined for histories in the models of [BSW79, Papas79]. We adapt them to our model below. Given $x = x_1x_2 \ldots x_n \in \text{COMP}(G)$, we define an augmented comp as $x_{n+1} = x_0x_1 \ldots x_n x_{n+1}$, where $D(x_{n+1}) = R(x_0) = M_0, R(x_{n+1}) = D(x_0) = \emptyset$. Let $Q_a = Q \cup \{x_0, x_{n+1}\}$ be the augmented
terminators. For m ∈ M, define the relation **reads m from** (denoted $RF^m_x$), as:

$$RF^m_x = \{(x^i_n, x^n_{n+1}) \mid x^n_{n+1} \in Q \} \cup \{(x^i_j, x^n_j) \mid (mSD(x^i_j) \cap \hat{R}(x^n_j)) \cap (i < j) \}$$

Define $RF_x = \bigcup_{m \in M} RF^m_x$.

Intuitively, $(x^i_j, x^n_j) \in RF^m_x$ means $x^n_j$ reads the value in cell $m$ written by $x^i_j$.

Given an augmented comp of $x \in COMP(G)$, $x_a = x_0 x^1 x_2 \ldots x^n x_{n+1}$, $x \in x_a$ is said to be **live** in $x$ iff it satisfies the following:

1. $x_{n+1}$ is live in $x$; and
2. if for some live operation $x^i_j$, $(x^i_j, x^n_j) \in RF_x$ then $x^n_j$ is also live in $x$.

Intuitively, an operation is live in a computation if its effect is evident after the computation is completed. An operation $x^i_j \in x$ is said to be **dead** in $x$ iff it is not live in $x$.

Using the above properties, we define a syntactic equivalence relation $E_S$ as follows. Given $G \in SCH(S)$ and $x, y \in Q$, $x \approx y (E_S)$ iff $x$ and $y$ have the same live operations and the same reads-from relations for the live operations.

2.4 Relationship between syntactic and semantic equivalence

Equivalence of two comps under $E_S$ requires that every live operation read the same set of values for all cells in its domain in both the computation sequences. From the definition of liveness, we know that the effect of only the live operations are evident in the shared memory after the computation.

This observation leads to the following relationship between the two equivalence relations $E_S$ and $E_H$.

**Lemma 2-1:** Given $x, y \in Q$, $x \approx y (E_S)$ iff $x \approx y (E_H)$.

This has been shown by Papadimitriou et al. [PBR77] for histories. Since comps may be thought of as possible execution histories, this result carries over to our model, and leads to the following corollaries.
Corr 2-1.1: $G \in \text{SCH}(S)$ $G$ is $E_g$-determinate iff $G$ is $E_H$-determinate.

Corr 2-1.2: Given two schemata, $G_1, G_2 \in \text{SCH}(S)$

$G_1 \equiv G_2$ $(E_g)$ iff $G_1 \equiv G_2$ $(E_H)$.

From now on when we talk of equivalence of comps or schemata, we mean both $E_g$ and $E_H$ equivalence (since each implies the other).

3. MODEL OF EXECUTION

In the last section we characterized the properties of a schema. We now define some properties of an execution; in particular, we extend the concept of a correct (i.e. serializable) execution from the sequential model to our model of parallel execution. We present a syntactic procedure for deriving a schema with the property that every possible execution allowed by it is correct.

3.1 Execution and correctness criterion

Definition 3-1: An execution graph $X = (V_x, E_x)$ of a transaction system $T$ is a directed acyclic graph, defined by:

$V_x = \{ t_{ij} | i=1,2,\ldots,n, \quad j=1,2,\ldots,k_i \}$

$E_x = \{ (t_{ij}, t_{kl}) | \text{FIN}(t_{ij}) < \text{SRT}(t_{kl}) \}$

where \text{FIN}(t_{ij}) = \text{finishing time of } t_{ij},

\text{SRT}(t_{kl}) = \text{starting time of } t_{kl}.$

Intuitively, the execution graph depicts the order in which the transaction steps of $T$ were executed. (Note that since some transaction steps were executed in parallel, this is a partial order.) To characterize the properties of $X$, we first define the set of computation sequences of the execution graph as follows. Given an execution graph $X$, its set of computation sequences, denoted $\text{COMPUT}(X)$, is defined to be the set of topological sort sequences of $X$. Intuitively, $\text{COMPUT}(X)$ is the set of sequential executions which are computationally equivalent to the parallel execution $X$. Informally, we can refer to $x$ as a schema and view its edges as precedence constraints.
Consequently, we can extend the properties of schema to executions. Note that any execution \( X \) permitted by a schema \( G \) satisfies the precedence constraints in \( G \). So, \( G \) must be a subgraph of \( X \). Therefore, all properties attributable to a schema \( X \), also hold for any schema that permits the execution \( X \).

**Definition 3-2:** Given a transaction system \( T \), and a permutation \( p \) of \( \{1,2,\ldots,n\} \), (where \( p \) is viewed as a function), a **serial execution** corresponding to \( p \), (denoted \( SX_p \)), is an execution in which all transaction steps of \( T_p(j) \) are executed before any transaction step of \( T_p(k) \) is executed iff \( p(j) < p(k) \).

Thus, a serial execution imposes a total ordering on the transactions, and its graph is a chain.

**Proposition 3-1:** \( \text{COMPUT}(SX_p) \) has exactly one element.

Informally, we shall denote this computation sequence also by \( SX_p \). With the above definition of serial execution, we are ready to define the correctness criterion of serializability.

**Definition 3-3:** Given a computation sequence \( x \in \text{COMPUT}(X) \), we say that \( x \) is **serializable** iff there exists a permutation \( p \) such that \( x = SX_p \). An execution is serializable iff every element of \( \text{COMPUT}(X) \) is serializable.

**Proposition 3-2:** \( X \) is serializable iff \( X = SX_p \) for some permutation \( p \).

Thus, a parallel execution is correct iff all of its computation sequences are equivalent to some serial execution. Note that in this parallel model of execution, serializability ensures that if there are \( m \) transaction steps executing in parallel, then every one of the \( m! \) possible sequences of commitment of these transaction steps is serializable.

### 3.2 Schema for a transaction system.

We give below a syntactic procedure to obtain a schema with the property that every resulting execution is serializable. The following definitions are adapted from [BSW79, Papa79] to our model.

**Definition 3-4:** Given an augmented comp for an \( x \in \text{COMP}(G) \), define the **interferes relation** (denoted \( I_x \)) as:

\[
I_x = \{(x_i, x_1) | (x_i, x_1) \in \text{GR}_x \} \cup \{ (x_j, x_k) | (x_j, x_k) \in \text{RF}_x \} \cap \bigcap \{(x_j, x_k) \} \in \text{COMP}(G) \}
\]
Intuitively, $I_\pi$ guarantees that if $x_j$ reads a value in cell $m$ from $x_i$ and
$x_k$ writes into cell $m$ then $x_k$ should either precede $x_i$ or follow $x_j$. Using the
interferes and reads-from relations, the serialization graph of a comp is
defined as follows.

**Definition 3-5:** Given $G(V,E) \in \text{SCH}(S)$, and $x \in \text{COMP}(G)$, we define a
serialization graph of $x$ (abbreviated SR-graph of $x$), $G_x = (V_x,E_x)$ from a comp
$x \in \text{COMP}(G)$ as follows

$$V_x = V, E_x = RF_x \cup I_x.$$ 

The edges of the SR-graph of $x$ represent all the reads-from relations and
the interferes relations for $x$. So if we use $G_x$ as the schema then any comp
$y \in \text{COMP}(G_x)$ must reflect this ordering. This is stated in the following lemmas
(the proofs of which are given in the Appendix).

**Lemma 3-1:** Given $x \in \text{COMP}(G)$, $y \in \text{COMP}(G_x)$, if $x$ has no dead operations,
then $y$ has no dead operations.

**Lemma 3-2:** Given $x \in \text{COMP}(G)$, $y \in \text{COMP}(G_x)$, it follows that $RF_x = RF_y$.

**Lemma 3-3:** Given $G \in \text{SCH}(S)$, $x \in \text{COMP}(G)$, such that there is no dead
operation in $x$ then $x \equiv y (E_g)$ for all $y \in \text{COMP}(G_x)$.

This follows directly from Lemmas 3-1 and 3-2.

So Lemma 3-3 shows that if we can choose a computation sequence $x$ based on
some correctness criterion and derive $G_x$, then every comp $y \in \text{COMP}(G_x)$ is also
equivalent to $x$. The following corollary is an obvious extension.

**Corr 3-3.1:** $G_x$ is $E_g$-determinate.

Given a serial execution $S_X_p$, we derive a SR-graph after deleting all the
dead transaction steps in $S_X_p$. For convenience, we denote this $G_p$, (instead
of $G_{S_X_p}$, the notation used earlier). We would like to use $G_p$ as the schema.

For this, we show that any execution permitted by it is serializable.

**Theorem 3-1:** Given a SR-schema $G_p$ of a serial execution $S_X_p$, every execution
permitted by $G_p$ is serializable.

**Proof:** From lemma 3-3 it follows directly that every computation sequence in
$\text{COMP}(G_p)$ is serializable.

We know that every edge (i.e. precedence constraint) in $G_p$ is also an edge in
any execution $X$ resulting from $G_p$. Therefore every topological sort sequence
of X must also be a topological sort sequence of \( G_p \). So, \( \text{COMPUT}(X) \subseteq \text{COMP}(G_p) \). Therefore we can conclude that every topological sort sequence of \( \text{COMPUT}(X) \) is serializable to \( SX_p \).

Thus we have proved that \( G_p \) is a schema which represents the transaction system \( T \), and provides sufficient information to result in a correct execution. From now on we refer to \( G_p \) as the SR-schema. To show that this schema imposes a minimal set of precedence constraints on the transaction steps in \( T \), we prove the following theorem.

**Theorem 3-2:** Given a SR-schema \( G_p \), let \( G_{p_{\text{min}}} \) be the schema with fewest edges such that \( (G_p)^+ = (G_{p_{\text{min}}})^+ \). (where \( G^+ \) is the irreflexive transitive closure of \( G \)). Then \( G_{p_{\text{min}}} \) has the necessary and sufficient precedence constraints required for any schema to be equivalent to schema \( SX_p \).

**Proof:** Sufficiency follows directly from Theorem 3-1.

Necessity is proved as follows. First, observe that \( G_{p_{\text{min}}} \) is unique for a given \( G_p \), as \( G_p \) is a directed acyclic graph. Now suppose we remove an edge \( e = (x_i, x_j) \) from \( G_{p_{\text{min}}} \).

Case 1: If \( e \) is a reads-from edge, then \( G_{p_{\text{min}}} \) is not equivalent to \( G_p \) (\( \neq SX_p \)) under the equivalence relation \( E^p \); thus it is not equivalent under relation \( E_H \) too.

Case 2: If \( e \) is an interferes edge, then (without loss of generality) let \( x_i \) read a value in cell \( m \) from \( x_k \), and \( x_j \) write into cell \( m \). If \( e \) is removed, then there is a computation sequence \( y \) in which \( x_j \) follows \( x_k \), and precedes \( x_i \). So in \( y \), \( x_i \) reads cell \( m \) from \( x_j \) instead of \( x_k \). Thus \( y \) is not equivalent to \( SX_p \), which implies that \( G_p \) is not serializable.

This proves that \( G_{p_{\text{min}}} \) represents the necessary and sufficient set of precedence constraints for any schema to be equivalent to \( SX_p \). [QED]

But it is clear that this minimality is for a particular choice of serial execution \( SX_p \), i.e. for a particular permutation \( p \). But there are \( n! \) SR-schemata to choose from as there are that many serial permutations \( p \). If we want to choose an optimal schema from the \( n! \) possible schemata, then minimising the number of edges may not necessarily be a good measure. In [KD81] we present five metrics for optimization, which have been culled from the literature. We show that one of these metrics is useless for this problem, and, for the remaining four, the optimization problems are intractable (i.e. NP-hard). Hence, it seems likely that finding an optimal
schema, using only syntactic information, is intractable. But Kung and Papadimitriou [KP79] have shown that semantic information can be used to achieve greater parallelism. We explore this idea in the next section.

4. TRANSACTION MODIFICATION

It was observed that the scheduling problem using syntactic information alone (i.e., readsets and writesets) is intractable for all interesting performance measures. There are two ways to cope with this intractability. One way is to find an approximation algorithm that finds a suboptimal solution. The other way is to redefine the problem, by changing the domain of optimization or by relaxing the optimality measure, so as to simplify the problem. This section shows how semantic information can be used to redefine the problem to make it simpler. One approach is to construct a schema that is serializable and then to optimize it using semantic information. Semantic information is incorporated by partially interpreting the operations. Thus, each transaction step is assumed to be a statement (retrieval or update request) in a high-level query language. We develop a set of rules for transaction modification. Each modification transforms a schema (for a given transaction system) into an equivalent schema which is better according to some performance measure. Initially this measure is assumed to be the diameter of the schema. (We point out in [KD81] the practical significance of this measure and its superiority over the others.) Later, this is extended to a measure based on data base cost.
4.1 Environment

The database is assumed to consist of a single relation \( R \). In a database with more than one relation, \( R \) can be thought of as the product of all the relations. Each tuple of \( R \) has a unique identifier (TID) and corresponds to a cell of the shared memory \( M_0 \). Transaction steps are assumed to be statements written in some high-level relational calculus-based language, such as QUELO [HSW75]. As the exact syntax is not of concern here, we represent each type of statement in the following manner:

1. MODIFY: \( \text{MOD}(\text{targ}_m, q_m, R) \)
2. INSERT: \( \text{INS}(\text{targ}_i, q_i, R) \)
3. DELETE: \( \text{DEL}(q_d, R) \)
4. RETRIEVE: \( \text{RET}(\text{targ}_r, q_r, R) \)

In each statement the qualification \( q \) is a predicate that selects a subset of the relation \( R \) to be the operand of the operation; the target list \( \text{targ} \) defines the computations to be performed on the operand. In QUELO, the qualification is a boolean combination of clauses of the form \( \langle \text{term} \rangle \langle \text{op} \rangle \langle \text{term} \rangle \), where \( \langle \text{term} \rangle \) is an attribute, a constant, or an arithmetic function (e.g. \( +,* \)) of other terms; and \( \langle \text{op} \rangle \) is an arithmetic comparison operator (e.g. \( =,\leq \)). The target list is a list of (attribute,term) pairs. Initially, we assume that terms in a target list are constants, but we show section 4-5 that this assumption can be relaxed. Note that we do not permit aggregation functions (e.g. average,sum) in our statements. Without loss of generality, we assume that a target list specifies every attribute in \( R \). This assumption and our earlier assumption that the database contains only one relation do not restrict the applicability of the theory; they are made merely to simplify the formalism.
Let \( Q_X \) be the set of tuples selected by qualification \( q_X \) and \( \text{TARG}_X \) be a function corresponding to target list \( \text{targ}_X \), such that, \( \text{TARG}_X(Q_X) \) is the set of tuples generated by evaluating \( \text{targ}_X \) on the selected tuples \( Q_X \). For a single tuple \( t \), \( \text{TARG}_X(t) \) is defined in the obvious manner. Figure 4-1 tabulates the effect of each operation.

We shall consider a general class of query processing strategies in which each operation is performed in two steps (as shown in figure 4-2): a qualification step \( q \) that selects the tuple ids (TIDs) of the \( R \)-tuples satisfying the qualification \( q \); then, an effect step \( e \) that performs the operation specified by the target list on the tuples whose TIDs were selected by \( q \). For this class of query processing strategies, the time taken to evaluate the qualification (step \( q \)) is likely to be much greater than the time taken for step \( e \). Also, step \( q \) does not update the database. These two observations make it a prime candidate for execution in parallel with other qualification steps (appropriately modified, as described in the sequel).

Consider an ordered pair of operations with the precedence constraints shown in figure 4-3. Suppose this pair can be modified to an equivalent schema (shown in figure 4-4) such that \( Te_1, Te_2, Tq_1, \) and \( Tq_2 \) can be syntactically determined. Then, the modified operation pair has smaller diameter and so is better according to our measure of parallelism.

In developing the modifications, we use a canonical representation of the update operations (see figure 4-5), called \( \mathcal{H} \)-graph. In this representation, \( q_d \) and \( q_i \) are the qualifications that select the tuples to be deleted and inserted respectively; \( e_d \) and \( e_i \) are the corresponding effects. It is obvious that the modify, insert and delete operations can be modelled by choosing appropriate \( q \)'s. To model retrieve operations we observe that all retrieve
In the following table $R, R', R''$ are the relations before the execution, after the execution and the result of the retrieve operation. Also let

$$Q_x = \{ t \mid t \in R \text{ and } t \text{ satisfies } q_x \}$$

and $\text{TARG}_x$ is the function corresponding to $\text{targ}_x$.

The following table lists the effect of each operation:

<table>
<thead>
<tr>
<th>Operation</th>
<th>$R' = (R - Q_m) \setminus \text{TARG}_m(Q_m)$</th>
<th>$R'' = \emptyset$</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOD($\text{targ}_m, q_m, R$)</td>
<td>$R' = R \setminus \text{TARG}_i(Q_i)$</td>
<td>$R'' = \emptyset$</td>
</tr>
<tr>
<td>INS($\text{targ}_i, q_i, R$)</td>
<td>$R' = R - Q_d$</td>
<td>$R'' = \emptyset$</td>
</tr>
<tr>
<td>DEL($q_d, R$)</td>
<td>$R' = R$</td>
<td>$R'' = Q_r$</td>
</tr>
<tr>
<td>RET($\text{targ}_r, q_r, R$)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$R_{OS}$ = the resulting relation after an execution permitted by OS.

$R_{TS}$ = the resulting relation after an execution permitted by TS.

Figure 4-1: Effects of the operations.
Fig. 4-2: Model of an operation

Fig. 4-3: Schema for an ordered pair of operations

Fig. 4-4: Modified schema for an ordered pair of operations

Fig. 4-5: Canonical representations of an operation graph.
operations in a transaction are live, i.e., the retrieved values are used in a subsequent update (modify, insert, or delete) operation. Hence, we can combine the retrieve operation with the update operation and correspondingly change the effect step e of the update operation. An example of such a modification is shown in Figure 4-6. From a practical standpoint, we do not expect to see dead transaction steps. At the very least, a transaction step should record the fact that it ran by writing on an output device which in our model is part of the shared memory.

An assumption that we make here is that every insert (ei) create a new tuple of relation R. Thus, every new tuple that is inserted into R is assigned a new TID. This implies that a modify operation is modelled as the deletion of old tuples and the insertion of new (modified) tuples with new TID's. This assumption in conjunction with an earlier assumption (that only constants are permitted as terms in target lists) ensures the disjointness property. It is this disjointness property that we need in our proofs for this subsection. We argue in section 4-5 that and implementation can, in fact relax these assumptions, and still satisfy the disjointness property.

With this canonical model of an operation, we define an elementary schema called an operation pair as shown in figure 4-7. Note here that the graph may have more than one statement associated with it. For this elementary schema, we derive an equivalent schema with greater parallelism (i.e. smaller diameter). Then we define a generalized canonical form of elementary schema and show how to extend the transformation to this generalized form. Further, we show that the transformed schema is of the same generalized canonical form so that it can be subsequently paired with some other operation. This closure property is useful in repetitive applications of the transformation.
Fig. 4-7: Schema (OS) for an operation pair.

Fig. 4-8: Transformed schema (TS) for an operation pair.
Tuples selected tuples by q's in the original schema:

\[ OQ_{i1} = \{ t \mid \text{t satisfies } q_{11} \} \]
\[ OQ_{d1} = \{ t \mid \text{t satisfies } q_{d1} \} \]
\[ OQ_{i2} = \{ t \mid \text{t satisfies } q_{12} \} \]
\[ OQ_{d2} = \{ t \mid \text{t satisfies } q_{d2} \} \]

Read/write sets for e's in the original schema

\[ OE_{i1} = OQ_{i1} \quad OE_{i2} = OQ_{i2} \]
\[ OE_{d1} = OQ_{d1} \quad OE_{d2} = OQ_{d2} \]

Tuples selected by Tq's in the transformed schema:

\[ TQ_{i1} = \{ t \mid \text{t satisfies } q_{i1} \} \quad k=1,2 \]
\[ TQ_{d1} = \{ t \mid \text{t satisfies } q_{d1} \} \quad k=1,2 \]
\[ TQ_{i2} = \{ t \mid \text{t satisfies } TARG(t) \} \quad i=1 \]
\[ TQ_{d2} = \{ t \mid \text{t satisfies } TARG(t) \} \quad i=2 \]

Read/write sets for Te's in the transformed schema:

\[ TE_{i1} = TQ_{i1} - TQ_{d1} \]
\[ TE_{d1} = TQ_{d1} \]
\[ TE_{i2} = TQ_{i2} - TQ_{d1} \]
\[ TE_{d2} = TQ_{d1} - TQ_{d2} \]
\[ TE_{i2} = TQ_{i1} /\ TQ_{i2} \]

Target lists for the Tei's in the transformed schema:

The target lists for TE_{i1} and TE_{i2} are the same as in the original schema. The target list for TE_{i2}/1 is the composition of TARG_{11} with TARG_{12}.

Figure 4-9: OE's, OQ's, TE's and TQ's
4.2 Transformation of an operation pair in series

Given a schema OS, (original schema,) for an operation pair in series, shown in figure 4-7, we can find an equivalent schema TS, (transformed schema), as shown in figure 4-8. In this transformed schema Ted₁, Ted₂, Tei₁, Tei₂, and Tei₃ are the transformed operations whose readsets and writesets and target list functions are defined in figure 4-9. This schema evaluates the qualifications based on the original relation R, to get TQd₁, TQi₁, TQd₂/₁, TQi₂/₁, TQi₁, and TQi₂. Using these sets of TIDs, the readsets and writesets of the operation can be determined. The validity of the transformation is proved formally below. Here, we attempt an intuitive justification. (The reader will find it helpful in following this explanation if he/she refers to the example in figure 4-10.) Ted₁ is the same as TQd₁, since both TQd₁ and OQd₁ select tuples from the original state of the relation R. Tei₁ is the set of tuples in TQi₁ and not in TQd₂/₁, because the tuples in TQd₂/₁ are subsequently deleted in step ed₂ of OS. Hence, those tuples which are inserted and subsequently deleted in OS are not inserted at all in TS. Consequently, Ted₂, the set of tuples to be deleted from R by ed₂, are only those which were not already deleted in step ed₁. Tei₂ is the set of tuples in R from which the step ei₂ creates new tuples. So, Tei₂ consists of only those tuples which were not deleted in step ed₁. Tei₂/₁ is the set of tuples inserted by step ei₂ based on tuples inserted by ei₁; the computation performed in this step is the composition of computation specified by the target lists of the two operations. Thus, the transformed schema performs the same deletions and insertions on R as the original schema. And, significantly, the transformed schema allows greater parallelism amongst the qualification steps, which, by our assumption, are more time consuming than the effect steps.
<table>
<thead>
<tr>
<th>TID</th>
<th>AGE</th>
<th>SALARY</th>
<th>qd₁ = (salary = 10K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17</td>
<td>10K</td>
<td>qi₁ = (age = 16 and salary = 30K)</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>10K</td>
<td>qd₂ = (age = 16)</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>30K</td>
<td>qi₂ = (age = 17)</td>
</tr>
<tr>
<td>4</td>
<td>19</td>
<td>30K</td>
<td>TARG₁ = (salary = 25K)</td>
</tr>
<tr>
<td>5</td>
<td>17</td>
<td>20K</td>
<td>TARG₂ = (salary = 27K)</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td>30K</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>30K</td>
<td>rig1 = 1, 2, 3, 4, 5, 6, 7, 8</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>15K</td>
<td>R = 3, 4, 5, 6, 10, 11, 12, 13, 14</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>25K</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>17</td>
<td>25K</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>18</td>
<td>25K</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>17</td>
<td>27K</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>17</td>
<td>27K</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>17</td>
<td>27K</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Tuples</td>
</tr>
</tbody>
</table>

OQd₁ = {1, 2}  OQd₂ = {7, 8, 9}  OQ₁₁ = {3, 6, 7}  OQ₁₂ = {5, 6, 10}
OE₁₁ = {1, 2}  OE₁₂ = {7, 8, 9}  OE₁₁ = {9, 10, 11}  OE₁₂ = {12, 13, 14}
TQ₁₁ = {1, 2}  TQ₁₂ = {2, 7, 8, 9}  TQ₁₁ = {3, 6, 7}  TQ₁₂ = {1, 5, 6}
TQ₁₁ = {7}  TQ₁₂ = {9}  
TE₁₁ = {1, 2}  TE₁₂ = {7, 8}  TE₁₁ = {3, 6}  TE₁₂ = {5, 6}

TE₁₂ = {2}

Fig 4.10: An example of the transformation of an operation pair
To show the validity of the transformation, we first show that the transformed schema is determinate; i.e. all computations for this schema are equivalent. For this we make this following observation, which are proved in the Appendix.

Ted’s and Tei’s are said to be **conflict free** if the writesets of Ted’s are mutually disjoint and pairwise disjoint with the read/write sets of Tei’s; the readssets of Tei’s are pairwise disjoint with the writesets of Tei’s; and the writesets of Tei’s are mutually disjoint.

**Lemma 4-1**: Ted_1, Ted_2, Tei_1, Tei_2 and Tei_3 are conflict free.

**Proposition 4-1**: If Ted_1, Ted_2, Tei_1, Tei_2, and Tei_3 are conflict free then the transformed schema is determinate.

This proposition has been proved by Keller [Kel73] for his general parallel program schemata. Intuitively, this follows from the observation that the disjointness property ensures that no tuple is updated by two steps executing in parallel. Lemma 4-1 and Keller’s proposition lead to the following theorem.

**Theorem 4-1**: The transformed schema is determinate.

Having shown that TS is determinate, we now have to show the equivalence of the two schemata to prove the correctness of the transformation. For this we must prove that every tuple t ∈ R ∪ R_0S (see figure 4-1) that was deleted, inserted, or remained unchanged in the original schema, was correspondingly deleted, inserted, or remained unchanged in the transformed schema. This is stated in the following sequence of lemmas (the proofs of which appear in the Appendix).

**Lemma 4-2**: If a tuple t ∈ R is deleted by OS, then t is also deleted by the transformed schema TS.

**Lemma 4-3**: If a tuple t ∈ R_0S is inserted by the original schema OS, then t is also inserted by the transformed schema TS, and t ∈ R_Ts.

**Lemma 4-4**: If a tuple t ∈ R is not updated by OS, then t is not updated by TS.
Let $U_{OS} = R \parallel \text{TARG}_{11}(OE_{i1}) \parallel \text{TARG}_{12}(OE_{i2})$, 
and $U_{TS} = R \parallel \text{TARG}_{11}(TE_{i1}) \parallel \text{TARG}_{12}(TE_{i2}) \parallel \text{TARG}_{13}(TE_{i3})$.

Thus, $U_{OS}$ and $U_{TS}$ are the universes for the tuples in relations $R_{OS}$ and $R_{TS}$ respectively.

Lemma 4-5: If a tuple $t \in U_{OS}$ and $t \notin R_{OS}$ then $t \in U_{TS}-R_{TS}$.

Lemma 4-6: $U_{TS} \subseteq U_{OS}$

Lemma 4-7: $(U_{TS}-R_{TS}) \subseteq (U_{OS}-R_{OS})$.

Theorem 4-2: $R_{OS} = R_{TS}$

Proof: To prove this we discuss the following cases:

case 1: $t \in R-R_{OS} \rightarrow t \in R-R_{TS}$  
(from lemma 4-2)

case 2: $t \in R_{OS}-R \rightarrow t \in R_{TS}-R$  
(from lemma 4-3)

case 3: $t \in R \cap R_{OS} \rightarrow t \in R \cap R_{TS}$  
(from lemma 4-4)

case 4: $t \in U_{OS}-R_{OS} \rightarrow t \in U_{TS}-R_{TS}$  
(from lemma 4-5)

case 5: $t \in U_{TS}-R_{TS} \rightarrow t \in U_{OS}-R_{OS}$  
(from lemma 4-7)

From the above five cases it follows that $R_{OS} = R_{TS}$.

[QED]

Thus we have shown that the transformed schema, TS, and the original schema, OS, are equivalent.

4.3 Generalized transformation

We have shown how to transform a pair of operations into a schema having greater parallelism. The transformed schema is not quite in the canonical form of an operation (figure 4-6) because step ei has more than one target list. Thus, if this operation pair is part of a bigger schema and we want it to participate in further transformations with successor nodes, the above transformation cannot be directly used. We now show how to generalize the transformation. First we generalize the concept of an operation pair. This generalization is shown in figure 4-11. The first stage has k insert nodes.
represents all the directed edges in the cross product.

Fig. 4-11: Schema for generalized operation pair
Fig. 4-12: Transformed schema for generalized operation pair along with the read/write sets.
and l delete nodes. The second stage has m insert nodes and n delete nodes. The transformation and the associated read/write sets are given in figure 4-12. The proof of the previous section can be extended in a straightforward manner for this generalized transformation and does not provide any new insight into the problem. (Further, the proof is messy because of the proliferation of subscripts and is omitted here.) The intuitive justification for the read/write sets is the same as before. TEd,a, a=1,2,...,l, are the same as TQd,a since both TQd,a and OQd,a select tuples from the original state of R. TEi,a, a=1,2,...,k, is the set of tuples in TQi,a and not in TQd,b/a for any b \in \{1+1,1+2,...,n\}, because the tuples in TQd,b/a are to be subsequently deleted in the second stage of OS. Hence, those tuples which are inserted and subsequently deleted in OS are not inserted at all in TS. Consequently, TEd,a, a=1+1,1+2,...,n, the sets of tuples to be deleted from R contain only those tuples that were not already deleted by the first stage. TEi,a, a=k+1,k+2,...,m, are the tuples in R from which the steps ei_{k+1}, ei_{k+2}, ...,ei_{m} created new tuples. So TEi,a consists of only those tuples which were not deleted in the first stage. As before, TEi,a/b, a=k+1,...,m; b=1+1,...,n, is the set of tuples that were inserted in the second stage based on tuples that were inserted in the first stage. Thus the transformed schema performs the same insertions and deletions as the original schema, but has greater parallelism. Furthermore, the transformed schema is in the generalized canonical form of figure 4-11, and so can be used in subsequent transformations.
4.4 Algorithm for transformation

Suppose that the data base machine has \( k \) processors. Once the scheduler has constructed a schema representing minimal precedence constraints, it now remains to assign available processors to execute the nodes of the DAG. To maximize processor utilization, it is important to ensure that at every point in time, as many of the \( k \) processors as possible are busy. We shall show how to transform the schema to meet this objective.

We have to select \( k \) nodes to execute on the \( k \) processors. Assuming that each node takes unit time to execute, all \( k \) nodes will complete at the same time. (This assumption is relaxed in the section 4.5). At some point in this process, if there are \( m \leq k \) nodes ready for execution, then we can use the generalized transformation developed above to obtain \( k \) nodes in parallel. To do this, let \( n \) be the number of nodes that can be enabled for execution after the \( m \) nodes have been executed. From these \( n \) nodes choose \( n' = \min(n, k-m) \) nodes. Then we can view the DAG as shown in figure 4-13. We see that there are no edges from level 1 to level 0, and from level 2 to either level 1 or 0. To this graph, add edges between every node at level 0 to every node at level 1 (this is not necessarily how the algorithm will be implemented). It is obvious that the graph is still acyclic and the added edges do not contradict any existing precedence constraints. Now the set of nodes at level 0 and level 1 conforms to the generalized canonical form of figure 4-11. So we can apply the generalized transformation to get \( m+n' \) nodes in parallel; the transformed graph is shown in figure 4-14. If \( m+n'<k \), then this process of transformation can be repeated until \( k \) parallel nodes are available.
Fig. 4-13: Original DAG

Fig. 4-14: Transformed DAG
4.5 Implementation considerations

Two main assumptions were made in developing the above theory. These were:
1. Only constants were allowed as terms in a target list;
2. all inserted tuples had new TID's.

Both these assumptions were used to ensure that the ed's and ei's were conflict free, i.e. the write sets of ed's were mutually disjoint and pairwise disjoint with the read/write sets of ei's; the readsets of the ei's were pairwise disjoint with the writesets of ei's; and the writesets of ei's were mutually disjoint. We show here that we can relax these assumptions by adopting the following implementation. Associated with each tuple are two flags: delete flag and an insert flag, whose use is described below. Step ed sets the delete flags of all those tuples which are selected for deletion but do not have their insert flags set. These tuples are then deleted during the execution of the next transaction system. Step ei always creates new tuples for insertion and sets their insert flags. If ei is part of a modify operation, then it reuses the old TID; otherwise, it generates a new TID. (This implies that at any point in time, there might be two versions of a tuple both having the same TID.) In reading an existing tuple x, in order to compute a tuple y for insertion, ei uses that version of x which does not have its insert flag set. Further, the insert flag is reset in these tuples at the beginning of the next transaction system.

So any step ed writes only delete flags, and therefore the writesets of ed's are pairwise disjoint with the read/write sets of ei's; we already know from the transformation that the writesets of ed's are mutually disjoint. The insert flag ensures that the readsets of ei's are pairwise disjoint with the writesets of ei's. Creating new tuples ensures that the writesets of ei's are
mutually disjoint. Therefore, the theory of sections 4-2 and 4-3 still holds.

Thus, these transformations can be used repetitively to reduce the diameter of the schema. But it is clear that reduction in the diameter may not be without cost: the number of nodes increases and so does the complexity of each node. It might be more appropriate to pick an optimality measure that takes into account the processing costs of the nodes. Several cost measures for query processing have been proposed in the literature [HY79, Yao79]. These are based on physical parameters such as file sizes, attribute selectivities, storage and access methods. Given any cost measure that imposes a total ordering on the set of schemata, we apply the schema transformations described in this section only if it is beneficial to do so, i.e. only if the estimated cost of the transformation is less than that of the original schema.

5. CONCLUSION

In this paper, we developed a parallel program schema model of transaction systems for parallel database machine. The concept of serializability, which is generally accepted as the correctness in the existing concurrency control theory for the sequential model, was extended to our model. We proposed a two-step technique for producing correct and highly parallel schedules: first, obtain a schema that imposes a minimal set of precedence constraints on correct executions; then, transform the schema using semantic information to increase parallelism. Although the model developed in this paper is theoretical, we believe it to be of practical utility — the proposed scheduling can be applied to any MIMD machine such as DIRECT [DeW78].

Several interesting performance related questions may be posed here. We described the scheduler as a single, centralized process. Will this become a bottleneck? Alternatively, given ample resources and the parallelism inherent
in the system, will it be beneficial to partition the system and distribute the scheduling activity over several processes. Our theory is independent of whether the scheduler is centralised or distributed. Further, we have implied a batched mode of operation for the machine. Each transaction system can be thought of as a batch. This has the advantage that while one transaction system is being executed, the scheduler can be working in parallel on the next transaction system. Clearly, the selection of transactions to comprise a transaction system is a crucial factor affecting performance. An alternative to batching is to dynamically schedule transactions as they arrive. Will this improve performance? Simulation studies or queueing analysis can provide the answers to these questions.

The transformation presented in section 4 produces nodes that must be capable of evaluating arbitrarily complicated set expressions. The complexity of some of these nodes may be reduced by refining the nodes (i.e. replacing each by a more detailed subgraph) and then detecting common subexpressions across nodes of the subgraphs. As we pointed out before, a cost based on physical database parameters, must be attached to each node. When this is done, it can be determined when it is beneficial to transform a given schema.

Lastly, in section 4 we ignored the problem of eliminating duplicate tuples when an insert or modify operation is executed. We treat this as a special case of integrity checking. Integrity checking could be implemented as part of the effect step of an update operation. However, a more intriguing possibility is to use query modification (as suggested in [ston75]), together with the schema transformation of section 4, to perform integrity checking in parallel with the execution of the update. (for example, tuples which are being duplicated can be flagged for subsequent deletions.) Working out the details of this modification is a topic of future research.
REFERENCES


M.Gouda, "Simultaneity in Distributed Databases", Technical Report, Dept. of Computer Sciences, Univ. of Texas, Austin, TX, Oct 1980.


M.R.Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification", Proc. of ACM-SIGMOD Intl. Conf. on Management of Data, San Jose, 1975, pp65-78

I. Appendix: PROOFS

Lemma 3-1: given x ∈ COMP(G), y ∈ COMP(G_x), if x has no dead operations then y has no dead operations.

Proof: Let x = x_1x_2...x_n and y = y_1y_2...y_n. Let y_j ∈ y be such that y_{j+1}, y_{j+2}, y_n are live, and y_j is dead. Find j' such that x_{j'} = y_j. Obviously there does not exist m such that (y_j, y_m) ∈ RF_y, but there exists k' such that (x_{j'}, x_k') ∈ RF_x. Find k such that y_k = x_{k'}. But, k < j is not possible because (x_{j'}, x_k') ∈ RF_x and therefore (x_{j'}, x_k') is a precedence constraint in G_x. So (y_j, y_k) ∈ E_x. From this it directly follows that j ≤ k. From the definition of liveness we can conclude that

\[ P(j < 1 < k) \land (W(y_j) \cap R(y_k) \cap \neg W(y_1) = \emptyset) \lor W(y_j) \cap W(y_1) \neq \emptyset. \]

Find 1' such that x_{1'} = y_1. But, j' < 1' < k' is not possible because x_{k'} reads from x_{j'}. And if j' > 1' or k' < 1' then there will be an interferes edge in G_x to impose that same order in y. Thus, there can be no dead operation in y. [QED]

Lemma 3-2: Given x ∈ COMP(G), y ∈ COMP(G_x) it follows that RF_x = RF_y

Proof: Let x = x_1x_2...x_n, and y = y_1y_2...y_n. Find a Y_j such that y_j reads the value in cell m written by y_k, and the corresponding x_{j'} (=y_j) does not read the value in cell m written by x_{k'} (=y_k). Let x_{j'} read the value in cell m written by x_{1'}. Find 1 such that x_{1'} = y_1.

As x_{j'} reads from x_{1'}, so

1' < j' .... (1)

As y_j reads from y_k, so

k < j .... (2)

If k' > j', then, by definition, (x_{j'}, x_{k'}) is an interferes edge, and consequently, a precedence constraint in G_x. Then, k < j as in (2) is not possible. So,

k' ≤ j' .... (3)

If 1' < k' < j', then x_{j'} will not read from x_{1'} as assumed. So

k' < 1' < j' .... (4)

using (1) and precedence constraints in G_x,

If k < 1 < j, then y_j will not read from y_k as assumed. So

1 < k < j .... (5)

From (4) we note that there will be an interferes edge (x_{k'}, x_{1'}) in G_{x_0} and
this order is not retained in \( y \), as given in (6), this contradicts the precedence constraint in \( C_x \).

Hence, we can conclude that if \( y_j \) reads \( m \) from \( y_k \), then \( x_j' \) reads \( m \) from \( x_k' \), for all \( m \) in the domain of \( y_j \). In other words \( RF^m_X \subseteq RF^m_X \) for all \( m \in M \).

Now we show that if \( x_j \) reads \( m \) from \( x_k \), then \( y_j' (=x_j) \) reads \( m \) from \( y_k' (=x_k) \) for all \( x_j \) and every \( m \in R(x_j) \). If this is not true, then there exists an \( x_j \) that reads \( m \) from \( x_k \), and \( y_j' \) does not read \( m \) from \( y_k' \). Then \( y_j' \) reads \( m \) from some other operation, which contradicts the above claim that \( RF^m_X \subseteq RF^m_X \).

So it must be that \( RF^m_X \subseteq RF^m_X \). Hence we have shown that, \( RF^m_X = RF^m_X \) for all \( m \in M \).
Lemma 2.1: Given a SR-schema \( G_p \) of a serial execution \( SX_p \), every execution resulting from \( G_p \) is serializable.

Proof: From lemma 3-3 it follows directly that every computation sequence in \( \text{COMP}(G_p) \) is serializable.

We know that every edge (i.e. precedence constraint) in \( G_p \) is also an edge in any execution \( X \) resulting from \( G_p \). Therefore every topological sort sequence of \( X \) must also be a topological sort sequence of \( G_p \). So,

\[
\text{COMPUT}(X) \subseteq \text{COMP}(G_p)
\]

Therefore we can conclude that every topological sort sequence of \( \text{COMPUT}(X) \) is serializable to \( SX_p \).

Lemma 4-1: \( \text{TEd}_1, \text{TEd}_2, \text{Tei}_1, \text{Tei}_2 \) and \( \text{Tei}_3 \) are mutually disjoint.

Proof: First, observe that every inserted tuple is a new tuple. From this it directly follows that \( (\text{TEd}_1 \cup \text{TEd}_2) \cap (\text{Tei}_1 \cup \text{Tei}_2 \cup \text{Tei}_3) = \emptyset \) and \( \text{Tei}_1, \text{Tei}_2, \) and \( \text{Tei}_3 \) are mutually disjoint. From the definition of \( \text{TEd}_1 \) and \( \text{TEd}_2 \) it follows that \( \text{TEd}_1 \cap \text{TEd}_2 = \emptyset \).

QED

Lemma 4-2: If a tuple \( t \in R \) is deleted by \( \text{OS} \), then \( t \) is also deleted by the transformed schema \( \text{TS} \).

Proof: case 1 \( (t \in \text{OQd}_1) \): As \( \text{TEd}_1 = \text{OQd}_1 \), \( t \) is also deleted by \( \text{TS} \).

case 2 \( (t \in \text{OQd}_2) \): This implies that \( t \) was not deleted by step \( \text{ed}_1 \); i.e. \( t \notin \text{OEd}_1 \). From the definition of \( \text{TQd}_2 \), we know that \( t \in \text{TQd}_2 \), and as \( \text{TEd}_1 = \text{TQd}_2 = \text{OQd}_1 = \text{OEd}_1 \); so \( t \in \text{TEd}_2 \) and is deleted by step \( \text{Ted}_2 \).

From the above two cases we know that \( t \) will be deleted and from the determinacy of \( \text{TS} \) we are assured of no other updates on \( t \).

QED

Lemma 4-3: If a tuple \( t \in R_{\text{OS}} \) is inserted by the original schema \( \text{OS} \), then \( t \) is also inserted by the transformed schema \( \text{TS} \), and \( t \in R_{\text{TS}} \).

Proof: Let \( t_1, t_2 \) be tuples such that \( t = \text{TARG}_{i1}(t_1) \) and \( t = \text{TARG}_{i2}(t_2) \).
depending on whether \( e_i \) or \( e_j \) inserted the tuple \( t \) in \( \text{OS} \).

**Case 1** \((\mathbf{t}_1 \in \text{OQ}_1 \text{ and } \mathbf{t}_1 \notin \text{OQ}_d)\): \( t_1 \in \text{OQ}_1 \) implies \( t_1 \in \text{TQ}_1 \) (by definition). If \( t_1 \in \text{TQ}_d/1 \) then the tuple \( t \) inserted in step \( e_i \) will be selected by \( \text{qd}_2 \); i.e. \( t \notin \text{OQd}_2 = \text{OE}_d \). Then \( t \) cannot be a tuple in \( \text{RO}_t \) as step \( \text{ed}_2 \) would have deleted it. So \( t_1 \notin \text{TQ}_d/1 \). It follows that \( t_1 \notin \text{TQ}_1 \). So from lemma 4-1 we can conclude that in both the schemata \( t = \text{TARG}_1(t_1) \) is inserted.

**Case 2** \((\mathbf{t}_2 \in \text{OQ}_1)\): If \( t_2 \in \text{TQ}_2 \) then \( t_2 \in \text{TE}_2 \) because if \( t_2 \in \text{TQ}_d \) = \( \text{Qd}_1 \) then \( t_2 \) could not have been selected by \( \text{qi}_2 \), since it would have been deleted by \( \text{ed}_2 \); but this would contradict \( t_2 \in \text{OQ}_2 \). If \( t_2 \notin \text{TQ}_2 \) then we observe from the precedence constraints of \( \text{OS} \) that \( t_2 \in \text{TARG}_1(\text{OE}_1) \). This implies that there exists a tuple \( t_1 \in \text{R} \), such that \( t_2 = \text{TARG}_1(t_1) \) and \( t_1 \notin \text{OQ}_1 \). From the definition of \( \text{TQ}_2/1 \), if \( t_1 \in \text{R} \) and \( t_2 = \text{TARG}_1(t_1) \in \text{OQ}_2 \) then \( t_1 \notin \text{TQ}_2/1 \). As we already know, \( t_1 \in \text{OQ}_1 \) implies \( t_1 \notin \text{Q}_1 \), so \( t_1 \notin \text{TQ}_1 \cap \text{TQ}_2/1 \). Thus, \( \text{OS} \) and \( \text{TS} \) insert the same tuple \( \text{TARG}_2(t_2) = \text{TARG}_2(\text{TARG}_1(t_1)) \). Once again, from Lemma 4-1 we can conclude that no other update was done on that tuple.

QED

**Lemma 4-4:** If a tuple \( t \in \text{R} \) is not updated by \( \text{OS} \), then \( t \) is not updated by \( \text{TS} \).

**Proof:** If \( t \in \text{R} \) and \( t \notin \text{RO}_t \), then \( t \) is not a tuple that was inserted by \( \text{OS} \). Further, since, \( t \) is not updated by \( \text{OS} \), \( t \notin \text{OQd}_1 \) and \( t \notin \text{OQd}_2 \). \( t \notin \text{OQd}_1 \) implies \( t \notin \text{TQd}_1 = \text{TE}_d \). If \( t \in \text{TQd}_2 \), then \( t \notin \text{OQd}_2 \) because \( t \in \text{R} \) and \( t \notin \text{TQd}_1 \). So \( t \notin \text{TQd}_2 \). Therefore, \( t \notin \text{TE}_d \). That \( t \) is not updated in \( \text{TS} \) follows from the mutual disjointness property.

QED

**Lemma 4-5:** If a tuple \( t \in \text{U}_\text{OS} \) and \( t \notin \text{RO}_t \) then \( t \notin \text{RT}_t \).

**Proof:** If \( t \in \text{R} \) then by Lemma 4-2 \( t \notin \text{RT}_t \). If \( t \in \text{TARG}_2(\text{OE}_2) \) then \( t \in \text{RO}_t \); so this lemma is vacuously true. If \( t \in \text{TARG}_1(\text{OE}_1) \) and \( t \notin \text{RO}_t \) then it
follows that $t \in OQd_2 = EQd_2$; so there exists a $t'$ such that $t = TARG_{i1}(t')$ and $t' \in TQd_{2/11}$ and from the definition of $TEi_1$ we see that $TARG_{i1}(t')$ was never inserted. so $t = TARG_{i1}(t') \notin R_{TS}$.

QED

Lemma 4-6: $U_{TS} \subseteq U_{OS}$

Proof: $TEi_1 \subseteq OEi_1$ directly follows from definition. If there is a tuple $t \in TEi_2$ such that $t \notin OEi_2$ then it follows that $t \in R$ and $t \notin R \cup OEd_{11} \cup OEi_1$; hence, $t \notin OEi_1$. From the definition of $TEi_2$ we can conclude that $t \notin OEd_1$, which is a contradiction. Therefore, $TEi_2 \subseteq OEi_2$. Finally, for some tuple $t \in R$, let $TARG_{i2}(TARG_{i1}(t)) \subseteq TARG_{i3}(TEi_3)$. From the definition we see that $t \in TEi_3$; that is $t \in TQ_{i1} \cap TQ_{i2}$. $t \in TQ_{i1}$ implies that there exists a tuple $t'$ such that $TARG_{i1}(t) = t'$ and $t'$ satisfies $q_{i2}$; i.e. $t' \in TQ-(i2)$. As $t \notin OQ_{i1}$, we can infer that $TARG_{i1}(t) \in (R \cup OEd_1) \cup TARG_{i1}(OEi_1)$. Therefore, $TARG_{i1}(t) \notin OEi_2$ since $TARG_{i1}(t) = t'$ satisfies $q_{i2}$. This implies that $e_{i2}$ inserts a tuple $TARG_{i2}(t') = TARG_{i2}(TARG_{i1}(t))$; that is, $TARG_{i2}(TARG_{i1}(t)) \in RO_{OS}$. Hence, $TARG_{i3}(TEi_3) \subseteq TARG_{i1}(OEi_1) \cup TARG_{i2}(OEi_2)$.

The above inferences imply that $TARG_{i1}(TEi_1) \cup TARG_{i2}(TEi_2) \cup TARG_{i3}(TEi_3) \subseteq TARG_{i1}(OEi_1) \cup TARG_{i2}(OEi_2)$; i.e. $U_{TS} \subseteq U_{OS}$.

QED

Lemma 4-7: $(U_{TS \cup R_{TS}}) \subseteq (U_{OS \cup R_{OS}})$.

Proof: Assume that there exist a tuple $t' \in U_{TS \cup R_{TS}}$ such that $t' \notin U_{OS \cup R_{OS}}$. From lemma 4-6 we see that $t' \in U_{TS}$ implies that $t' \notin U_{OS}$. So it follows that $t' \notin U_{OS \cup R_{OS}}$. From lemma 4-2 and lemma 4-3 we see that $t' \notin U_{OS \cup R_{OS}}$. Hence, it follows that $t' \notin U_{OS \cup R_{OS}}$. QED