Concurrency coordination in a locally distributed database system*

by GRUIA-CATALIN ROMAN
Department of Computer Science
Washington University
St. Louis, Missouri

INTRODUCTION

A database is called internally consistent (or just consistent) with respect to a given set of invariant properties if, in the absence of any database activity, the above said invariants hold true. These properties reflect relations among objects of the application domain and characterize its semantic consistency (e.g., "last name appears first on all government forms"). Because a single database may support many different applications, these relations are generally assumed to be known solely by the database user and not by the database designer. The very definition of a transaction (a sequence of primitive database activities which, when acting alone, preserves, as a whole, all invariants) is in recognition of the fact that it is the user's duty to guarantee the integrity of the data he manipulates.

While the assumption that during the execution of one transaction no other transaction exists in the system is reasonable from the user viewpoint, it is not acceptable from a design perspective since it would preclude any concurrency. It is the designer's task to assure the preservation of the user viewpoint while maximizing performance through the use of concurrency. The problem of developing concurrency control or access synchronization algorithms is perhaps the most difficult issue facing the database designer today. On one hand, while each transaction by itself preserves database consistency, concurrent execution of several transactions may ultimately result in: (1) the violation of the invariants persisting even after termination of all involved transactions and (2) the reading of inconsistent data by some of the transactions. On the other hand, excessive overhead in propagating or waiting for concurrency control messages can seriously degrade database performance. Starting in the early seventies and more extensively during the last three years, concurrency coordination has received considerable attention among database research groups. The very first approach to be considered for the synchronization of concurrent updates in distributed databases was the use of locks (shared and exclusive). However, locking, which was suitable in centralized databases, proved to be very inadequate for geographically distributed databases due to extreme communication delays.

Consequently, efforts have been made toward the development of more effective coordination schemes which reduce the number of messages required to propagate locking information, e.g., Thomas and Ellis. The savings gained by these approaches are, however, insufficient when a large number of sites and high transaction volume are involved. Other authors attempted to bring forth improvements through the incorporation of a minimal level of centralization with a distributed system, e.g., Alsberg and Day, Stonebraker and Neuhold. The performance of such algorithms depends upon the degree to which a proper partitioning of database activities is achievable.

Still another strategy was adopted by Bernstein, et al. This method takes advantage of an a priori classification of transactions based upon the different nature of their interactions. In this manner locking efficiency is achieved by avoiding unnecessary global locking when possible.

A "best" algorithm has not yet been found, and the search continues. This paper is concerned with highly distributed local databases anticipated to emerge due to the advent of VLSI technology. A pipelined architecture for a multi-processor database system is proposed along with a concurrency coordination scheme. The architecture can be modeled as a collection of single-rooted directed acyclic graphs (DAG's). The root of each DAG is associated with a column of a cross-bar switch; the rows correspond to separate user groups. Each node of the DAG represents a processor, and each arc indicates a communication link between processors. The leaves of the DAG's are called data processors and may contain single files or larger portions of the database. The other nodes, called directory processors are assumed to store, in a distributed fashion, an index structure designed to maximize throughput while maintaining a relatively constant response time. For the sake of simplifying the exposition, changes to the index structure will not be considered here.

Transactions are entered serially at the root of each DAG and the concurrency coordination mechanism enforces their "proper" pipelining through the networks. Each directory

* This work has been supported in part by the Division of Research Resources of the National Institutes of Health under Grant RR 0396.
CONCURRENCY COORDINATION

It has already been stated that concurrent execution of several transactions can result in violations of database consistency due to undesirable interferences among transactions. The network organization suggested in this paper adds a new level of complexity since each transaction is present concurrently in many nodes of the DAG. Thus, concurrency coordination is required even when a single transaction exists in the system. The coordination scheme described below handles uniformly both types of situations. Furthermore, it assumes no centralization, is simple, and involves a small synchronization overhead. However, in contrast to previously referenced approaches, it is architecture specific.

The idea behind the scheme is the following. Given any two data processors, \( X_p \) and \( X_q \), for any two transactions \( T_i \) and \( T_j \) which access (read or write) data from both \( X_p \) and \( X_q \), the two processors are forced to see \( T_i \) and \( T_j \) in the same order. This can be accomplished by requiring the behavior of every node to be describable by the functions below:

\[ (S) \text{ is the set of all possible serial schedules} \]

**MATCH:**
\[ S^* \rightarrow S \]
\[ SO = MATCH(S_1, S_2, ..., S_n) \]
where
\[ (SO)/l \leq T_i \iff (S_j)/l \leq T_i \quad \text{for } j = 1, 2, ..., n \]
\[ (SO)/l = \{ Q_i \text{ if } \exists k: (S_k)/l = Q_i \} \]
\[ \quad \text{if } i \neq j \]
\[ (SO)/l = i \quad \text{for } j = 1, 2, ..., n \]

**TRIM:**
\[ S^n \times S^n \rightarrow S^n \]
\[ (S_1', S_2', ..., S_n') = TRIM(S_1, S_2, ..., S_n; SO) \]
where for \( j = 1, 2, ..., n \)
\[ S_j = S_j' \]
\[ (S'_j)/l \leq T_i \iff (SO)/l \leq T_i \]

**FORK:**
\[ S \rightarrow S^n \]
\[ (S_1, S_2, ..., S_m) = FORK(SO) \]
where for \( j = 1, 2, ..., m \)
\[ (S_j)/l = \{ Q_i \text{ or } i \text{ if } (SO)/l \leq T_i \text{ and } (SO)/l = i \} \]
\[ \quad \text{if } (SO)/l = i \]

Each node \( X_u \) processes some transaction \( T_i \) if and only if it receives a message associated with \( T_i \) from all its predecessors in the DAG. Some of the messages may be null, some may represent update requests, and others may be read requests. \( X_u \) (conceptually) combines all messages into a single one. The MATCH function simulates precisely this action while TRIM is used to express the removal of the respective messages from all input queues. It is essential for \( X_u \) to wait until all messages associated with \( T_i \) are received; otherwise, improper transaction processing and destruction of the serialization would occur.

The interpretation of the combined query, built on behalf of the transaction \( T_i \), results in subqueries (associated with \( T_i \)) to be sent to some of the successors of \( X_u \). However, in order to assure correct propagation of the schedule, all

---

**DEFINITION OF TERMS**

The terms "database," "transaction" and "consistency" are used as defined by Gray.9 Each transaction is assumed to have three phases:

1. **Data Selection Phase**—A selection criterion or query is passed to the root nodes of the networks (DAG’s). Each root node distributes the selection process among its successors by initiating appropriate subqueries. Later, when the answers to the subqueries return, it concatenates and sends them to the user along with an end-of-selection signal. All other nodes repeat the same scenario throughout the network.

2. **Data Modification Phase**—Upon receipt of the required data, the necessary updating is carried out.

3. **Data Commitment Phase**—The updated data is sent back into the network to permanently replace the old copies. As soon as all updates are acknowledged, a final commit message is issued by the user.

Some notation is introduced next as a necessary tool for enabling a formal treatment of the special case where a single DAG is present. The extension to multiple networks is made, in an informal manner, later on.

- \( X_z \) denotes a single node (data or directory processor) in the network. \( X_O \) is used to represent the root node.
- \( T_i \) represents the transaction \( i \). Each transaction is assumed to have a unique identifier \( i \) (e.g., distinct time stamps). All messages exchanged by processors in behalf of \( T_i \) carry the identifier \( i \).
- \( Q_i \) denotes any non-null message associated with the transaction \( T_i \) during the data selection phase; it is called a query message. The notation \( Q_i \leq T_i \) means \( i = j \).
- \( i \) is used to represent a null query message associated with \( T_i \).
- \( S_k \) is called a schedule and is defined as an arbitrary sequence of query messages. \( S_k \) is a serial schedule iff no two query messages in \( S_k \) have the same identifier, i.e.,
  \[ (S_k)/l \leq T_i \text{ and } (S_k)/q \leq T_j \quad \text{and} \quad p \neq q \quad \Rightarrow \quad i \neq j \]
  where \( (S_k)/r \) is the \( r^{th} \) query message in \( S_k \).
- A period denotes schedule concatenation.
**DATABASE CONFIGURATION**

**PROCESSING OF THE SCHEDULE T1,T2**

**ENTRY:**
(T1 : add 2 mod 6 to one-digit numbers). (T2 : list even numbers) Q1.Q2

**ODD:**
(T1 : nil). (T2 : nil) 1.1

**EVEN:**
(T1 : nil). (T2 : list even numbers) 1.Q2

**ONE-DIGIT:**
(T1 : add 2 mod 6 to one-digit numbers). (T2 : nil) Q1.2

**TWO-DIGIT:**
(T1 : nil). (T2 : nil) 1.2

#1:
(T1 : nil). (T2 : nil)
= (T1 : add 2 mod 6). (T2 : nil) Q1.2
(T1 : add 2 mod 6). (T2 : nil)

#6:
(T1 : nil). (T2 : list)
= (T1 : add 2 mod 6). (T2 : list) Q1.Q2
(T1 : add 2 mod 6). (T2 : nil)

#11:
(T1 : nil). (T2 : nil)
= (T1 : nil). (T2 : nil) 1.2
(T1 : nil). (T2 : nil)

#18:
(T1 : nil). (T2 : list)
= (T1 : nil). (T2 : list) 1.Q2
(T1 : nil). (T2 : nil)

*Figure 1—Sample schedule processing*
other successors of $X_u$ also receive messages on behalf of $T_i$. These messages are null messages needed solely for concurrency coordination purposes. The function FORK is non-deterministic and simulates the process by which $X_u$ decides what query messages to send to its successors as a consequence of processing a combined query message generated by MATCH. Since each node processes and puts out messages in the same order, the initial total order over the transactions $T_i$ is maintained throughout the network.

**Proposition**

Given the fact that the entry node $X_0$ is presented with a serial schedule $S_0$, every node $X_u$ will execute an order equivalent serial schedule $S_u$:

$(S_u) \preceq (S_0) \iff (S_0) \preceq (S_u)$.

**Proof by induction**

(O) The proposition holds trivially for $X_0$

MATCH($S_0$) = $S_0$

TRIM($S_0$) = $\emptyset$

FORK($S_0$) = ($(S_0, S_0, ..., S_m)$) — where $S_k$ represents the schedule generated by $S_0$ for its $k^{th}$ successor and is order equivalent to $S_0$.

(N) Let us assume the proposition to be true for nodes at distance $N$ from $X_0$. Distance is defined as the length (number of links) of the longest path from $X_0$ to the particular node.

(N + 1) Given any node $X_u$ at distance $N + 1$, due to assumption (N) all its input schedules are order equivalent to $S_0$. By applying again the definitions of MATCH, TRIM, and FORK, one establishes the proposition to be true for the level $N + 1$.

Figure 1 describes a database and the sample schedules seen by each node as a result of processing two transactions, $T_1$ and $T_2$. The reader should note, however, that in the example of Figure 1, the updates are carried out at the nodes. If node #6, for instance, sends data to the entry point to be updated by the user processor that initiated the particular transaction, node #6 will not process the next message until the result comes back and is committed throughout the network.

The extension of this coordination scheme from one to several networks requires that all root nodes receive transactions spanning across nets in the same order. This can be achieved through the use of a hardware cross-bar switch, Figure 2. When a user terminal issues a transaction which concerns only one of the database networks, the query message transmission takes place as soon as the vertical path becomes available. However, if the query involves more than one network, the user must request allocation of all needed paths before sending query messages to the various networks. This strategy will guarantee that the separate serial schedules are all consistent with each other, i.e., any two transactions occur in the same order in any serial schedule in which they appear together. Furthermore, if switching paths are viewed as resources and always allocated in the same order, the possibility of deadlock is eliminated. At the same time, blocking within the switch can be minimized by allowing single network users to use paths which have been allocated to a multiple network user but are not yet in use.

**PERFORMANCE ISSUES**

The architectural solution described above has its justification in the application domain for which it is intended — medical information systems. Such systems tend to be confined to a single geographical location, grow relatively fast, require quick response, and exhibit a processing pattern dominated by data retrieval and creation rather than updates. As such, modifications of the directories, other than additions of new entries, can be assumed to be few. Therefore, the user should be willing, in those rare occasions, to pay an additional waiting penalty for coordinating the concurrent update of several directories.

The distribution of the index structure over several directory processors is meant to reduce the searching time through the use of concurrency in a pipelined-like fashion. The goal is to assure a good average time response through the addition of new directory and data processors when faced with transaction volume increases. However, the system's ability to handle the higher throughput relates not only to the number of processors being used but also to the "appropriateness" of the data and index distribution. Ideally, all data processors should be equally utilized. Furthermore, the searching load within each net should be equally distributed among directory processors at equal distance from the root since the coordination scheme forces each processor to work at the rate of the slowest predecessor.

With respect to transaction recovery and roll-back, a transaction failure in some node could be signaled by passing a failure message to the user processor, which, in turn would
send a "forget about my updates" message in place of the commit. Subsequently, the transactions would be started again with a new identifier. A node failure would have the effect of cancelling any transaction that requires its use.

SUMMARY

An architecture for a locally distributed database system was suggested. A simple solution to the problem of coordinating concurrent transactions within the database was presented. The solution requires no centralized control, is deadlock free, uses no locks, is fair, and involves little overhead.

REFERENCES
