Single-Site and Distributed Optimistic Protocols for Concurrency Control

M. A. BASSIOUNI

Abstract—**In spite of their advantage in removing the overhead of lock maintenance and deadlock handling, optimistic concurrency control methods have continued to be far less popular in practice than locking schemes. There are two complementary approaches to help render the optimistic approach practically viable. For the high-level approach, integration schemes can be utilized so that the database management system is provided with a variety of synchronization methods each of which can be applied to the appropriate class of transactions. The low-level approach seeks to increase the concurrency of the original optimistic method and improve its performance. In this paper we examine the latter approach, and present algorithms that aim at reducing backups and improve throughput. Both the single-site and distributed networks are considered. Optimistic schemes using timestamps for fully duplicated and partially duplicated database networks are presented, with emphasis on performance enhancement and on reducing the overall cost of implementation.**

Index Terms—**Centralized database systems, distributed database systems, optimistic concurrency control, performance evaluation of database systems, serializable schedules.**

I. Introduction

THE problem of designing and implementing concurrency control protocols is extremely important and very crucial to the performance of database systems. No single solution can perform well under all circumstances; each approach has its pros and cons and its domain of environment types in which it is most successful. Further research is still needed to design new methods, improve and increase the scope of applicability of current methods, and compare and integrate the different approaches. This paper addresses the problem of improving optimistic concurrency control methods, enhancing their performance, and alleviating the problems associated with the optimistic approach.

Traditionally, *locking* mechanisms have been used to achieve control of concurrent transactions [3], [4], [14], [18], [24]. Before accessing any object, a transaction must acquire a lock for that object. If the request of the transaction to lock certain object is granted (possibly after a waiting period), certain other transactions may be denied access to this object until the lock is released. Locks may be of different types (e.g., read locks and write locks), and they must be requested and released according to certain protocols so as to ensure that serializability is maintained. Usually, locking schemes are not deadlock-free, and therefore must be augmented with mechanisms for detecting and solving deadlock situations. The locking approach has been used both for single-site and distributed databases. A central controller (site) may be used in distributed systems to handle and grant locks for all objects in the distributed database. Another scheme is to use *timestamps* [5], [11], [18], [25] to determine the order of application of concurrent updates. A timestamp is created by appending the clock time (when the transaction is submitted) with the Id of the site at which the transaction originated; thus each transaction in the system has a unique timestamp. A more recent approach to concurrency control is the *optimistic* synchronization method [7], [15], [22], [23], [2]. Optimistic methods use a different philosophy from that of locking and timestamp algorithms. Transactions are allowed to proceed without any synchronization; but updates are made on local copies of objects. At the end of execution, a validation is performed to examine whether conflicts may have occurred (i.e., whether the transaction may have used a database state that lacks consistency and hence violates serializability). If a conflict is detected, the transaction is backed up and is submitted at a later time; otherwise the transaction is committed and the local updates are made global. Optimistic methods are suitable in environments in which conflicts are rare (e.g., query dominant systems). If applied to the appropriate environment, optimistic methods can lead to more concurrency in addition to removing the overhead of lock maintenance and deadlock handling.

Although locking mechanisms have been more popular, we share the opinion expressed in [8] that optimistic methods should not be discouraged. Optimistic methods can be more useful than other concurrency control schemes for workloads having low conflict rate. Database systems vary widely in workload and application types, and certain synchronization method might be more suitable than others for a given environment. We believe that there are two complementary steps through which the optimistic approach can gain practical success:

1) The low level approach aims at improving the original optimistic concurrency control method through increasing its scope of applicability and reducing its rate of backups.

2) The high level approach uses integration of various concurrency control methods (including optimistic meth-

Manuscript received February 15, 1986; revised June 30, 1987.

The author is with the Department of Computer Science, University of Central Florida, Orlando, FL 32816.

IEEE Log Number 8822022.

ods) in the same database system with the ability to apply each method to the appropriate class of transactions and ensure that conflicts among the different classes are resolved correctly [8], [17].

In this paper, we concentrate on the low-level approach: improving the performance of the optimistic method. We present a concurrency control method that is based on the optimistic approach and the concept of timestamps. The scheme increases the concurrency and widens the domain of applicability of optimistic methods. Both single-site and distributed methods are presented.

II. Notation

A transaction is a computational process that maps the database from one consistent state to another consistent state (could be the same state). The state of the database is determined by the values of the objects in it. Each object has a unique Id. During execution, a transaction reads a set of certain objects and may need to update another set of objects. We use the notation $RS(T)$ and $WS(T)$ to denote the read-set and the write-set of transaction T, respectively. The set of objects in $RS(T)$ or $WS(T)$ are not required to be known in advance and may in fact be data dependent. In order to finish, each transaction goes through three phases (two for read-only transactions). These phases are in the following order:

1) Read Phase: The transaction executes and updates are made in local copies. The sets $RS(T)$ and $WS(T)$ are recorded for each transaction T. Each object in $RS(T)$ is read only once (i.e., local copy of the object is used if the transaction reads the object several times).

2) Validation Phase: The concurrency control algorithm will determine if the transaction might have seen the database in an inconsistent state (e.g., whether its read-set might have been partially updated by another transaction). The algorithm aborts any transaction that, because of detected conflicts, may violate the serializability requirement.

3) Committing Updates: If no conflict is detected, the transaction is committed and its updates are made global.

A. Critical Sections

Concurrency control protocols based on the optimistic approach usually use the concept of *critical sections*, and the function and correctness of these protocols are often implicitly based on the proper synchronization provided by the critical section constructs. Because of their important role in optimistic protocols, the notation and definition of critical sections (as used in this paper) are briefly discussed next.

Critical sections will be denoted by double angular brackets. Within a single-site system, critical sections indicate that at most one transaction can be executing in the code of any of these sections at a time. Fore example, the sequence

$$C_1;\ \ll C_2 \gg;\ C_3;\ C_4;\ \ll C_5 \gg$$

has C_2 and C_5 as critical sections. The sequence can be executed concurrently by many transactions provided that if any transaction is executing C_2 (or C_5), no other transaction is allowed to execute the code of C_5 or the code of C_2. In practice, implementation of critical sections [19] can be achieved by several mechanisms, e.g., disabling interrupts, semaphores, monitors, etc. It is also possible to have classes of critical sections (each class represents a separate mutual exclusion requirement), but this facility will not be needed in this paper.

Properly designed optimistic concurrency control protocols for distributed database networks should be based on the assumption that the enforcement of synchronization for critical sections is done on a single-site basis (i.e., global synchronization among the different sites is not possible). Thus if the previous sequence is used within a global distributed protocol, the mutual exclusion requirement of critical sections will be enforced by each site individually. This means that execution of C_2 by a transaction in one site does not prevent the execution of C_2 or C_5 by a transaction in a different site.

III. The Optimistic Approach and Proposed Enhancement

Optimistic methods are mainly suitable for environments where conflicts are rare, thus making backups infrequent. Environments of this type usually have large number of objects, transactions usually manipulate a small subset of the database, and the majority of transactions are read-only queries.

The optimistic approach was first introduced in [15] for the single-site model. In [23], another variation of optimistic concurrency control is presented: in the single-site model, priority is given to read-only transactions so that they are not subject to backups. Write transactions can be backed up and may be made to wait until certain read transactions finish. The distributed model assumes nonredundant data network (i.e., multiple copies of objects are not allowed) and both read and write transactions can be backed up. In [22], the optimistic approach is used for the development of concurrency control with emphasis on solving the phantom problem [14]. Other work on optimistic concurrency control and its performance appeared in [7], [21].

In this paper, we present an optimistic concurrency control algorithm that uses timestamps. In the case of single-site, timestamps will be simply generated by a global counter. For the distributed model, timestamps are obtained by concatenating the clock time with the Id of the site. Compared to previous optimistic approaches [15], [22], [23], the algorithm presented here has the following characteristics:

- Write-write conflicts do not lead to backups and are resolved by the timestamp concept. This allows more concurrency and makes the optimistic approach more applicable to environments with a higher fraction of update transactions. It should be noted here that the use of

timestamps is a well-known method that has been utilized in concurrency control schemes (especially for writing objects using the Thomas update rule). However, the combined use of timestamps and the philosophy of the optimistic method in the manner described in this paper represents a new approach that enhances the practical significance of the optimistic method.

- Only writing one object at a time is needed to be in a critical section (rather than the entire write phase). Thus transactions could be concurrently writing objects in their write-sets even if they have write-write conflicts (the only requirement is that writing one object is an indivisible operation). Notice that in [15] two schemes were presented: the first method requires the entire write-phase to be in a critical section, and the second scheme removes this requirement by aborting any transaction that has a write-write conflict with a transaction that is still writing. Since write transactions are the main cause of rollbacks, reducing backups of write-transactions will have a positive effect on read-transactions and the performance of the system. We believe that this improvement represents an important step towards making the implementation of optimistic concurrency control practically acceptable.
- One of the problems of the optimistic concurrency control method is that it requires maintaining certain information about each transaction even after the transaction is committed. The proposed scheme uses a simple mechanism of reference counts to answer the question of how long the set $WS(T)$ should be retained after T is committed. Therefore, unlike the schemes in [15], no transaction needs to be backed up because of missing information. Also, the scheme reduces the probability of invalidating a transaction T by a write-transaction even though the latter transaction is itself invalidated.
- Another problem with the optimistic approach is its discrimination against transactions having long read phase (because of higher probability of backup). The proposed scheme alleviates the severity of this problem in two ways. First, reducing backups of writers (which are the cause of rollbacks of readers) creates an environment in which transactions with long read phase would have a better chance for not being interfered with. Second, timestamps are used to resolve read-write conflicts in which the reader has read a consistent state of the database (details are given in Section IV-B). Thus it is possible for a long read-transaction to finish execution correctly in spite of the presence of conflicting write-transactions (which would otherwise cause the reader to be invalidated). We believe that these improvements integrated with a high level approach as described in [8] will practically eliminate discrimination against long transactions.
- The scheme does not require affixing a permanent timestamp to each object in the database. Timestamps are created in main memory only for those objects accessed by current active transactions (details will be given later). Thus although our method uses timestamps, it does not require changing the scheme of the database. Using main memory to store the timestamps of the subset of objects needed by active transactions can be an affordable overhead. This approach is encouraged by the fact that main memory keeps getting cheaper in price and larger in size. Some research results, e.g., [13] are even based on the assumption that, in the near future, it will be possible to store entire large databases in main memory.
- An efficient scheme is used in the distributed case to avoid the high cost normally associated with protocols used to synchronize clocks at all sites in order to ensure that timestamps work properly.
- Because of the above properties, the scheme can be used for a wider scope of databases and transaction loads. It can tolerate a higher percentage of write-transactions and of conflicts among transactions.

In addition, the proposed scheme inherits the following advantages of optimistic methods.

- Objects may be accessed by any transaction at any time, i.e., in contrast to locking, transactions may not be denied access to certain objects at any time.
- The overhead of detecting and handling deadlock is eliminated.
- The scheme does not require prior knowledge of the read and write sets.

IV. Protocol for the Single Site Case

As in other optimistic methods, the protocol of the single site algorithm requires each transaction to execute a sequence of steps of the form:

transbegin; read phase; transend & validation; commit or backup

In *transbegin*, when the transaction is submitted, it is assigned a number T which acts as the Id of this transaction (alternatively the system Id, if any, of the transaction can be used to replace T). At the end of the read phase (i.e., in *transend*) another number, $ts(T)$, is assigned to the transaction and is used as its timestamp. The system maintains two global sets:

- *active* is the set of transactions that have finished the read-phase but have neither been committed or backed up, and
- *submitted* is the set of transactions which have been submitted but are still in the read-phase.

For each transaction, the following sets and variables are needed:

- $RS(T)$ is the read set of transaction T
- $WS(T)$ is the write set of transaction T
- $c\_active(T)$ is a copy of *active* maintained for transaction T
- $ref\_count(T)$ is the number of transactions that can be affected by a conflict with T.

The set $WS(T)$ and the value of $ts(T)$ need to be retained until each transaction implied by $ref\_count(T)$ validates.

A. The Basic Algorithm

The basic concurrency control algorithm is given below. The notation $|$ *submitted* $|$ denotes the number of ele-

ments in the set *submitted*. Further optimization details will be discussed in Section IV-B.

```
<<Id_count := Id_count + 1; T := Id_count;
  for each t ∈ active do
      ref_count(t) := ref_count(t) + 1;
  c_active(T) := active; ref_count(T) := 0;
  RS(T) := WS(T) := φ;
  submitted := submitted ∪ {T} >>
```

read-phase

```
<< submitted := submitted − {T};
  for each t ∈ submitted do
      c_active(t) := c_active(t) ∪ {T};
  ref_count(T) := ref_count(T) + | submitted |;
  active := active ∪ {T};
  ts_count := ts_count + 1; ts(T) := ts_count >>

  valid := True;
  for all t ∈ c_active(T) do
      if RS(T) ∩ WS(t) ≠ φ then valid := False;
  if valid then  /* commit */
    begin
      for each x ∈ WS(T) do
        << if ts(T) > timestamp(x) then
            [write x; timestamp(x) := ts(T)] >>
      << active := active − {T} >>
      if ref_count(T) = 0 then discard WS(T);
    end
  else  /* abort */
    begin
      << active := active − {T} >>
          if ref_count(T) = 0 then discard WS(T)
          else WS(T) := φ;
    end
  for each t ∈ c_active(T) do
      << ref_count(t) := ref_count(t) − 1;
          if ref_count(t) = 0 then discard WS(t) >>
  if valid then cleanup else backup
```

As can be seen from the above algorithm, write-write conflicts are resolved by the timestamp $ts(T)$ assigned to each transaction T at the beginning of validation. If two transactions T_1 and T_2 are concurrently in validation, with say $ts(T_1) < ts(T_2)$, then $T_1 \in c\_active(T_2)$, and T_2 is validated only if it does not conflict with T_1, i.e., if $RS(T_2) \cap WS(T_1)$ is empty. If there is no conflict, both transactions can be writing objects concurrently. If T_2 writes an object before T_1 does, then T_1 will ignore writing that object so that the final state is equivalent to the serial execution of T_1 followed by T_2.

Correctness Proof: It is easy to see that serializability [14] is maintained by the above algorithm. We give only a brief outline for the different cases of two transactions T_1 and T_2.

1) T_1 and T_2 are concurrently in their respective validation phases, say T_1 started validation first, i.e., $ts(T_1) < ts(T_2)$. Then, as mentioned above, T_1 is a member of $c\_active(T_2)$ and T_2 can validate only if its read-set does not intersect with the write-set of T_1. This is necessary since in the serial schedule $T_1 T_2$, transaction T_2 must read objects after they are updated by T_1. Thus if both transactions validate successfully, the final state of the database will correspond to the serial execution of T_1 followed by T_2.

2) T_1 finishes validation while T_2 is still in the read phase, moroever, T_1 started validation before T_2 was submitted. It is easy to see that when T_2 starts validation, its set $c\_active(T_2)$ contains T_1. The set $WS(T_1)$ will be retained by the system (since $ref\_count(T_1) > 0$) until T_2 is validated. Therefore T_2 is committed only if it does not have a conflict with T_1.

3) T_1 finishes validation while T_2 is still in its read phase; moreover, T_1 started validation after T_2 was submitted. We have two cases: T_1 was submitted before T_2 was submitted (i.e., $T_1 < T_2$) or T_2 was submitted first (i.e., $T_2 < T_1$). However, in both cases, when T_1 starts validation, the set *submitted* contains T_2. Hence T_1 is inserted into $c\_active(T_2)$. As before $WS(T_1)$ is retained until T_2 finishes validation. Thus T_2 validates correctly.

4) T_1 finishes validation before T_2 is submitted. Then T_2 gets to see a consistent database state after it is updated by T_1.

Notice that timestamps for objects in the above scheme do not have to be permanently attached to the database; they are created only in main memory for the subset of objects currently being used. When an object (that does not have a timestamp in memory) gets accessed, a timestamp for that object is created in main memory and is initialized to -1. The timestamp is kept in memory for as much as the given transaction and other concurrent or subsequent transactions require. Reference counts can be used to determine the duration required to keep the timestamp for any object.

The problem that a write-transaction t in $c\_active(T)$ may result in the invalidation of T even though the transaction t is itself invalidated, has been recognized in [15], and a partial solution was presented using several stages of preliminary validation. Another partial solution has been presented in the above scheme: when transaction T gets invalidated, the set $WS(T)$ is immediately set to null, so that other transactions will not get invalidated because of T. Notice also that the problem of how long the set $WS(T)$ should be maintained (after T is committed) is solved by using $ref\_count(T)$. Thus backing up transactions because of the destruction of information about other transactions [15] is completely eliminated.

B. Further Optimization

In the previous algorithm, transaction T is backed up if $RS(T) \cap WS(t)$ is not empty. It is possible however that T did read all objects in $RS(T) \cap WS(t)$ after transaction t updated them. Transaction T gets aborted although it has read a consistent state (as far as transaction t is concerned). The above problem can be solved by storing local copies of the timestamps of objects accessed by each transaction. If $c\_timestamp(T \mid x)$ denotes the copy of *time-*

stamp(x) stored locally by T, then the code to check for read-write conflicts (after finishing the read phase) becomes:

```
<< submitted := submitted − {T};
   for each t ∈ submitted do
      c_active(t) := c_active(t) ∪ {T};
   ref_count(T) := ref_count(T) + | submitted |;
   active := active ∪ {T};
   ts_count := ts_count + 1; ts(T) := ts_count >>
   valid := True;
   for all t ∈ c_active(T) do
      for all x ∈ RS(T) ∩ WS(t) do
         if c_timestamp(T | x) < ts(t)
         then valid := False;
   if valid then commit
   else abort
```

Notice that the above modification will validate T correctly for any number of transactions in $c\_active(T)$.

V. Performance Results

In this section, we describe the simulation model which has been used to compare the performance of our proposed algorithm and the optimistic algorithm described in [15]. All the numerous test cases conducted have shown that the proposed algorithm considerably reduces the number of transactions' backups and hence gives a consistent improvement in system's throughput.

Fig. 1 gives a high-level block-diagram of the closed queueing system used as the simulation model. This queueing model is similar to that used in [9], [10] to compare the performance of several concurrency control methods. The results reported here are for the basic scheme without the optimization explained in Section IV-B. This optimization is expected to further enhance the performance of the proposed scheme.

Initially, a fixed number of transactions are generated and put into the start-up queue. These transactions continually cycle around the model. When a transaction enters the start-up queue, it is assigned a workload consisting of a read set and a write set (read only transactions are assigned null write sets). Transactions then go through the following stages:

- Transactions leaving the start-up queue are placed at the end of the concurrency control queue (cc-queue). These transactions make the request to start the read phase (as new transactions) and the concurrency controller performs the *transbegin* phase for each transaction. The transaction is then placed at the end of the database queue (db-queue).
- Transactions in the db-queue make requests to access the required objects, one at a time. Thus a transaction keeps cycling in the db-queue until it finishes the read phase, and is then placed at the end of the cc-queue to start validation.
- After the validation phase, transactions are either placed in the db-queue (case of successful validation of writers), in the start-up queue as new transactions (case of successful validation of readers), or in the start-up queue as aborted transactions (case of unsuccessful validation). Successfully validated writers join the db-queue to write the objects in their write-set, then they are returned to the cc-queue to finish the commit protocol, and finally to the start-up queue to initiate new transactions.

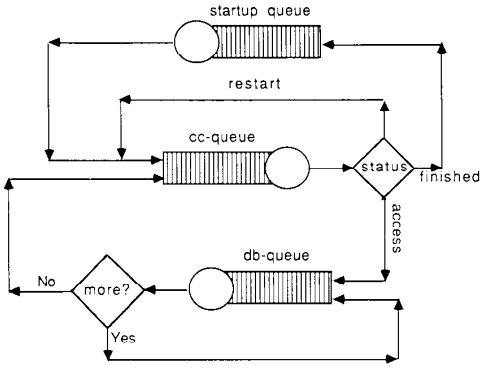


Fig. 1. The closed queueing model.

The scheduling policy in each of the logical servers shown in Fig. 1 is assumed to be FCFS. Transactions are divided into two classes: large or small. The distribution of the size of the read sets is exponential with a mean value that depends on the class of the transaction. Similarly, two exponential distributions with different mean values are used to generate the sizes of the write sets for large and small write-transactions. The simulation system allows control over the values of the following parameters.

- The level of multiprogramming, the ratio of the number of readers to the number of writers, and the ratio of the number of small transactions to the number of large transactions.
- The size of the database (total number of objects) and the mean values for the size distributions of the different classes of transactions.
- The CPU and I/O average times required to process requests in the different logical queues.
- The probability of write-write conflicts among writers.

The simulation system is written in Concurrent Euclid (subprograms to generate the workload and other random numbers are written in C). The system consists of six procedures, five monitors, and five processes. A large number of test cases were conducted to cover a variety of system's parameters. Throughput results and their 90 percent confidence interval estimates were obtained using the batch method as was done in [9]. Each single throughput result in our tests was obtained by a batch of 10 runs; each run is 100 000 units of simulated time long. We will first give the results of representative test cases, then we discuss the conclusions of this simulation study.

Representative test cases for a number of system's parameters are shown in Figs. 2–5. Fig. 2 shows a typical

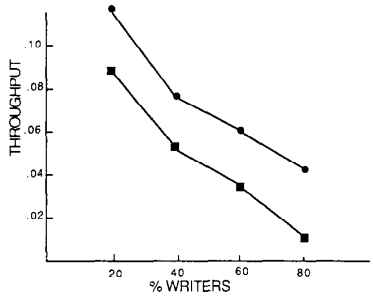


Fig. 2. Throughput versus percent writers.

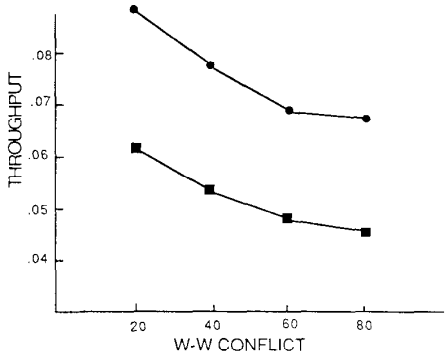


Fig. 3. Throughput versus write-write conflict.

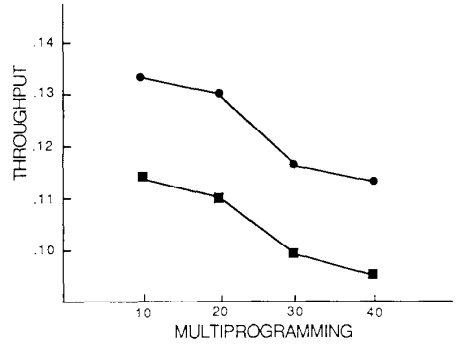


Fig. 4. Throughput versus multiprogramming.

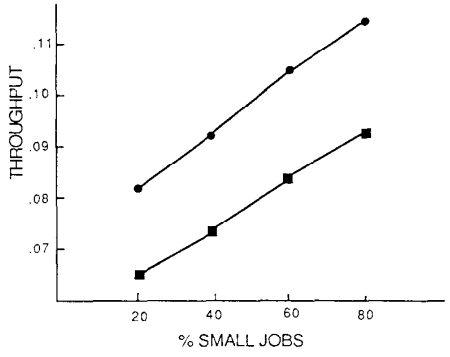


Fig. 5. Throughput versus percent small jobs.

relationship between the throughput of the system (number of finished jobs per second) and the percentage of write transactions. All points in this figure were obtained for a level of multiprogramming of 50, database size of 5000 objects, 90 percent small transactions, and 40 percent write-write conflict (i.e., for a given writer, 40 percent of other writers are expected to have some write-write conflict with it). The performance of proposed algorithm is given by the line with circles and that of Kung's algorithm is given by the line with squares. The relative improvement in throughput for the proposed algorithm increased from 34.1 to 305.7 percent when the percentage of writers increased from 20 to 80 percent.

Fig. 3 shows a typical result for the relationship between the percentage of conflicting writers and throughput. The percentage of writers was fixed at 40 percent of the total workload. All other parameters are the same as those used in Fig. 2. Fig. 4 shows a typical relationship between the degree of multiprogramming and the throughput of the system. The workload always consisted of 30 percent writers, 90 percent small transactions, and 30 percent write-write conflict. The size of the database was 5000.

Fig. 5 shows the relationship between throughput versus the percentage of small transactions. The effect of other system's parameters was also monitored and the proposed algorithm was observed to give consistent improvement in throughput. The rate of rollbacks was also recorded for each test run. The reduction in the rate of rollbacks showed trends similar to those of the improvement in throughput. The proposed algorithm improved throughput by well above 30 percent in most of the test runs. The optimization of Section IV-B is expected to further enhance the performance of this algorithm.

VI. Distributed Databases with Full Duplication and Synchronized Clocks

In this and the following sections, we develop optimistic methods for fully replicated distributed networks and for distributed networks with partial replication. We start by extending the single-site algorithm presented before to the case of fully duplicated distributed databases. For each transaction T, three timestamps are used:

- *start_ts*(T): is the *begin_timestamp*, i.e., the time when T is submitted. This timestamp is used only by the site where T originates.
- *validate_ts*(T): is the *validation_timestamp*, i.e., the time when T finishes the read-phase and starts validation. This timestamp is transmitted from the originating site to all other sites. It is used as the transaction's priority in resolving conflicts.
- *commit_ts*(T): is the *commit_timestamp*, i.e., the time when T finishes writing the objects in its write-set. This is a local timestamp for each site.

When T is submitted in site i, the read-set and the write-set are computed using local copies of objects at site i. After the read-phase (and if no conflict is detected by site

i) a message is sent to every site in the network. The message for transaction T consists of: the id of the originating site, the validation-timestamp *validate_ts*(T), and the write-set $WS(T)$ including the new values for updated objects. The originating site acts as a central controller for the transaction. In this case, site i (the originating site) waits for a reply from all other sites. If all replies are "Yes" then the message "commit T" is sent to every site; otherwise the message "abort T" is sent. Each site maintains the following (local) sets.

- *committed:* the set of transactions that have been committed (in the given site). Transactions are removed from this set when they are no longer needed for validation.
- *local_que:* the set of transactions in the validation-phase which originated (locally) in that site.
- *foreign_que:* the set of transactions in the validation-phase which originated in a foreign site.

In the protocol described below, we only deal with concurrency control issues, i.e., recovery protocols are not discussed (see [3], [6], [12], [20] for discussion on concurrency control and recovery in distributed databases).

When T is submitted at site i, the following protocol is executed at site i (we call this the local **validation** protocol).

```
<< start_ts(T) := clock; RS(T) := WS(T) := φ >>
    read-phase
<< validate_ts(T) := clock; valid := True;
  for each t ∈ local_que ∪ foreign_que do
      if RS(T) ∩ WS(t) ≠ φ then valid := False;
  for each t ∈ committed and
  start_ts(T) < commit_ts(t) do
      if RS(T) ∩ WS(t) ≠ φ then valid := False;
  if valid then local_que := local_que ∪ {T} >>
  if valid then send message T to all sites
  else rollback T
  wait for all replies of T to arrive
  if all replies = Yes then send commit T to all sites
      else send abort T
```

The following is the protocol to commit T at any site (the **commit** protocol).

```
for each x ∈ WS(T) do
    << if timestamp(x) < validate_ts(T)
      then [write x;
      timestamp(x) := validate_ts(T)] >>
<< commit_ts(T) := clock;
committed := committed ∪ {T}
  remove T from either local_que or foreign_que >>
```

The following protocol is executed at site j when it receives a message for transaction T from site i. We will call this protocol the **foreign validation** protocol.

```
valid := True;
<< for all t ∈ local_que and
validate_ts(t) < validate_ts(T) do
    if RS(T) ∩ WS(t) ≠ φ then valid := False; >>
if valid = False then [send message {T, No}
to site i; exit]
send message {T, Yes} to site i
<< foreign_que := foreign_que ∪ {T};
  for each t ∈ local_que and
  validate_ts(t) > validate_ts(T) do
      if RS(t) ∩ WS(T) ≠ φ then abort t >>
```

Notice that when site j receives message T from site i, it only validates T against transactions originating at j. This is because if T has a conflict with a transaction originating at a different site, say site k, the conflict will be detected by either site i or k, and the appropriate transaction will be aborted. A variation, however, is to let site j validate T against both *local_que* and *foreign_que* (these two sets can then be combined). Both versions work correctly. Proof of correctness is based on showing that serializability [1], [14] is enforced through the use of the different timestamps described before. The proof is not difficult and is omitted.

As in the case of the single-site model, timestamps do not have to be attached permanently to the database. In each site, transactions are retained in the set *committed* as long as there is a transaction in *local_que* which has a begin-timestamp that is smaller than the commit-timestamp of the committed transaction. The comment on further optimization given in Section IV-B for the single-site case is also applicable to the distributed model.

VII. Case of Unsynchronized Clocks

The begin-timestamp *start_ts*(T) and the commit-timestamp *commit_ts*(T) are local timestamps generated at the site where transaction T originates, and they do not require the clocks to be synchronized. However, the validation timestamp *validate_ts*(T) is transmitted to all sites and is used as the transaction's priority in resolving conflicts and in writing objects according to the Thomas update rule. The validation timestamp therefore requires that clocks at all sites are synchronized. Clock synchronization is a well-known problem [16] and the overhead of its implementation is normally high. We have examined different ways to implement our optimistic protocols on networks that lack an underlying low-level mechanism for clock synchronization. We concluded that an efficient and simple method is to use a central site to issue validation timestamps. The modification of the protocols to handle this case is described next.

The purpose of the validation timestamp *validate_ts*(T) is twofold: 1) to ensure that the same serial execution of transactions is adopted at all sites, and 2) to write objects correctly according to the Thomas update rule. To see why strict clock synchronization is required, consider a short transaction T_1 which originated at site i and whose validation timestamp was *validate_ts*(T_1) = v_1. Shortly after T_1 committed at site $j \neq i$, transaction T_2 was submitted at site j, read objects written by T_1 and was later assigned a validation timestamp *validate_ts*(T_2) = v_2. If the clocks at sites i and j are not synchronized,

it is possible that $v_2 < v_1$ (as if T_2 were first to reach the validation phase). When T_2 is committed, say at site j, it will not overwrite objects updated by T_1, thus violating the serializability requirement.

A single site, called the clock site, is used to assign validation timestamps to all transactions according to its clock. When a transaction finishes its read phase and before it enters the validation phase, the message "timestamp-request" is sent to the clock site. The clock site replies by sending the value of its clock. The transaction then enters the validation phase using the validation timestamp sent by the clock site. Before discussing the details of this scheme, it is important to point out that the scheme is considerably less costly than protocols to synchronize clocks at all sites.

Sending the timestamp-request and waiting for the reply must be done outside the validation critical section (for practical considerations, waiting inside critical sections should be avoided). Thus sending the timestamp-request is done as the last activity of the read phase, i.e., the read phase for a transaction is completed only when the timestamp-reply is received from the clock site. This however has the implication that timestamp-replies from the clock site must arrive to a given site in the order they were transmitted from the clock site. This is because if T_1 and T_2 are two transactions from the same site, and *validate_ts*$(T_1) <$ *validate_ts*(T_2), then T_1 must enter the validation critical section before T_2 (otherwise the validation will not be correct). In general, however, strict ordering for the reception of transmitted messages might not be guaranteed in all distributed networks. Notice that our previous optimistic protocols do not require a specific order in the way messages arrive. We would like therefore to find a solution that is robust with respect to the order of message delivery.

One solution is to keep a local variable, *last_vts*, in each site to store the value of the validation timestamp for the last transaction (local or foreign) that validated in that site. If the next local transaction entering the critical section, say transaction T, has a validation timestamp smaller (i.e., earlier) than *last_vts*, its validation phase is discontinued and another timestamp-request for T is sent to the clock site. This, in effect, elongates the read phase of T but does not violate serializability. A careful examination, however, reveals that a better solution is possible. If the local transaction T has a timestamp smaller than *last_vts*, the timestamp of T is modified by setting its value to *last_vts* $+ \delta$, where δ is a small quantity chosen such that the value of *last_vts* $+ \delta$ is no larger than the value of the timestamp that would be obtained by resending a timestamp-request to the clock site, i.e., *last_vts* $+ \delta$ is no larger than the current value of the clock at the clock site (notice that this reassignment does not violate the uniqueness of timestamps since the id of the originating site is appended as part of the timestamp). Thus the protocol executed at site i when transaction T is submitted at site i is as follows (the local validation protocol).

$<<$ *start_ts*$(T) :=$ *clock*; *RS*$(T) :=$ *WS*$(T) := \phi >>$

read-phase

Send a timestamp-request to the clock site
Wait for the reply from the clock site
Assign value of reply to *validate_ts*(T)

$<<$ if *validate_ts*$(T) \leq$ *last_vts*
then validate_ts$(T) :=$ *last_vts* $+ \delta$;
last_vts $:=$ *validate_ts*(T); *valid* $:=$ *True*;
for *each* $t \in$ *local_que* $\cup$ *foreign_que do*
if *RS*$(T) \cap$ *WS*$(t) \neq \phi$ *then valid* $:=$ *False*;
for *each* $t \in$ *committed*
and *start_ts*$(T) <$ *commit_ts*(t) *do*
if *RS*$(T) \cap$ *WS*$(t) \neq \phi$ *then valid* $:=$ *False*;
if *valid then local_que* $:=$ *local_que* $\cup \{T\} >>$
if *valid then send message T*
to all sites else rollback T
wait for *all replies of T to arrive*
if *all replies* $=$ *Yes then send commit T to all sites*
else send abort T

The foreign validation protocol is modified so that if a validated foreign transaction has a validation timestamp greater than the value of *last_vts* of that site, then *last_vts* is set to the value of the validation timestamp of that foreign transaction (the validation timestamp of a transaction cannot be changed by a foreign site). Notice that this modification is necessary because local transactions entering the local validation critical section must have validation timestamps greater than those of already validated local and foreign transactions. Notice also that it is not possible for a transaction T to have a validation timestamp smaller than that of a transaction which committed or started validation while T was in its read phase. The commit protocol remains unchanged. We conclude this section by the following remarks.

1) It is possible to use more than one clock site provided that their clocks are synchronized. For example, two sites can be selected as clock sites and timestamp-requests can be sent to any of the two sites. The clocks of these two sites must be synchronized (this should be less costly than synchronizing clocks at all sites).

2) Another modification is to perform a preliminary validation check (outside the critical section) so that a timestamp-request is sent only if the transaction passes this preliminary check. The purpose of this preliminary validation is to avoid sending timestamp-requests for transactions that have conflicts and that will be backed up. The preliminary validation is not done inside a critical section and is similar to the scheme of moving some of the validation phase outside the critical section [15] in order to increase concurrency at the validation phase level.

VIII. Partially Duplicated Distributed Databases

It is easy to see that the scheme presented for the fully duplicated network is also applicable (with minor modification) to the case of partially duplicated distributed sys-

tem in which all objects accessed by a transaction reside in the site where the transaction originated (in fact only the objects in the read set need to reside in the originating site). When this transaction is committed at any site, writing of nonresident objects is simply ignored.

Storing a given file in several sites is usually done if the file is of global interest to the community of users in these sites. One advantage for duplication is to reduce the response time for transactions accessing these files. In practice, it is expected that most transactions will access objects for which there is a copy in the originating site. Requests to access nonresident objects are usually unfrequent. We therefore advocate that such requests are handled by first obtaining a (consistent) copy of the missing object and temporarily storing it in the site where the transaction originated. This copy is kept (at least) until the transaction is committed. The copy is subject to updates by any transaction that commits in that site. It is easy to see that this approach reduces the model to that of the above restricted case of partial replication (i.e., all needed objects reside in the originating site). We feel that this approach is justified for the following reasons: 1) its simple logic, 2) access to nonresident objects is expected to be unfrequent, and 3) cost of transmitting objects from one site to another can usually be considerably reduced by data compression methods.

The retrieved copy of the missing object must be a consistent one. To see this, suppose that site i issued a request to site j for a copy of object x. If, at an earlier time, a third site, say site k, issued commit message that updated x, then this message would be ignored by site i since x is nonresident. Now if site j sent an unupdated copy of object x (because, for example, the commit message from site k has not been received by site j yet), then site i in this case would receive an inconsistent copy of x.

The following protocol is executed by site i in order to obtain a consistent copy of object x. Several variations and refinements of this protocol are possible, but are not discussed in this paper.

1) Site i sends a request for object x to site j (j is any site that has a copy of x). Once site i issues that request, it will no longer ignore updates to x issued by transactions being committed at site i. These updates are stored until the protocol to retrieve x is finished.

2) Site j has a copy of x which is consistent (updated) with respect to its local transactions. To handle foreign transactions, site j examines its *foreign_que* set for transactions that update x. It then makes a list of the id's of these transactions, and sends the appropriate sublist to each of the originating sites, i.e., each originating site gets a list of its local transactions that could have updated x.

3) Let us assume that site k is one of these originating sites and that it has received the list L_k from site j. Site k then sends to site j a list of transactions which is the intersection of L_k and its *local_que* set. For this version of the protocol, we assume that messages are received in the order they are transmitted, and therefore site j must have received commands to abort or commit transactions that are in the list sent to k (i.e., L_k) but not in the list received back from k. Thus site j has now a copy of x that is consistent with respect to those transactions in the set difference of the sent lists and the received lists.

4) Site j sends to site i a copy of object x along with the list of transactions received back from other sites. Since site i has saved information about updates to x (saving of updates to x started when the protocol to retrieve x started), it can examine each saved update, and either performs it or aborts it depending on its timestamp. Site i has now a consistent copy of x.

IX. Conclusion

Optimistic concurrency control protocols have the advantage of removing the overhead of lock maintenance and deadlock handling. They have the potential of becoming practically viable methods for handling concurrency in environments with low conflict rates. In this paper we have presented techniques to improve the concurrency and enlarge the scope of application of optimistic concurrency control methods. Both the single-site and distributed models have been considered. The techniques presented in this paper are examples of one approach through which optimistic methods can gain practical success. Another approach is to integrate various concurrency control methods (including optimistic methods) into the same database system with the ability to apply each method to the appropriate class of transactions and ensure that conflicts among the different classes are resolved correctly. Both approaches seem worthy of further investigation and new advancement using either approach would greatly enhance the applicability of optimistic concurrency control methods.

References

[1] D. Badal, "Correctness of concurrency control and implications in distributed systems," in *Proc. COMPSAC Conf.*, 1979, pp. 588–594.

[2] M. Bassiouni and U. Khamare, "Optimistic concurrency control-schemes for performance enhancement," in *Proc. IEEE 10th Int. COMPSAC Conf.*, October 1986, pp. 43–49.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Reading, MA: Addison-Wesley, 1987.

[4] P. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surveys*, vol. 13, no. 2, pp. 185–122, June 1981.

[5] P. Bernstein, D. Shipman, and B. Rothnie, "Concurrency control in a system for distributed database," *ACM Trans. Database Syst.*, vol. 5, no. 1, pp. 18–51, Mar. 1980.

[6] B. Bhargava, "Resiliency features of the optimistic concurrency control approach for distributed database systems," in *Proc. 2nd IEEE Conf. Reliability on Distributed Software and DBS*, 1982, pp. 19–32.

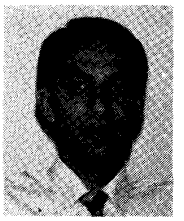
[7] C. Boksenbaum, M. Cart, J. Ferrie, and J. Pons," "Concurrent certifications by intervals of timestamps in distributed database systems," *IEEE Trans. Software Eng.*, vol. SE-13, no. 4, pp. 409–419, 1987.

[8] H. Boral and I. Gold, "Towards a self adapting centralized concurrency control algorithm," in *ACM SIGMOD Conf.*, Boston, MA, June 1984, pp. 18–31.

[9] M. Carey, "Modeling and evalution of concurrency control algorithms," Ph.D. dissertation, Univ. California at Berkeley, 1983.

[10] M. Carey and M. Stonebraker, "The performance of concurrency

control algorithms for database management systems," in *Proc. VLDB*, 1984, pp. 107–118.
[11] W. Cheng and G. Belford, "Update synchronization in distributed databases," in *Proc. VLDB*, 1980, pp. 301–308.
[12] S. Davidson, "Optimization and consistency in partitioned distributed database systems," *ACM Trans. Database Syst.*, vol. 9, no. 3, pp. 456–481, 1984.
[13] D. DeWitt, R. Katz, F. Olken, and L. Shapiro, "Implementation techniques for main memory database systems," in *ACM SIGMOD Conf.*, Boston, MA, June 1984, pp. 1–18.
[14] K. Eswaran, J. Gray, R. Lorie, and I. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, no. 11, pp. 624–633, Nov. 1976.
[15] H. Kung and J. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, June 1981.
[16] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.
[17] G. Lausen, "Concurrency control in database systems: a step towards the integration of optimistic methods and locking," in *Proc. ACM Computer Conf.*, 1982, pp. 64–68.
[18] W. Lin and J. Nolte, "Basic timestamp, multiple version timestamp, and two-phase locking," in *Proc. VLDB Conf.*, Florence, Italy, Oct. 1983, pp. 109–119.
[19] M. Maekawa, A. Oldehoeft, and R. Oldehoeft, *Operating Systems—Advanced Concepts*. Menlo Park, CA: Benjamin/Cummings, 1987.
[20] D. Menasce, G. Popek, and R. Muntz, "A locking protocol for resource coordination in distributed databases," *ACM Trans. Database Syst.*, vol. 5, no. 2, pp. 103–138, 1980.
[21] P. Peinl and A. Reuter, "Empirical comparison of database concurrency control schemes," in *Proc. VLDB Conf.*, Florence, Italy, Oct. 1983, pp. 97–108.
[22] M. Reimer, "Solving the phantom problem by predicative optimistic concurrency control," in *Proc. VLDB Conf.*, Florence, Italy, 1983, pp. 81–88.
[23] G. Schlageter, "Optimistic methods for concurrency control in distributed database systems," in *Proc. VLDB Conf.*, Cannes, France, Sept. 1981, pp. 125–130.
[24] A. Silberschatz and Z. Kedem, "A family of locking protocols for database systems that are modeled by directed graphs," *IEEE Trans. Software Eng.*, vol. SE-8, no. 6, Nov. 1982.
[25] R. Thomas, "A majority consensus approach to concurrency control," *ACM Trans. Database Syst.*, vol. 4, no. 2, pp. 180–209, June 1979.



M. A. Bassiouni received the Ph.D. degree in computer science from the Pennsylvania State University, University Park, in 1982.

He is currently an Associate Professor of Computer Science at the University of Central Florida, Orlando. His current research interests include distributed systems, databases, and performance evaluation. He authored several papers and has been actively involved in research on data encoding, I/O measurements and modeling, schemes of file allocation, local area networks, and user interfaces to relational database systems.

Dr. Bassiouni is a member of the IEEE Computer Society, the Association for Computing Machinery, and the American Society for Information Science.