SEMANTICS BASED TRANSACTION MANAGEMENT TECHNIQUES FOR REPLICATED DATA

Akhil Kumar[1] *and Michael Stonebraker*[2]

University of California
Berkeley, Ca., 94720

Abstract

Data is often replicated in distributed database applications to improve availability and response time. Conventional multi-copy algorithms deliver fast response times and high availability for read-only transactions while sacrificing these goals for updates. In this paper, we propose a multi-copy algorithm that works well in both retrieval and update environments by exploiting special application semantics. By subdividing transactions into various categories, and utilizing a commutativity property, we demonstrate cheaper techniques and show that they guarantee correctness. A performance comparison between our techniques and conventional ones quantifies the extent of the savings.

1. Introduction

There have been several proposals for reducing transaction management overhead in a database system by exploiting application specific semantics. Some of these ideas entail locking schemes for specialized environments that would permit greater concurrency (e.g. [ELLI83, KORT83, MOHA85]). Other proposals exploit the commutativity property whereby certain arithmetic operations can be performed in any order [GRAY81, BEER83, GARC83, WEIN84, SARI85, BADR87]. The escrow method [ONEI86, REUT82] also exploits commutativity and allows enforcement of integrity constraints, such as ensuring that a data item is non-negative.

This paper focuses on exploiting commutativity for transaction management when replicated data is present. In our environment not only do we wish to lower transaction overhead but also desire to improve availability.

Several inter-site messages per transaction are usually required for conventional multi-copy algorithms. For example, in the majority consensus method [THOM79], a transaction must read from and update the copies at a majority of the sites before it can commit. The quorum consensus method [GIFF79] is a generalization of the majority consensus method with similar properties. In the read-one write-all method [BERN84], query transactions can read from any single site while update transactions must write to all sites. This leads to better performance for queries at the expense of poorer performance for updates. Additional methods are discussed in [STON79, EAGE81, ELAB85]. Obviously, response time is inversely related to the number of sites a transaction must communicate with before commiting, and deteriorates as the inter-site traffic volume increases.

This paper shows that commutativity can be exploited to alleviate this problem. Specifically, we examine how commutativity can be extended to a replicated data environment by introducing a transaction classification scheme, and formalize the conditions under which certain types of transactions can execute independently in parallel at different sites. In addition we develop protocols for executing transactions which conflict and then analytically compute the message savings from our composite algorithms.

We naturally take advantage of the fact that replication provides multiple versions of every data object. In our algorithms these versions do not necessarily have the same value. However, read-only transactions always see a consistent, albeit perhaps out-of-date version, if they read objects from a single site or a restricted set of sites. Furthermore, we include techniques for bounding out-of-datedness when appropriate and also for enforcing integrity constraints.

The paper is organized as follows. In section 2, a brief theoretical framework for commutativity in a replicated data environment is presented. This forms the basis for our multi-copy algorithm which is presented in section 3 along with a performance comparison against the majority consensus algorithm. Section 4 turns to techniques for enforcing additional integrity constraints and imposing limits on out-of-datedness. Finally, in section 5 we add a new category to the transaction classification scheme, and highlight the advantages which result.

---

[1] Graduate School of Business, 350 Barrows Hall

[2] EECS Department, 549 Evans Hall

This research was sponsored by a grant from the IBM Corporation and an Arthur Andersen & Co. Foundation Doctoral Dissertation Fellowship.

2. Multi-Copy Commutativity

A database, D consists of a collection of objects, o_i. We assume there are n sites and each of m data objects is replicated at all n sites. This complete replication assumption simplifies the presentation and can be easily relaxed by introducing the concept of **site groups**, which are collections of sites with at least one copy of each object in that group. Further, it is assumed that a scheduler [BERN81] at each site serializes local transactions using two-phase locking or any standard concurrency control mechanism.

The **object type** of an object defines the set of values from which instances of that object are drawn, e.g., numeric, character string, etc. We view a transaction as a function, $F_R(X)$ which transforms an object X from an initial value X_{old} to a new value X_{new} as follows:

$$X_{new} = F_R(X_{old})$$

where R is a read vector (**r-vector**) of constants or other database objects $(r_1, r_2, \cdots, r_n)$. In the special case where R is a vector of all constants, c_i the transaction is represented as $F_C(X)$ and C is called a constant vector (**c-vector**). We shall restrict our treatment here to transactions with one function of the above type. It is a straight forward generalization to consider functions that update multiple objects and transactions that consist of multiple functions.

Example 1: Some examples of functions on numeric data objects are:

$F1_C(X) = c_1X$
$F2_C(X) = X + c_1$
$F3_C(X) = X + c_1X$

Two transactions, F and H, **commute** if†:

1. $H_{R2} * F_{R1}(X) = F_{R1} * H_{R2}(X)$ for *all* X, $R1$, and $R2$
2. $F_{R1}(X)|Y = F_{R1}(X)|H_{R2}(Y)$ for *all* $X, Y, R1$, and $R2$
3. $H_{R2}(Y)|X = H_{R2}(Y)|F_{R1}(X)$ for *all* $X, Y, R1$, and $R2$

where R1 and R2 are r-vectors associated with F and H respectively.

In 1 above, the left hand side denotes the state that is obtained if F and H are performed successively on an object X, while in the expression on the right hand side the functions are applied in the reverse order; in either case, the final state of X is the same. Conditions 2 and 3 state that $F_{R1}(X)$ and $H_{R2}(Y)$ may be executed in either order and yet produce the same final state of objects X and Y. A transaction, F commutes with itself (or **self-commutes**) if:

1. $F_{R1} * F_{R2}(X) = F_{R2} * F_{R1}(X)$ *for all* X, $R1$, and $R2$
2. $F_{R1}(X)|Y = F_{R1}(X)|F_{R2}(Y)$ for *all* $X, Y, R1$, and $R2$

where R1, R2 are instances of r-vectors associated with F.

Example 2: Consider the following two functions:

$$F1_R(X) = X + r_1$$
$$F2_R(Y) = Y - r_2$$

where r_1 and r_2 are other objects in the database.

F1 and F2 do not commute in general; for instance, if $r_1 = Y$ and $r_2 = X$ then the final result will depend upon the order in which the two transactions run. On the other hand, if r_1 and r_2 are replaced by constants, c_1 and c_2 respectively, then F1 and F2 will always commute. This is an example situation in which $F1_R(X)$ and $F2_R(Y)$ do not commute but $F1_C(X)$ and $F2_C(Y)$ do. Since few transactions satisfy general commutativity, we shall consider only $F_C(X)$ type transactions as candidates for the commutativity and self-commutativity properties. These are transactions in which the r-vector consists only of constants. Such transactions occur frequently in real applications, e.g., withdrawing 40 dollars from a bank account.

We now turn to extending this concept to multi-copy commutativity and show how greater concurrency may be realized in the context of replicated data.

Definition 1: H is **mc-compatible** with respect to F, if there exists a vector S, such that:

$$F_S(X) = H_R(X) \quad \textit{for all } X, R$$

It is assumed that an inverse function, IH_F is associated with every transaction H, and allows each transformation produced by function H to be reexpressed in terms of another function, F. Therefore, transaction $H_R(X)$ may be represented in terms of F as $F_S(X)$, where S is a substitution vector (**s-vector**) of H with respect to F. Moreover, any transaction F is made **self mc-compatible** by choosing the vector S equal to R, i.e., the s-vector of $F_R(X)$ with respect to itself is R. Determining the exact conditions under which the above transformation exists is left as a future exercise. However, we shall restrict our attention here to only those transactions which satisfy the above definition.

Example 3: Consider transactions F and H such that:

$$F_C(X) = X + c_1$$
$$H_C(X) = c_1X$$

The function IH_F in this case is written as:

$$IH_F(X, c_1) = c_1X - X$$

The s-vector of H with respect to F consists of the single element, $c_1X - X$.

Theorem 1: If F and H are two transactions on an object type, such that $F_C(X)$ self-commutes, and $H_R(X)$ is any general transaction mc-compatible with F, then

$$F_C * H_R(X) = F_S * F_C(X) \qquad (1)$$

where S is an s-vector of H with respect to F.

Sketch of Proof: Since $F_C(X)$ self-commutes, the right hand side of equation (1) may be rewritten as:

$$F_S * F_C(X) = F_C * F_S(X)$$

Moreover, from Definition 1, the left hand side of equation (1) may be rewritten as:

$$F_C * H_R(X) = F_C * F_S(X)$$

Clearly both the left and right hand sides give the same result and hence the proof follows. □

Corollary 1: If a self-commutative transaction F, and another transaction, H are performed on separate versions of the same object (perhaps at different sites), the s-vector with respect to F corresponding to each transaction is sent to the other site and transaction $F_S(X)$ is performed, then the same final state is reached at both sites. We shall term this property as **multi-copy commutativity** (mc-commutativity), and transactions F and H will be said to **mc-commute.**

Sketch of Proof: This corollary follows directly from Theorem 1. The left hand side of equation (1) denotes the sequence in

† 1. The * sign represents a functional composition.

2. The notation A|B means "A given B".

which the two transactions are performed at the first site, i.e., H followed by F. On the other hand, the right hand side of this equation denotes the sequence in which the transactions are performed at the second site, i.e., F is executed twice, first with a c-vector C and then with an s-vector denoted by S. Since the resulting state is the same in both cases, this corollary is true. In fact, the final result is the same as would be produced if H ran first and F later at both sites. □

The implications of Theorem 1 and Corollary 1 are as follows. Consider n self-commutative transactions $F1_{C1}, F2_{C2}, \cdots, Fn_{Cn}$ defined on object X such that Ci is the c-vector associated with Fi and all pairs Fi, Fj of these transactions commute, while H_R is any general transaction. Then $F1_{C1}, F2_{C2}, \cdots, Fn_{Cn}, Fk_S$ may be performed on X in any sequence and yet will produce the same final state, where $1 \leq k \leq n$, and S is the s-vector of H with respect to Fk.

Now, we propose a classification scheme based on a preanalysis of the various transactions in an application and shall subsequently illustrate how this scheme can be used to devise algorithms for replicated data transaction management. The preanalysis consists of first identifying all transactions which self-commute and grouping them such that all pairs of transactions in a group also commute with each other. If there are m such transaction groups, $G1$, $G2$, ... ,Gm, then any one of these groups is chosen and all transactions in that group are called **C type transactions.** This choice could be arbitrary or it may be based on a criterion like transaction volume. Hence, the transaction group with the highest volume might be selected. All other transactions in the application are categorized as **NC type.** The matrix of Table 1 represents the mc-commutativity of C and NC type transactions with transactions of the same or different type.

Example 4: Consider a simplified banking application. The main transactions are:

Deposit: Add c_1 to account Y
Withdrawal: Subtract c_1 from account Z
Transfer: Move c_1 from account Y to Z
Inquiry: Return the balance of account X
Add Interest: Compute 5% of the amount in account X and add it to X

For now, we will assume that account balances can go below zero. In this example, the deposit and withdrawal transactions can be represented as functions, F1 and F2 respectively as follows:

$$F1_C(X) = X + c_1$$
$$F2_C(X) = X - c_1$$

	C	NC
C	yes	yes
NC	yes	no

Table 1: mc-commutativity between C and NC transactions

Both F1 and F2 self-commute. Moreover, the "transfer" transaction is a composite transaction consisting of two sub-transactions, $F1_C(Z)$ and $F2_C(Y)$. The inquiry transaction is again interpreted as an $F1_C(X)$ transaction where c_1 is 0. Since F1 and F2 self-commute and also commute with each other, the deposit, withdrawl, transfer and inquiry transactions are classified as C type. Finally, the "add interest" transaction is represented as:

$$H_C(X) = X + c_1 X$$

This transaction does not self-commute, and therefore, it is an NC transaction.

To summarize our approach so far, we have defined the mc-commutativity property, shown that C and NC transactions mc-commute, and illustrated how transactions can be categorized into these two categories. It should be clearly noted that a distinction must be drawn between serializability and correctness. Our approach leads to a correct solution, not necessarily a serializable one. For instance, if two transactions are performed separately at two different sites, and the corresponding s-vectors are spooled across the network, the two sites will see their effect in opposite order. However, since correctness is assured and the processing overhead is reduced, we believe this approach is worth pursuing. In [SARI85, BLAU85], the objectives of the authors are similar to ours, but their approach is based on retroactively establishing a system-wide serial order of transactions and applying log transformations to ensure that a correct final state is reached at all sites eventually. In the next section we turn to develop an algorithm that can take advantage of the properties defined here.

3. Algorithm Description and Analysis

In this section, we first describe a multi-copy algorithm for transaction management in a replicated data environment with C and NC transactions and then analyze its performance.

3.1. Algorithm 1

Each site maintains a state vector (nc_i, c_i) where:
c_i: number of C type transactions completed at site i
nc_i: number of NC type transactions completed at site i
c_i and nc_i are counters which are advanced each time a new transaction is performed at a site. C and NC transactions observe different protocols while running.

A C transaction:

(1) Performs updates to local copies and commits upon completion. (A scheduler at each site guarantees serializability among local transactions).

(2) After commit, the corresponding c-vector, the transaction name, and the object name are spooled to all remote sites.

An NC transaction:

(1) forms a quorum of sites by locking a majority of copies of accessed objects.

(2) selects the object at the site with the highest value of nc_i for updating.

(3) performs updates to copies at the chosen site

(4) computes the s-vector with respect to one C type transaction, and executes it at the other sites in the quorum

(5) releases locks and spools the s-vector and the object name to all sites not in the quorum.

Upon receiving a spool message from another site, the s-vector (or c-vector) contained in the message is executed as an atomic transaction. Spool messages sent out from a site are sequenced and a receiving site must execute such messages in the correct order. The spooler program runs at each site and performs the following functions:

(1) It accepts an update message from a transaction and ensures that it is transmitted reliably to all other sites.

(2) It receives messages from other sites and runs them as transactions at the local site.

(3) updates the state vector.

In this algorithm, a C type transaction is allowed to run in isolation at a site while two conflicting NC transactions are made serializable by the quorum consensus algorithm. Note that any other standard method could also be used for this purpose.

If a remote site is down then the spooler program buffers the message until the site comes up. The spooler can either send each transaction as it commits or batch several transactions and spool them together to save messages.

3.2. Performance Analysis

Algorithm 1 has two advantages relative to an algorithm that treats all transactions as NC. First, C type transactions can run locally and need never wait even when partitions and multiple-site failures occur. Secondly, Algorithm 1 requires a reduced number of messages per transaction.

In order to make a quantitative comparison, the following parameters are defined:

k: number of objects accessed by a transaction
n: number of sites
c: conflict ratio [†]
f_c: fraction of C type transactions
n_{maj}: majority among n sites, i.e.,
n_{maj}=n/2 + 1, for even n
n_{maj}=(n+1)/2, for odd n

These parameters are used to compute the average number of messages per transaction in each case. If the quorum consensus algorithm is observed, a transaction must acquire locks on a majority of copies of each object that it accesses (for simplicity, we assume a majority is required for a quorum). Thus, the average number of messages per transaction, denoted by $M_{majority}$ is:

$$M_{majority}=(n_{maj}-1)*k*2*(1+c)+3*(n_{maj}-1)+(n-n_{maj})$$

The first term in the above expression is the cost of accessing n_{maj} sites. It is inflated by the $(1+c)$ factor to take conflicts into account. (No piggy-backing of messages is assumed here). The second term is the cost of performing a two-phase commit [GRAY78] on completion of the transaction. Finally, the third term represents the cost of spooling the updates to the sites not in the majority quorum. The corresponding cost, in terms of messages per transaction ($M_{Alg.1}$), for Algorithm 1 is:

$$M_{Alg.1}=(n-1)\times f_c+(1-f_c)\times M_{majority}$$

The expression for $M_{Alg.1}$ consists of two parts. The first part pertains to C type transactions and is the cost of spooling the s-vector to all the other sites after a transaction commits. The second part pertains to NC transactions and is the cost of communicating with $(n_{maj}-1)$ sites. In Figure 1, $M_{majority}$ and $M_{Alg.1}$ are plotted against f_c for representative values of the other parameters. Here n is assumed as 5, k as 5, and c as 0.02. The gap between the two algorithms increases linearly with decreasing f_c, and is 43% when f_c is 0.5, and 86% when f_c is 1.0.

In the above comparison it was assumed that the s-vector of one transaction is sent to other sites as a separate message. Still larger savings would result if the updates of several transactions are batched and sent together as one spool message. There is a clear trade-off here between out-of-datedness and message cost.

Since updates made by a C type transaction at a site are spooled to other sites, the sites will be temporarily inconsistent with each other. These inconsistencies between the local states will, however, get resolved once the updates of all sites are spooled to and run at all other sites. This out-of-datedness can be bounded in one of two ways. The first method is to ensure that the spool programs at all sites are assigned a high enough priority so that all messages are spooled out within a given maximum amount of time. The other way this can be done is by periodically running a marking procedure to establish a global state or snapshot [CHAN85].

4. Integrity Issues

4.1. Introduction

In example 4, we treated the withdrawal transaction as a C type transaction by assuming that negative bank balances were allowed. Now we consider an environment which requires the constraint that all account balances are positive. It is obvious that a withdrawal transaction can no longer be treated as C type

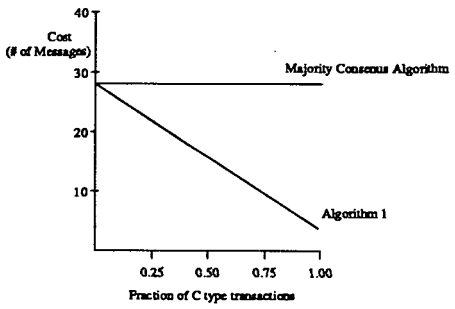

Figure 1: Plot of cost vs. f_c

[†] The conflict ratio is defined as the probability that a request for an object will be denied because it is already locked by another transaction; hence a second attempt must be made.

because two simultaneous transactions at different sites could independently withdraw the entire balance from the same account. Treating such transactions as NC within the framework of Algorithm 1 would rectify the problem but impose a much higher processing overhead.

We propose to classify transactions which must not violate integrity constraints but are otherwise C type (such as the withdrawal transaction) as **PC type** and extend our analysis to this third class. The execution of a PC transaction can be decomposed as shown in Figure 2. First, one must determine whether this transaction might violate any integrity constraint. If so, the transaction must be rejected. Otherwise, it can run as a C type transaction.

Algorithm 2, described below, enforces such integrity constraints. This algorithm is a variant of the escrow method [ONEI86] and by examining just one site, it can often guarantee that integrity will not be violated. This will allow a PC transaction to run just like a C transaction. In section 4.3, we analyze the performance of this technique and determine the conditions under which it is a superior alternative to treating PC transactions as NC.

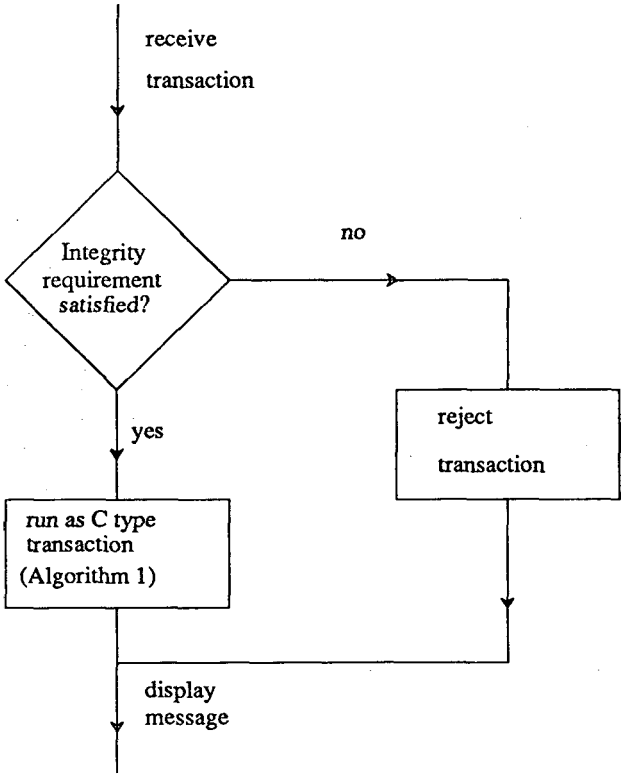

Figure 2: Procedure for executing a PC type transaction

4.2. Algorithm 2

This algorithm is a general mechanism for implementing the following classes of constraints:

1) Value of object B > B_{min}, and
2) Value of object B < B_{max}

where B is a numeric database object to which the integrity constraints apply.

These are example lower and upper bound constraints respectively. For ease of exposition we will illustrate the algorithm for lower bound constraints only; upper bound constraints may similarly be enforced. The main steps in this algorithm are:

1. At periodic intervals a global state is created by executing the marking procedure (described below) so that all sites have a consistent view of the database at that time.

2. Each site is assigned an escrow quantity, es_i for every numeric object to which an integrity constraint applies, and its initial value is computed as:

$$es_i = \frac{confirmed\ balance - B_{min}}{number\ of\ copies}$$

where *confirmed balance* is the balance as of the most recent global state.

3. Changes made to object B by PC or C type transactions will have a corresponding effect on the escrow quantity and these are tracked at every site. For instance, before a decrement transaction is executed, a check is made to ensure that es_i will remain greater than 0 even after the transaction. Thus, if the value of object B at site i is to be decremented by X, then the following check is made first:

$$es_i \geq X$$

If this test fails, then site i must communicate with some number, k of additional sites and borrow from their escrow such that:

$$es_{local} + \sum_{i=1}^{k} es_{rem_i} \geq X$$

where

es_{local}: escrow amount at the local site
es_{rem_i}: escrow amount at the i^{th} remote site

If the left hand side remains smaller than X even after including all the sites, then the transaction must be disallowed or else the final balance would drop below B_{min}.

4. An NC transaction at site i updates the escrow quantity,es_k at all sites k: $k \in S$, where S is a set of sites constituting a majority of the copies of the data object being updated. If the magnitude of the change produced on an object by this transaction is X, then the escrow quantity at each such site is adjusted by an amount X/n_S, where n_S is the number of sites in set S. This is only done for fairness although X can also be allocated arbitrarily subject to availability. Again, if it is not possible to accumulate an amount equal to X, in spite of adding the escrow pools at all sites, then the transaction is rejected.

5. The timings of performing the increment and decrement operations on the escrow quantity are different. An increment to escrow must be delayed until transaction commit

time, while a decrement to escrow must be made before the operation itself. The reason for the first rule is that any additions to escrow should be made available to other transactions only after the transaction which produces this effect commits itself. On the other hand, a decrement succeeds only if it can obtain the necessary quantity from escrow and hence, this step must be done before the operation itself. A transaction that aborts must undo its partial effect on escrows just as it would on any other objects.

Marking Procedure

The marking Procedure is executed at periodic intervals to establish a global state; thus, ensuring that all sites have the same value for various objects in the database.† This algorithm is implemented by sending two rounds of messages between all sites in a predetermined sequence. Each site maintains a **spool queue,** i.e., a queue of s-vectors not yet spooled to other sites. The algorithm is initiated at any site and in the first cycle the initiating site sends its spool queue to the next site in the sequence. Upon receiving a spool queue, each subsequent site:

(1) executes the collection of s-vectors received,

(2) adds its own spool queue to it, and

(3) sends the queue to the next site in the sequence.

The second cycle starts when the spool queue returns to the initiating site after visiting all sites once. At this point, the initiating site executes the global spool queue again, and transmits it to the next site in the sequence without making any additions to it. Each site upon receiving the spool queue a second time executes the s-vectors it did not receive the previous time and the resulting state is the new global state for the site.

All transactions initiated at any site between the first and second visits of the spool queue are either suspended or stored as a collection of s-vectors. These stored s-vectors at a site are executed locally and spooled to other sites only after the new global state is reached. Moreover, between the two visits of the spool queue, a site does not accept spooled s-vectors from other sites. This procedure is equivalent to the global-state-detection algorithm in [CHAN85].

4.3. Performance

It was shown above that PC transactions may be treated in two ways:

Alternative 1: as NC transactions in Algorithm 1

Alternative 2: as C type transactions in Algorithm 2

These two alternatives are compared below.

A PC transaction (say, "decrement X from B") will have to accumulate from escrow at one or more sites an amount X; hence, the number of sites it must communicate with depends on X. If the global state value of object B is denoted as B^*, and an integrity requirement *object* $B \geq B_{min}$ exists, then this transaction must exchange messages with at least $\left(\left\lceil n \times X/(B^* - B_{min})\right\rceil - 1\right)$ sites.† Moreover, if X is uniformly distributed in the range (B_{min}, B^*), then the transaction would on an average exchange messages with half the number of sites or n/2. Therefore, in terms of messages, alternative 2 would cost as much as alternative 1 for this distribution of X.

To compare alternative 2 against alternative 1 for other scenarios, we introduce a Performance Improvement factor (PIF) computed as:

$$PIF = \frac{n_{maj} - 1}{\left\lceil n \times Avg(X)/Avg(B^* - B_{min})\right\rceil - 1}$$

where

n: total number of sites

n_{maj}: number of sites required for a majority

Avg(B^*): average value of object B

Avg(X): average amount to be decremented

B_{min}: minimum value of object B permissible

(Note: when $n \times Avg(X)/Avg(B^* - B_{min}) < 1$, PIF is undefined).

Thus, if n is assumed to be 5, n_{maj} is 3. Further, assume Avg($B^* - B_{min}$) is 1000, and Avg(X) is 400. Therefore, PIF is 2 which is equivalent to a 100% performance advantage for alternative 2.

It should also be noted that for ease of analysis we have ignored the additional cost of running Algorithm 2. A more detailed simulation study is necessary in order to take this into account but it is outside the scope of this paper.

5. Further Enhancements

In this section, we turn to two methods for further enhancing the performance of Algorithm 1. First, the mc-commutativity matrix of Table 1 is expanded by splitting NC transactions into two categories. The second idea is to introduce site-dependent quorums for executing NC transactions. The first proposal and an analysis of its performance appear in sections 5.1 and 5.2 respectively, while the second scheme is described in section 5.3.

5.1. Increased Commutativity

With a view to increasing the fraction of transactions which commute, we suggest splitting NC transactions into two groups: **NC1** transactions which mc-commute with other NC1 transactions and C transactions but do not mc-commute with NC2 transactions, and **NC2** transactions which mc-commute with only C type transactions. Figure 3 indicates the complete classification of transaction types, and Table 2 shows the mc-commutativity among them.

To recapitulate the theoretical framework so far, a transaction Fi is C type only if it self-commutes and also commutes with all other transactions, Fj that belong to the group of C type transactions. The attempt to further split NC transactions is motivated by the observation that even if the c-vector, C contains other objects from the database and not just external constants, two such transactions might still mc-commute under certain conditions.

Definition 2: $F_R(X)$ is an **NC1 transaction** if:

1. The corresponding $F_C(X)$ obtained by substituting c_i for each r_i is a C type transaction and,

† As mentioned in section 3.2, this marking procedure also limits the extent to which a site becomes out-of-date with respect to other sites. The frequency at which it is executed is varied according to the needs of an application.

† $\lceil x \rceil$ (or ceiling of x) is the smallest integer number larger than or equal to x.

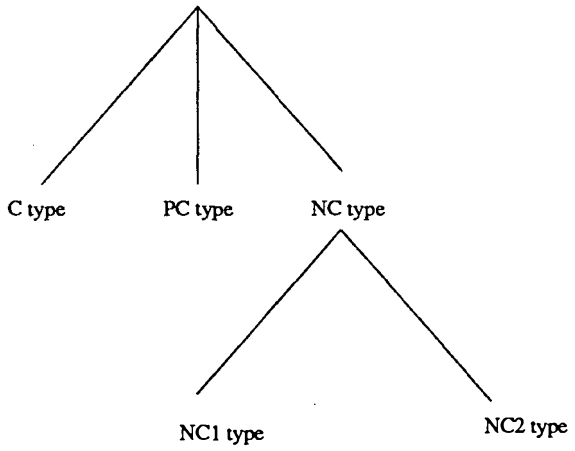

Figure 3: Transaction Classification Scheme

	C	PC	NC1	NC2
C	yes	yes	yes	yes
PC	yes	yes	yes	yes
NC1	yes	yes	yes	no
NC2	yes	yes	no	no

Table 2: mc-commutativity matrix with 4 transaction types

2. The combined RO-X dependency graph of all NC1 transactions does not contain cycles.

A dependency in this graph is indicated by a directed arrow from an object in the RO set (the read-only set) to the object X. Basically, the main difference between C and NC1 transactions is that the r-vector of C transactions consists of constants while that of NC1 transactions may also include other database objects. The following example illustrates an NC1 transaction.

Example 5: Consider the transaction:

$$F_R(X) = X + Y + c$$

where R: the r-vector, (Y,c)

Y: a database object

c: a constant.

For the above $F_R(X)$, the corresponding $F_C(X) = X + c_1 + c_2$, and it self-commutes. Therefore, if $F_C(X)$ is a C type transaction then $F_R(X)$ is a potential NC1 transaction.

In order to identify NC1 transactions in a group of NC transactions, all transactions which satisfy condition 1 in the above definition are isolated. Then a dependency graph is constructed for the isolated transactions. The combined dependency graph is obtained by superimposing the individual graphs for all potential candidates. If any cycles are found in the combined graph, they must be eliminated by removing one or more transactions from it. The remaining transactions in this group may be treated as NC1, and all other transactions within NC are treated as NC2. Readers familiar with concurrency control in SDD-1 [BERN80a, BERN80b] should notice the similarity between the conflict graph analysis technique used there and our method for isolating NC1 transactions. However, there conflict graph analysis is applied in the context of a single-copy system in order to identify those transactions which can be run without any concurrency control mechanism in force.

Because NC1 transactions mc-commute, this new category increases concurrency and reduces transaction processing overhead. The following example illustrates a dependency graph.

Example 6: Consider the following three transactions:

T1: add 10% of object X to object Y
T2: add 10% of object Y to object Z
T3: add 10% of object Z to object X

These transactions may be expressed in functional form as F1, F2 and F3 respectively as follows:

$$F1_R(Y) = Y + 0.1X$$
$$F2_R(Z) = Z + 0.1Y$$
$$F3_R(X) = X + 0.1Z$$

The corresponding $F1_C(Y)$, $F2_C(Z)$, and $F3_C(X)$ functions self-commute and also commute with each other. Therefore, T1, T2 and T3 are potential NC1 transactions. However, their combined dependency graph shown in Figure 4 contains a cycle. In order to break the cycle it is necessary to make one of these three transactions NC2. The choice of a victim could be made arbitrarily or based on the volume of each type of transaction. The transaction which is executed least frequently may be made NC2 because the cost of executing an NC2 transaction is greater than an NC1 transaction.

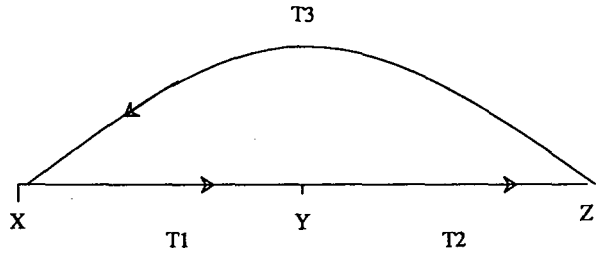

Figure 4: Dependency graph for T1, T2 and T3

5.2. Performance Considerations

Now we turn to examine the performance improvement which results from dividing NC transactions into these two categories. NC transactions, in Algorithm 1, execute by acquiring locks on a quorum equal to a majority copies of an object to be accessed, and therefore, any two quorums always intersect. The main improvement here arises from the fact that since NC1 transactions mc-commute, their quorums need not intersect. However, since NC1 and NC2 transactions do not mc-commute, their quorums must intersect ([GIFF79]), and the following invariant must hold:

$$n_{nc1} + n_{nc2} \geq n$$

where

n_{nc1}: number of copies locked by a NC1 transaction
n_{nc2}: number of copies locked by a NC2 transaction

This means that it is not necessary for an NC1 transaction to lock a majority of sites (i.e., $n_{nc1} < n_{maj}$ is possible), and thus, there is greater flexibility in quorum assignments. Here again, one criterion for determining suitable values of n_{nc1} and n_{nc2} is the volume of transactions of each type. If the volume of NC1 transactions is large in comparison with the volume of NC2 transactions then it is preferable to make n_{nc1} small and n_{nc2} correspondingly large. In summary, we have argued that dividing NC type transactions into two sub-classes can potentially improve performance and determining the correct values for n_{nc1} and n_{nc2} is an optimization issue.

5.3. Site-dependent Quorums

It was stated earlier that two or more concurrent NC type transactions must be made serializable. In Algorithm 1, this was done by implementing the majority consensus protocol which requires that each transaction must acquire locks on a majority of sites. The concept of site-dependent quorums is an enhancement to that scheme. A quorum is site-dependent if the size of the quorum is a function of the site from which the transaction is executed. If the quorums are defined in such a way that any two site-dependent quorums intersect, then it is easy to show that serializability is guaranteed.

This scheme is advantageous if some NC transactions are executed mainly from designated primary sites. In such a case it is possible to stipulate that a NC transaction must assemble a quorum of n_p^t, (where n_p^t is the number of primary sites for transaction t) if it runs at the primary site, but if it runs at another site then a quorum of $n + 1 - n_p^t$ is necessary. For instance, in example 4 the "add interest" transaction is NC and it is likely that this will be executed only from a single site in the system. Therefore, this site would be assigned a quorum of 1 while other sites would be assigned a quorum of n since it is unlikely that they will execute the same transaction.

6. Conclusion

We introduced new transaction types in the context of replicated data and described how transactions should be subdivided into these groups in order to enhance the performance of a multi-copy distributed database application. A new property called mc-commutativity was defined for this environment and compatibility between the various transaction types was determined in terms of this property. Moreover, algorithms for executing multiple types of transactions were discussed. It was also demonstrated that integrity issues can be incorporated into this framework and therefore, situations like overdrawn bank accounts do not occur and committed transactions need not be undone. Finally, savings in the number of messages were analytically computed.

Our techniques ensure correctness, though not serializability, and take advantage of the fact that several versions of each object exist in a multi-copy environment. Since methods for limiting the extent of out-of-datedness between versions are also included, we believe these techniques should be acceptable in most applications. It is evident that with intelligent applications design, dramatic speed-ups in response time and throughput can be realized. A promising direction for future research would be to simulate or actually implement this approach and evaluate its performance in a real application environment. Furthermore, even though the examples that have been presented here pertain to numeric data, we believe it should be possible generalize the use of this scheme to any abstract data types.

References

[BADR87] Badrinath, B.R., and Ramamritham, K., "Semantics-Based concurrency Control Beyond Commutativity", Third IEEE International Conference on Data Engineering, Los Angeles, Feb 1987.

[BEER83] Beeri, C., et. al., "A Concurrency Control Theory for Nested Transactions", Proceedings of Second Symposium on Principles of Distributed Computing, August 1983.

[BERN80a] Bernstein, P.A., and Shipman, D.W., "The Correctness of Concurrency Control Mechanisms in a System for Distributed Databases (SDD-1)", ACM TODS, Vol. 5, No. 1, March 1980.

[BERN80b] Bernstein, P.A., et. al., "Concurrency Control in a System for Distributed Databases (SDD-1)", ACM TODS, Vol. 5, No. 1, March 1980.

[BERN81] Bernstein, P.A., and Goodman, N., "Concurrency Control in Distributed Database Systems", ACM Computing Surveys, Vol. 13, No. 2, June 1981.

[BERN84] Bernstein, P.A., and Goodman, N., "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases", ACM TODS, Vol 9, No. 4, December 1984.

[BLAU85] Blaustein, B.T., and Kaufman, C.W., "Updating Replicated Data during Communication Failures", Proc. Eleventh International VLDB Conference, Stockholm, August 1985.

[CHAN85] Chandy, K.M., and Lamport, L., "Distributed Snapshots: Determining Global States of Distributed Systems", ACM Transactions on Computer Systems, Vol 3, No. 1, February 1985.

[EAGE81] Eager, D.L., "Robust Concurrency Control in Distributed Databases", Technical Report CSRG #135, Computer Systems Research Group, University of Toronto, October 1981.

[ELAB85] El Abbadi, A., Skeen, D., and Cristian, F., "An Efficient, Fault-Tolerant Protocol for Replicated Data Management", Proc. 4th ACM SIGACT-SIGMOD Symp, on Principles of Database Systems, pages 215 -

228. Portland, Oregon, March 1985.

[ELLI83] Ellis, C.S., "Extendible Hashing for Concurrent Operations and Distributed Data", ACM Symp. on PODS, 1983.

[GARC83] Garcia-Molina, H., "Using Semantic Knowledge for Transaction Processing in a Distributed Database", ACM TODS, 8, 2, June 1983.

[GIFF79] Gifford, D.K., "Weighted Voting for Replicated Data", Proc. 7th ACM SIGOPS Symp. on operating Systems Principles, pages 150 - 159. Pacific Grove, CA, December 1979.

[GRAY78] Gray, J., "Notes on Data Base Operating Systems", in Operating Systems: An Advanced Course, Springer-Verlag, 1978, pp393-481.

[GRAY81] Gray, J.N., "The Transaction Concept: Virtues and Limitations", Proceedings Seventh International VLDB conference, September 1981.

[KORT83] Korth, H. F., "Locking Primitives in a Database System", Journal of the Association of Computing Machinery, Vol. 30, No. 1, Jan 1983.

[KUNG81] Kung, H. and Robinson, J., "On Optimistic Methods for Concurrency Control," TODS, June 1981, pp 213-226.

[MOHA85] Mohan, C., et. al., "Lock Conversion in Non-Two-Phase Locking Protocols", IEEE Transaction on Software Enginering, Vol 11, No. 1, 1985.

[ONEI86] O'Neill, P., "The Escrow Transactional Method", ACM TODS, Vol 11, No. 4, December 1986.

[REUT82] Reuter, A., "Concurrency on High-Traffic Data Elements", ACM Symposium on PODS, 1982.

[SARI85] Sarin, S. K., et. al., "System Architecture for Partition Tolerant Distributed Database", IEEE Transactions on Computers, Vol. C-34, No. 12, December 1985.

[STON79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed Ingres", IEEE Transactions on Software Engineering, Vol 3, No. 3, May 1979.

[THOM79] Thomas, R. H., "A Majority Consensus Approach to Concurrency Control for multiple-copy databases," TODS, June 1979.

[WEIH84] Weihl, W., "Specification and Implementation of Atomic Data Types", Ph.D. Thesis, MIT, March 1984.