VLDB Journal, 3, 325-353 (1994), Peter Scheurermann, Editor
©VLDB

The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems

Daniel Barbará-Millá and Hector Garcia-Molina

Received August, 1992; revised version received June, 1993; accepted June, 1993.

Abstract. Traditional protocols for distributed database management have a high message overhead; restrain or lock access to resources during protocol execution; and may become impractical for some scenarios like real-time systems and very large distributed databases. In this article, we present the demarcation protocol; it overcomes these problems by using explicit consistency constraints as the correctness criteria. The method establishes safe limits as "lines drawn in the sand" for updates, and makes it possible to change these limits dynamically, enforcing the constraints at all times. We show how this technique can be applied to linear arithmetic, existential, key, and approximate copy constraints.

Key Words. Consistency constraints, transaction limits, serializability.

1. Introduction

Traditional protocols used to manage distributed data (e.g., two-phase commit) require one or more rounds of messages, restrain or lock access to resources during the protocol execution, and have a high overhead. Moreover, these protocols could render data unavailable during failure periods, decreasing system availability. This may be impractical for some scenarios, like those involving very large distributed databases or real-time systems. In this article, we propose an alternative protocol that overcomes these problems by using explicit consistency constraints as the correctness criteria. The method gives the involved nodes substantial autonomy for performing changes to individual data items.

Consider an application where "items" are stored in two separate locations. Assume that there must be sufficient stock of these items between the two locations. This could, for instance, correspond to a military application in which planes are

Daniel Barbará-Millá, Ph.D., is Senior Scientist, Matsushita Information Technology Laboratory, 2 Research Way, Princeton, NJ 08540; Hector Garcia-Molina, Ph.D., is Professor, Department of Computer Science, Stanford University, Stanford, CA 94305.

stationed at two different bases, with the requirement that at all times the total number of stationed planes cannot be below some specified limit (perhaps out of fear of being short of planes in case of an attack). Or the application could correspond to a retail business in which the "parts" stock at two warehouses must be maintained above a certain limit all the time. Transactions will try to withdraw or add to the stock, and the system should verify that the constraint is obeyed at all times. In the military application, planes can be put out for maintenance, or sent off to missions, but the minimum number of planes should be kept stationed at all times.

In the following example we have two nodes, each storing the value of the stock kept at a certain location. Let A and B be the data items at locations a and b, respectively. Let the constraint be $A + B \geq 100$. Consider a typical transaction that attempts to withdraw Δ units from A:

```
If $A - \Delta + B < 100$ then abort
else $A = A - \Delta$
```

In a conventional transaction processing system, such a transaction would have to lock data item B at location b, and A at a, verify whether the updated value satisfies the constraint and, in that case, update A and follow a two-phase commit protocol to ensure that the transaction commits. Notice that this would require two rounds of messages, and would lock and limit access by other transactions to A and B during protocol execution, resulting in a high overhead. The protocol could also render the data unavailable if one node or the network fails during execution, thus decreasing the availability of the system. This can occur because the two-phase commit protocol can block (i.e., after a failure the nodes may not be able to determine the outcome–commit or abort–of the transaction). Therefore, the nodes cannot release the locks and other transactions cannot run.

Alternatively, we could have a variable A_l at node a that acts as a limit, and state that transactions can continue withdrawing units of A as long as the final value of A remains above $A_l (A \geq A_l)$. Similarly, a variable B_l will be stored in node b, serving as a limit for the updates made to $B (B \geq B_l)$. (Think of A_l and B_l as "lines drawn in the sand.") The transactions will produce correct results as long as we ensure that $A_l + B_l \geq 100$. Notice that now, A (or B) can be modified by a transaction without involving the other node, as long as the updated value remains larger than the limit A_l (B_l). In these cases, what used to be a global transaction has become a local one, so there is no need for global concurrency control or a two-phase commit protocol. This increases data availability since transactions of this type may run even if the other node (or the network) is unavailable. One would expect that in many applications there is often slack in the constraints, so that in a very large number of cases transactions will be able to run locally as illustrated. In the aircraft example, say each base has 80 aircraft and that a total of 100 aircraft must be kept between the two bases. If we set $A_l = B_l = 50$, we satisfy the global constraint, leaving each base with a slack of 30 aircraft. Thus, each base could dispose of 30 aircraft without consulting the other.

The limits do not have to be static, but can change over time as needed. However, the changes have to be made in such a way that the constraint $A_l + B_l \geq 100$ is obeyed at all times. This is the goal of the *demarcation protocol* presented here. The protocol is designed for high autonomy. Clearly, some changes in limits must be delayed because they are not "safe"; however, these delays do not block safe changes or transactions that operate within the current limits. Besides a protocol for changing limits, we also need a *policy* for selecting the values of the desired new limits. (In our aircraft example, do we set $A_l = B_l = 50$ or do we select $A_l = 30$, $B_l = 70$?) Policies will also be discussed in this article.

Our approach does *not* guarantee global serializable schedules (neither conflict nor view serializable). Local executions are serializable, however, because each node uses conventional techniques (e.g., two-phase locking) to run local transactions. Our approach does guarantee that inter-node constraints, which we assume are given, are satisfied. A conventional global concurrency control mechanism, on the other hand, would guarantee globally serializable schedules and the satisfaction of all constraints without the need to explicitly give them to the system.

One could argue that having to explicitly list the global consistency constraints is a disadvantage of our method. We can counter this argument, first by noting that with conventional approaches programmers still have to know and understand consistency constraints to write transactions (a transaction must preserve all constraints). Second, by knowing the constraints, the system can exploit their semantics, yielding better availability and performance.

Third, we are only talking of specifying constraints that span more than one node (local control ensures that local constraints are satisfied.) If we look at inter-node consistency constraints in practice, we see that they tend to be very simple. For instance, it is very unlikely that we would encounter an application with employee records in New York and an index to those records in Los Angeles. Data that are closely interrelated tend to be placed on a single node. (In a parallel database machine or in a local cluster, there may be complex inter-computer constraints. But in this case, the autonomy of each computer and network delays are not the critical issues. We are focusing on geographically distributed systems.) If we look at the types of constraints that are found in databases (Date, 1983), we claim that the following ones are the most likely to involve data stored in different nodes of a distributed system:

1. Linear arithmetic inequalities. Example: the available funds for a computer at an ATM machine should be less than or equal to the actual balance of an account.

2. Linear arithmetic equalities. Example: the hourly wage rate at one plant must equal the rate at another plant.

3. Referential integrity constraints. Example: if an abbreviated customer record exists on one node, then the full customer record must exist at headquarters.

4. Object copies. Example: the employee benefits brochure must be a copy of the brochure at the personnel office.

Our main focus here is on linear arithmetic inequalities. If an arithmetic equality is tight (e.g., $A = B + \delta$, then maintaining it will be expensive. Every change involves two-phase commit or the equivalent, because B must change immediately each time A changes. However, in many applications a tight equality can be treated as an inequality (e.g., $|A - B - \delta| \leq \varepsilon$. This is simply the two constraints $A - B - \delta \leq \varepsilon$ and $-A + B + \delta \leq \varepsilon$, which can be handled via our demarcation protocol. In Section 7 we show that the same principles that are used for arithmetic constraints can be applied to existential, key, and copy constraints.

This article is organized as follows. Section 2 lists related research. Section 3 offers additional examples and an informal description of our approach. The protocol and a policy under a particular arithmetic inequality are presented in Section 4. Section 5 presents an analytical model to evaluate the protocol. The generalization to arbitrary arithmetic constraints is in Section 6. Section 7 discusses other types of constraints. Section 8 discusses serializability issues related to the demarcation protocol. Section 9 presents the conclusions.

2. Related Research

There has been a lot of recent interest in trying to reduce the delays associated with conventional transaction processing and on exploiting semantics. In this section, we briefly summarize research that is related to the demarcation protocol.

The idea of setting limits for updates has been suggested informally many times (Hammer and Shipman, 1980; Davidson, 1982). These ideas were formalized by Carvalho and Roucariol (1982) in the context of enforcing assertions in distributed systems. Their limit-changing protocol is more general than ours, but is substantially more complex and, we believe, less practical.

First, the protocol of Carvalho and Roucariol (1982) exchanges the values of the limits between the nodes, instead of exchanging the increments/decrements, as the demarcation protocol does. This makes the protocol, in principle, vulnerable to the order of the messages. This problem is corrected by the use of auxiliary variables or counters that keep track of the time at which the messages were sent. A message with a timestamp lower than the value of the local counter is always discarded. This way of proceeding makes the changes sequential. That is, if a node, for any reason, decides to make two consecutive changes to a limit, it cannot effectively propagate the second change until the first one has been acknowledged by every other node involved in the treaty. On the other hand, the demarcation protocol is immune to delays in the message or the order of reception of them and would allow the propagation of any number of consecutive changes to be made without having to wait for acknowledgments.

The Carvalho and Roucariol (1982) protocol forces each node to maintain in memory one treaty variable per node involved. One of them indicates the compromise made by the node itself, while the others are images of the compromises made by the other nodes. An instantiation of the protocol is needed to maintain each treaty variable. For instance, in a two-node network, node i would maintain two treaty variables W_i^j and M_i^j, indicating to node i the compromises taken by itself and node j, respectively. Meanwhile, node j would maintain W_j^i and M_j^i. The variables M would be images of the variables W. An instantiation of the protocol would maintain the assertion between W_i^j and M_i^j, while other instantiation would do the same for pair W_j^i and M_i^j. Once a node makes a change in its limit (or treaty variable), it should propagate the change to *all* nodes that maintain images of this variable, and it cannot effectively propagate a new change until all those nodes acknowledge the reception of the original message. This could be a burden if the number of nodes involved is large. On the other hand, the demarcation protocol maintains a single limit per node (per constraint) and is flexible enough to allow pairwise treaties (i.e., once a node has changed a limit, it may only have to communicate the increment/decrement to one of the nodes involved, most likely the node that requested the change in the first place, greatly reducing the number of messages involved. Furthermore, Carvalho and Roucariol (1982) do not discuss policies for changing limits. Consequently, the protocol presented here is a more efficient and practical way of dealing with distributed transaction management than their protocol. Moreover, important details (e.g., when and how the changes on the limits take place) are missing from their discussion.

The demarcation protocol is loosely related to O'Neil's (1986) escrow mechanism. This technique was devised to support high-speed transaction updates by enforcing integrity constraints without locking items. However, the mechanism was designed to be used only in a single database management system. (O'Neil proposed the application to the management of replicated data as a research topic.) Kumar and Stonebraker (1988) also have proposed a strategy for the management of replicated data based on exploiting application semantics. They present an algorithm to implement the constraints $B > B_{min}$ and $B < B_{max}$. However, their technique relies heavily on the commutativity of the transactions involved, and does not generalize to arbitrary arithmetic constraints. Soparkar and Silbershatz (1990) developed a protocol to partition a set of objects across nodes. A node may "borrow" elements from neighbors. This approach does not deal with arbitrary constraints, but it does guarantee serializable global schedules. Pu and Leff (1991) introduced *epsilon-serializability,* a correctness criterion that allows temporary and bounded inconsistency of replicas. Krishnakumar and Bernstein (1992) developed a protocol called the Generalized Site Escrow algorithm that uses quorum locking and gossip messages to dynamically allocate resources among nodes while providing a high degree of site autonomy and throughput.

The notion of quasi-copies was defined by Alonso et al. (1990) as a way of managing copies that may diverge in a controlled fashion. The goals of their work were the same as ours for the demarcation protocol. However, quasi-copies are not useful for arithmetic constraints. For managing copies, the notion of quasi-copies is more flexible in some ways. For instance, one can specify that a copy must be equal to some value that the primary had within the last 24 hours. This type of constraint is not handled by the demarcation protocol. On the other hand, updates may occur at a primary location. The demarcation protocol allows updates at any site containing an arithmetic value (Section 7.3).

There are other articles that deal with weaker notions of serializability and use of application semantics (Fischer et al., 1982; Garcia-Molina, 1983; Lynch et al., 1986; Korth and Speegle, 1988; Du and Elmagarmid, 1989; Fernández and Zdonik, 1989).

3. Examples

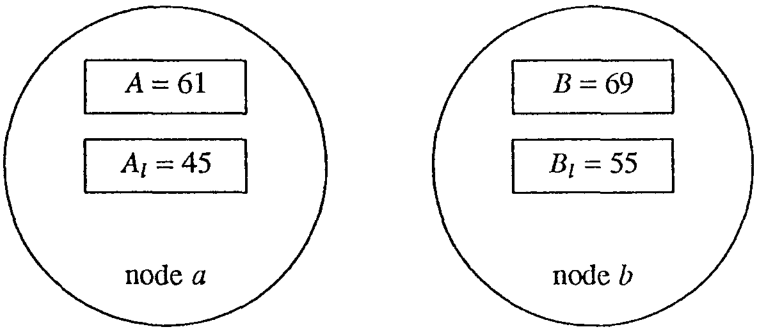
In this section, we present two examples that illustrate how the demarcation protocol works and the kinds of problems that are to be dealt with. First, we return to the example presented in Section 1. The proposed method of operation imposes some restrictions. As pointed out, a transaction that attempts to lower A (or B) under the limit A_l (B_l) would be aborted. The only way to run such a transaction would be first to get the site to lower its limit. Since this is not a "safe" operation (lowering A_l may violate the constraint $A_l + B_l \geq 100$), it can only be achieved by asking the other node to raise its limit first. The only safe operation in this example is to raise the limit above the current value.

To see how limits can be changed, assume that $A = 61$, while $B = 69$, and the limits are chosen initially to be $A_l = 45$ and $B_l = 55$. Figure 1 shows the setting of the base scenario for this example. Notice that transactions at node a can update A without further intervention from b, as long as the final value remains greater than or equal to 45. The same is true for b and B, as long as B remains greater than or equal to 55. Assume now that for some reason, node a wants to raise its limit A_l to 50. Since this is a safe operation, a can go ahead. Node a will send a message to b, informing it of the increment to A_l by 5 units. Upon receipt of this, b may lower B_l by 5 to 50 (an originally unsafe operation). No reply to a is necessary.

If a wants to lower A_l, the procedure is not as easy. Lowering the limit is an unsafe operation in this case, so node a would have to send a request to b, asking it to raise B_l by the necessary amount. Node b is free to reject this request. However, if it does honor it by raising B_l, it will send a a message, and only then could a lower A_l. If the nodes follow this protocol, it can be assured that at all times $A_l + B_l \geq 100$.

This is essentially the way the demarcation protocol operates. Whenever a node wishes to perform an unsafe operation, it requests that the other node perform a corresponding safe operation and waits for notification. Notice that the demarcation protocol is not two-phase commit. (The decision to give another node slack by increasing a limit is made by one node only.) The nodes are still autonomous and

Figure 1. Base scenario for sufficient supply problem



there is no need for locking remote resources. While limits are being changed, transactions that modify A or B can still run (as long as $A \geq A_l$, and $B \geq B_l$.)

There are still two issues to be discussed. The first is the establishment of a policy for invoking the protocol. The policy is orthogonal to the protocol, and the changes can be triggered at any time. For instance, the changes could be triggered whenever A (B) gets too close to A_l (B_l). The second issue is the selection of the new limits. Each node needs a formula for computing the new limits when a change is to be made. In this example, it may be desirable to split the "slack" evenly (i.e., to make the distance between A_l and A equal to the one between B_l and B).

For instance, using the initial values of Figure 1, consider a transaction that updates B to 57. Say that $B - B_l = 2$ is considered "close," so the change mechanism is triggered. However, node b does not know how to split the slack time since it does not know the value of A, and it cannot lower B_l anyway. Therefore, b sends a message to a, requesting that A_l be raised, and including the current value of B. Upon receipt of this message, a knows that $B = 57$ and $A = 61$. The slack is $A + B - 100 = 18$. Subtracting half of this from each value we get $A_l = A - 9 = 52$, and $B_l = B - 9 = 48$. These are the new limits that split the slack evenly. Finally, a can safely increase A_l from 45 to 52 (increment of 7), sending a message to b, allowing it to decrement B_l by 7 to 48.

So far we have only discussed one type of constraint (i.e., $A + B \geq \delta$). Do the ideas generalize to other types of arithmetic inequalities? Fortunately, they do. For any constraint, some operations are safe while others are not. To illustrate, consider the following example: Ensuring that project expenses E do not exceed budget B. Assume that we store B at node b and E at node e. Node b could be located at company headquarters, while e is at the project location. We require that $E \leq B$ at all times. Both the expenses and the budget are updated over time. For the demarcation protocol, we will keep two limits E_l and B_l such that $E \leq E_l$, $B_l \leq B$, and $E_l \leq B_l$. In the supply example, it was safe to increase the limits of both variables. In this budget problem, the limit E_l can be *decreased* safely by *e*, while B_l can be *increased* safely by *b*. On the other hand, the operations of incrementing E_l at *e*, or decrementing B_l at *b*, are unsafe. Notice again that once one node performs a safe operation, it leaves room for the other to perform what was originally an unsafe change. Consider that initially $E = 0$, $B = 20$ and that limits E_l and B_l have been set to 10. As long as E stays under 10, node *e* is free to modify E without consulting *b*. Similarly, node *b* can lower B to 10.

4. Protocol and Policies

In this section, we present the demarcation protocol and its associated policies. To simplify the explanation, we assume a particular constraint, $A \leq B + \delta$. We also show a particular policy choice for splitting slack, although many others are possible (e.g., Section 6).

4.1 The Demarcation Protocol

The demarcation protocol consists of two operations, one for changing a limit and one for accepting the change performed by the other node. Recall that the constraint we are dealing with is $A \leq B + \delta$. Let the predicate SAFE(X, σ), where X is either A or B, and σ is a desired change in value, be defined as follows:

$$\text{SAFE}(X,\ \sigma) = \textit{if}\ (X = A \text{ and } \sigma \leq 0) \text{ or } (X = B \text{ and } \sigma \geq 0)\ \textit{then}\ \text{TRUE}\ \textit{otherwise}\ \text{FALSE}$$

Essentially, SAFE is TRUE when we decrement the limit of A or increment the limit of B. The other two operations are unsafe. We also define the following predicate to signal when the change in a limit exceeds its data value:

$$\text{LIMIT\_BEYOND}\ (X,\ \sigma) = \textit{if}\ (X = A \text{ and } A_l + \sigma < A) \text{ or } (X = B \text{ and } B_l + \sigma > B)\ \textit{then}\ \text{TRUE}\ \textit{otherwise}\ \text{FALSE}$$

When we refer to one of the values as X, we will use Y to refer to the complementary variable in the constraint $A \leq B + \delta$ (i.e., $Y = A$ if $X = B$, and vice versa). We use the notation $N(X)$ for the node holding X. The demarcation protocol is composed by two procedures: `change_limit()` and `accept_change()`:

`P1`: *The Demarcation Protocol*

```
change_limit($X,\sigma$)
    if LIMIT_BEYOND ($X,\sigma$) then
        abort the change
    else
        if not SAFE ($X,\sigma$) then
            send message to $N(Y)$ requesting it to perform change_limit($Y,\sigma$)
        else
        { $X_l \leftarrow X_l + \sigma$;
        send message to $N(Y)$ requesting it to perform accept_change($Y,\sigma$).
accept_change($Y,\sigma$)
    $Y_l \leftarrow Y_l + \sigma$.
```

Conventional database techniques should be used at each node to make changes in the limits and variables atomic and persistent. For instance, the values of the limits should not be lost due to a node failure. Loss of messages is undesirable but does not cause the constraint to be violated. For example, say $N(A)$ decreases its limit by 5 and sends an `accept_change` message to $N(B)$. If this message is never delivered, B_l will be 5 units higher than need be. This is safe, but it means that $N(B)$ will be unable to "use" these 5 units. Thus, for proving our protocol correct (Theorem 4.1) we make no assumptions about message delivery. However, in practice it is desirable to use persistent message delivery (messages are delivered eventually, without specifying how long this might be). Also note that since messages from different calls to `change_limit` only include decrements/increments, they need not be delivered in order at the other node.

For simplicity, we have assumed that nodes are always willing to cooperate with their partners. In reality, nodes need not comply with `change_limit` or `accept_change` requests. When node a gets a request from b to decrease A_l by 10 units, a is free to ignore the request, or to decrease A_l by whatever amount it wishes. Similarly, when a receives `amount_change`(A, 10), it may increase A by any amount up to 10. Of course, in most cases it is advantageous for a to perform the full increment, as indicated by the code. This is what we assume here.

Theorem 4.1: The demarcation protocol ensures that at all times $A_l \leq B_l + \delta$, assuming that the system starts with limits A_l^0 and B_l^0, where $A_l^0 \leq B_l^0 + \delta$.

Proof: All the increments or decrements to limits are done by adding a value v_i or subtracting a value u_i to the old limits, where i is simply an ascending index. For data value A, v_i^A represents a change performed via the `accept_change` call, while u_i^A represents a change made via `change_limit`. For B, v_i^B represents a change made using `change_limit`, and u_i^B represents one made using `accept_change`. At any time, we have:

$$A_l = A_l^0 + \sum_{i \leq k_l} v_i^A - \sum_{i \leq k_2} u_i^A \text{ and } B_l = B_l^0 + \sum_{i \leq k_3} v_i^B - \sum_{i \leq k_4} u_i^B,$$

where k_1 and k_3 are the indexes of the last increments seen by nodes $N(A)$ and $N(B)$, respectively, and k_2 and k_4 are the indexes of the last decrements seen by nodes $N(A)$ and $N(B)$, respectively. □

Every increment v_i^B performed by `change_limit` produces an equivalent increment v_j^A done by `accept_change`. The increments for A may not be done in the same order as for B. However, the increment v_i^B is always done before the increment of the same magnitude v_j^A is done. Thus,

$$\sum_{i \leq k_1} v_i^A \leq \sum_{i \leq k_3} v_i^B$$

By a similar argument,

$$\sum_{i \leq k_4} u_i^B \leq \sum_{i \leq k_2} u_i^A$$

We can combine these two inequalities with $A_l^0 \leq B_l^0 + \delta$ by adding all left-hand sides to all right-hand sides, obtaining that at all times $A_l \leq B_l + \delta$ □

Note, incidentally, that δ plays a limited role in the protocol (and proof). As long as A_l and B_l are initially δ units apart, the protocol ensures that they continue to be that far apart.

Corollary 4.1 Using the demarcation protocol, we ensure that $A \leq B + \delta$ at all times.

Proof: The line in `change_limit` that checks LIMIT_BEYOND(X, σ) ensures that $B \geq B_l$ and $A \leq A_l$. Then, by Theorem 4.1, the result follows. □

Corollary 4.2 Suppose that $A_l^0 = B_l^0 + \delta$, that all update activity stops, and that all messages have been delivered. Then $A_l = B_l + \delta$ (note equality).

Proof: Same as the proof for Theorem 4.1 except that after all messages are delivered,

$$\sum_{i \leq k_1} v_i^A = \sum_{i \leq k_3} v_i^B \text{ and } \sum_{i \leq k_4} u_i^B = \sum_{i \leq k_2} u_i^A$$

□

4.2 Policies

The policies associated with the demarcation protocol specify when to initiate limit changes, how to compute new limits, and what to do in case a transaction tries to change the data value beyond its limit. We describe here the framework for such policies, and present some choices for them. However, the reader should bear in mind that other policies exist.

We begin by explaining how a transaction will actually perform changes on the data items. To change a value, a transaction will use a system call `change_value`(X, θ), where X is the data item and θ is the amount (positive or negative) to change. Within this call, we will have invocations of three policies. The first will be triggered whenever the change would exceed the limit. The second will be fired up when the final value gets too "close" to the limit. The last one will be triggered if the final value gets too "far" from the limit. These policies are implemented by procedures associated with the constraint that we are enforcing. Since a data value may be involved with more than one constraint (and thus, have more than one limit), we will have to test the limits on a per constraint basis. We will denote the constraints by the symbol Φ_j, where $1 \leq j \leq m$ and m is the number of constraints. Up to this point, we have only discussed the constraint $A \leq B + \delta$, but we are now generalizing to emphasize that the policies are constraint dependent.

Before presenting the change_value procedure, we introduce the following predicate, for the case Φ_j with the constraint $A \leq B + \delta$. (A generalization is presented in Section 6.)

```
VALUE_BEYOND $(X,\theta)$ = if $(X = A$ and $A + \theta > A_l)$
                            or $(X = B$ and $B + \theta < B_l)$
                            then TRUE otherwise FALSE
```

This predicate is analogous to LIMIT_BEYOND, in this case checking whether a change would violate the limit constraint.

The system calls for changing a data value as follows. Again, X refers to the variable being changed (A or B is our sample constraint). The limit for X under constraint Φ_j is X_{lj}. Parameter T_{code} is a pointer to the code that runs the calling transaction and is explained below. (Note that the procedure below is valid for any constraint, not just our sample.)

```
change_value $(X,\theta,T_{code})$
   for each constraint $\Phi_j (1 \leq j \leq m)$ in which $X$ is involved do
      { if $\Phi_j$. VALUE_BEYOND $(X,\theta)$ then
         { Fire up process $\Phi_j$. policy1$(X,\theta,T_{code})$ at $N(X)$
         abort calling transaction; }
      if $|X + \theta - X_{lj}| < \Phi_j.\varepsilon$ then
         Fire up process $\Phi_j$, policy2$(X,\theta,T_{code})$ at $N(X)$;
      if $|X + \theta - X_{lj}| > \Phi_j.\beta$ then
         Fire up process $\Phi_j$. policy3$(X,\theta,T_{code})$ at $N(X)$ }
   $X \leftarrow X + \theta$.
```

The procedures Φ_j.*policy*1, Φ_j.*policy2*, and Φ_j.*policy3* are associated with the constraint Φ_j. The first policy is invoked when a change exceeds one of the limits. In some cases *policy1* may be null, but in other cases it may be important to initiate some action, such as trying to increase the limit. The code pointer parameter T_{code} is useful for restarting the transaction once the limit has been changed. An alternative would be to move the "abort transaction" command in change_value into *policy1*. This way, *policy1* could choose between aborting the transaction or simply delaying it until the limit has been successfully changed. Incidentally, notice that if the abort option is taken, there is a potential starvation problem. That is, by the time an aborted transaction T_1 is rerun, the slack generated by its limit change request may have been used up by another transaction that needed it. If this is a problem, the system can "reserve" the slack generated by T_1 for its exclusive use.

The second procedure deals with the case where the value is getting close to the limit; close is defined by the constraint-specific constant $\Phi_j.\varepsilon$. It may be desirable at this point to initiate a change in limits. For instance, if a military base is running low on planes, it may be a good idea to renegotiate its limits with the other base. The third procedure handles the case in which the value is getting too far away from the limit, as defined by constant $\Phi_j.\beta$. In our example, if a base has too many planes, it may wish to notify the other base to arrange new limits.

A fourth policy is needed to cover the case where the `change_limit` procedure encounters a change that exceeds the data value. In Section 4.1, we opted for aborting the change in general, but we can have a policy that decides what action is to be taken. The procedure (Φ_j.*policy4*) can be invoked when the LIMIT_BEYOND check in `change_limit` detects a violation. Due to space limitation we will not present the modified `change_limit` procedure here; a more general version is presented in Section 6.

The policies must also implement some form of load control. For example, say the system has reached a point where $A = A_l$ and $B = B_l$. Transactions that attempt to increase A will repeatedly try to increase A_l by sending a message to $N(B)$ to increase B_l. Since B_l cannot be raised, all these attempts will fail. Instead of wasting $N(B)$'s time $N(A)$ may remember how many times it has attempted to change the limits, and thus decide to wait for a given period of time before trying again.

Finally, there is the issue of how to compute the limits. A formula should be agreed upon to compute the new values when needed. We illustrate one possible formula for splitting the available slack, for the constraint $A \leq B + \delta$. (A generalization is presented later.)

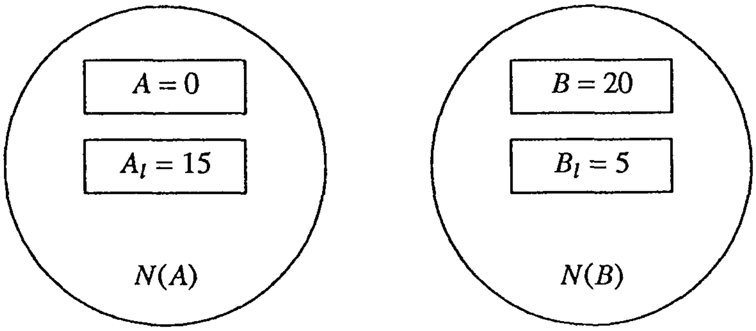
$$A_l = A + (B - A + \delta)k$$

$$B_l = A_l - \delta \qquad (1)$$

Equation 1 is derived by computing the slack between A and B, and by giving a fraction k of it to variable A as the "room to move up." The remaining $1 - k$ of the slack is given to variable B. By setting up k, one can tune the system to favor or constrain one of the two variables. By setting $k = 0.5$, one divides the interval evenly. Notice, however, that computing the new limits by using Equation 1 requires information about both variables A and B, information that neither of the nodes has. However, we can design *policy2* and *policy3* to overcome this problem as follows:

```
$\Phi_j$.policy2($X$,$\theta$,$T_{code}$)
    send message to $N(Y)$ node, requesting it to perform $\Phi_j$.split_slack($X$)
$\Phi_j$.policy3($X$,$\theta$,$T_{code}$)
```

Figure 2. Initial scenario



```
    send message to $N(X)$ node, requesting it to perform $\Phi_j$.split_slack($X$)
$\Phi_j$.split_slack($X$)
    ***local variable is $Y$; value of remote variable is parameter $X$;***
    compute new limits, $Y_{lj}^{new}$ and $X_{lj}^{new}$, using Equation 1 and $X$,$Y$ values.
    let $\sigma = Y_{lj}^{new} - Y_{lj}$;
    if SAFE($Y$,$\sigma$) then invoke $\Phi_j$.change_limit($Y$,$\sigma$)
    else send message to $N(X)$ node, requesting it to perform $\Phi_j$.split_slack($Y$)
```

Notice that we prefixed the function *split_slack* with the constraint identification Φ_j. In general, the way the slack is split will depend on the specific constraint. (The generalization of Equation 1 is given in Section 6.) It may be desirable to include a timestamp in a *split_slack* message, so that the receiving node may discard messages that took "too long" in transit and represent stale data. Also note that we ignored load control issues in these policies.

To illustrate how limit changing and slack splitting work, consider the following example. Assume that we are using the single constraint $A \leq B + \delta$ so that all our constraints and policies refer to it. Initially $A = 0$, $B = 20$ with $\delta = 10$, $\beta = 20$, and $\varepsilon = 2$. The limits have been set to $A_l = 15$ and $B = 5$, assuming that we are using $k = 1/2$. Figure 2 shows the initial scenario.

Now assume that a transaction calls `change_value` at $N(B)$ to update B to 30. (Notice that this update is allowed since $B \geq B_l$.) Since $B - B_l > \beta$, the request to change limits is initiated by $N(B)$, by triggering *policy3.* A message is sent to $N(A)$, requesting it to perform `split_slack(`B `= 30)`. Using Equation 1, node $N(A)$ computes the new limits, obtaining a value of 20 for A_l. (The slack is $30 - 0 + 10 = 40$; half of this added to A to give the desired new limit.) To obtain $A_l = 20$, $N(A)$ must add 5 to the current limit; since this is an unsafe increment, it does not perform the change. Instead, it sends a message to $N(B)$, requesting it to perform `split_slack(`A `= 0)`. Node $N(B)$ will compute the same limits, and this time the change in B_l will be performed there, updating it to 10. Node $N(B)$ will send a message to $N(A)$, requesting accept_change(A,5). Finally $N(A)$ will raise A_l to 20. This example illustrates the case where split_slack is called twice. In other cases, split_slack is only called once (e.g., from the initial scenario, A is updated to 15).

Now consider a scenario with concurrent updates. Starting with the values of Figure 2, assume that both $N(A)$ and $N(B)$ complete transactions. The transaction at $N(A)$ updates A to 13, while the one at $N(B)$ updates B to 5. Both nodes send their new values to the other node, since both want to change their limits. Upon receipt of the new value $B = 5$ at $N(A)$, the limit A_l is computed to be 14. Therefore, the change is made and a message is sent to $N(B)$, requesting it to perform accept_change(B,-1). In the meantime, $N(B)$ processes the first message from $N(A)$, computing a desired new limit B_l^{new} of 4. Since this change is not safe, a message is sent to $N(A)$, requesting that the change is made there first. Next, each node receives the second message generated by its partner. When the accept_change(B,-1) arrives at $N(B)$, B is updated to 4. When the second split_slack(B=5) message arrives at $N(A)$, it is ignored. (In Equation 1, $A = 13$, $B = 5$ implies that A_l should be 14, but that is already the value of the limit, so nothing is done.) The final values are thus $A_l = 14$, $B_l = 4$, which evenly split the slack. Notice how the limits converge to the correct values, without any tight coordination. Each node still makes decisions autonomously.

In general, it is difficult to prove formal properties for the policies, first because they can be arbitrary programs, and second because we purposely do not wish to guarantee any properties that may hurt autonomy. For instance, for slack splitting, one may be tempted to prove that the selected limits are indeed the ones that split the current slack. However, there is no such thing as the "current slack," because the nodes may autonomously change A and B at any time (within limits). To enforce such a property, we would have to restrict updates in one way or another, which is clearly undesirable.

In the sample of slack splitting policies, we simply assure that the value received in a *split_slack* message is current, and act accordingly. If indeed it is current, then the slack is split properly; if not, the selected limits will be out of date. But remember that no matter what the policies do, the underlying constraints are always guaranteed by the demarcation protocol.

In closing this section, we clarify two subtle points. One is that there are certain global transactions that cannot be run as a sequence of local transactions, with the help of the demarcation protocol. To illustrate, consider the constraint $A + B > 100$ with the values $A = 50$ and $B = 60$. We cannot run the transaction

$$\begin{aligned} T_1\colon\ & A \leftarrow A - 50 \\ & B \leftarrow B + 50 \end{aligned}$$

as a sequence of two local transactions, since this would temporarily invalidate the constraint. Global transactions that must temporarily invalidate constraints need to be run with a locking protocol (which can be added on top of the demarcation protocol.) Incidentally, note that if we reverse the order of the two steps in T_1, then it does not violate the constraints and no locking is needed.

The second point relates to the checks that transactions must perform. In general, a transaction must preserve the consistency of the database. Thus, if T_1 is going to modify A in our example, it should check that the constraint $A + B > 100$ is preserved. However, with the demarcation protocol, enforcement of the constraint is handled by the system, so the transaction no longer needs to check if its update may violate the constraint. If T_1 attempts to violate the constraint, it will be aborted.

5. Analytical Evaluation

As discussed in the introduction, the main advantage of the demarcation protocol is the added autonomy and fault tolerance that it provides. However, a second advantage may be improved performance during normal operation, because for some transactions (hopefully the majority) no communication between nodes is needed for commit. In this section, we present a comparative performance analysis of the demarcation protocol versus the standard two-phase commit protocol. The protocols will be evaluated with respect to throughput and response time.

Analyzing the performance of a transaction processing system, under either the demarcation or the two-phase commit protocol, is very difficult. Predicting performance depends on many unknown parameters and issues: How many conflicts will there be among transactions? How expensive is it to get a lock or check a limit? How often will limits have to be changed? Given the complexity of these issues, we select two simple, but key scenarios: One where there are no conflicts and we can evaluate the maximum system throughput, and one where there is a single remote conflict. We believe these two scenarios will be the most common and, thus, it will be instructive to study performance in those cases.

The analysis assumes a system with two nodes, each one holding a portion of the database. In the first scenario, no conflicts arise. That is, in the demarcation protocol, all transactions are able to run to completion, without triggering any change of limits. In the two-phase commit protocol, there are no transaction blocks due to lock contention. This extreme case maximizes throughput for both protocols; therefore we use that metric for comparison. In the second scenario every transaction encounters a conflict when it is run for the first time. In the two-phase commit protocol, the transaction waits for a remote lock before it can proceed. In the demarcation protocol, the predicate VALUE_BEYOND is TRUE when the transaction is submitted for the first time. The transaction is aborted, and a change of limits is triggered. After the limits are changed, the transaction is rescheduled and run to completion. In this case, we compute the response time for a transaction under both protocols.

Table 1 presents the parameters used in the analysis. The second column explains the meaning of each symbol. To illustrate the performance gains of the demarcation

Table 1. Parameters

Parameter	Meaning	Case 1	Case 2
t_s	context switch	100	100
t_l	request locks	100	100
t_{cl}	check limits	300	300
t_T	running time	1,000	100,000
t_r	release locks	50	50
t_m	send/receive message	1,000	10,000
t_d	message delay	100,000	100,000
t_{ch}	change limits	300	300
t_x	scheduling delay	10,000	10,000

Times in microseconds

protocol, we consider two "representative" parameter settings, given in columns 3 and 4. There are obviously a great many possible settings, and by selecting only two of them we are leaving many out. However, to visualize the performance gains, we think it is useful to study at least two concrete settings at opposite ends of the transaction cost spectrum. The first setting represents a system with lightweight transactions and low cost for sending a message. The second system represents a system with larger transactions and high message costs (i.e., a more conventional transaction processing system). The only parameters that change in the two cases are the running time and the time to send/receive a message. All the values are in microseconds.

In the first scenario (no conflicts), a transaction running under the demarcation protocol arrives at the node and is scheduled for processing. Before the transaction starts, a request message is received (t_m), and a context switch takes place (t_s). Then the transaction requests local locks (t_l), runs (t_T), and the system checks limits (t_{cl}). Since there are no conflicts, limits do not have to change, so the locks are released (t_r) and the transaction commits, sending the answer back to the user (t_m). Therefore, the total CPU resources consumed by a transaction in this case are:

$$t_{dc}^1 = 2t_m + t_s + t_l + t_T + t_{cl} + t_r$$

For the same transaction running under two-phase commit, the timing is as follows. First a message is received (t_m), a context switch is done (t_s), the locks are requested (t_l), and then a message is sent to the other node (t_m) to request remote locks. At this point, this node switches to run another transaction while waiting for the answer to the lock request. (We assume an infinite supply of transactions to maximize throughput.) At the other node, the transaction produces a context switch (t_s), requests the remote locks (t_l), and a message is sent back acknowledging the locks

Table 2. Throughput values for Scenario 1

Protocol	Case 1	Case 2
T^1_{dc}	563 Trans/sec	16.6 Trans/sec
T^1_{2pc}	308 Trans/sec	13.3 Trans/sec

(t_m). When this message arrives, the first node can switch back to the original transaction (t_s), run it (t_T), release the locks (t_r), and commit sending a message to the other node (t_m) which, in turn, releases the locks (t_r). Finally, a message to the user is sent (t_m). The CPU resources consumed *at both nodes* by the transaction are:

$$t^1_{2pc} = 3t_s + 2t_l + 5t_m + t_T + 2t_r$$

In analyzing the throughput, we must notice that since we have two nodes, we have two units of CPU resources available in one second. The respective throughputs are:

$$T^1_{dc} = \frac{2}{t^1_{dc}} \text{ and } T^1_{2pc} = \frac{2}{t^1_{2pc}}$$

From the preceding equations, one can immediately establish that $T^1_{dc} > T^1_{2pc}$ if the following equation holds:

$$t_{cl} < t_l + 3t_m + t_r + 2t_s$$

That is, the demarcation protocol throughput will be greater than the one for two-phase commit if the cost of checking limits is less than the sum of the costs on the right hand side. We believe that, in general, this will be true, since checking limits is a simple operation whose cost should be comparable to that of locking an item. To see the gains for our two-parameter settings, we substituted the values in the respective equations. For the values shown in Table 1, the throughput values are shown in Table 2. As can be seen, the performance of the demarcation protocol is significantly better; 83% higher throughput in Case 1, 25% in Case 2. In Case 2, gains are smaller because the transaction processing cost t_T (equal for both protocols), is much higher.

Let us turn our attention to the second scenario and compare the response time for both protocols. Recall that in this scenario every transaction encounters a conflict when it is run for the first time. Under the demarcation protocol a request message is received (t_m), a context switch takes place (t_s), locks are requested (t_l), the transaction is run (t_T), and the limits checked (t_{cl}). Since the value goes beyond the limit, the transaction is aborted, the locks released (t_r), and a message is sent to the other node, requesting a change of limits (t_m). This message takes t_d time to arrive. When it arrives, there is a scheduling delay t_x, until the processor can

Table 3. Response times for Scenario 2

Protocol	Case 1	Case 2
t^2_{dc}	228	462
t^2_{2pc}	340	556

Times in milliseconds.

service the request. Then a context switch takes place (t_s), the limits are changed (t_{ch}), and a message acknowledging the changes is sent back (t_m). This message arrives at the first node t_d units later, and there is a scheduling delay (t_x). Finally, after a context switch (t_s), the limits are changed (t_{ch}), and the transaction can be resubmitted. Resubmitting the transaction takes again $t_s + t_l + t_T + t_{cl} + t_r$. Finally, a message is sent to the user with the answer t_m. Thus, the total response time for the transaction is:

$$t^2_{dc} = 4t_m + 4t_s + 2t_l + 2t_T + 2t_{cl} + 2t_r + 2t_d + 2t_x + 2t_{ch}$$

For two-phase commit, we can use part of the analysis of Case 1. To the value t^1_{2pc} we should add $2t_d$ (i.e., two message delays), and $2t_x$ (i.e., the scheduling delay at both nodes when the messages arrive). (We did not take this into account earlier because we were computing throughput, not response time.) This yields t_n, the response time with no conflicts. But, since we are assuming that the remote locks will be taken, the transaction will wait until the locks are released. We assume that the transaction will wait on the average of half of the total running time of the transaction holding the locks (i.e., half of t_n). Therefore, the total response time for the transaction is:

$$t^2_{2pc} = \frac{3}{2}(t^1_{2pc} + 2t_d + 2t_x)$$

Table 3 compares the response times for the two protocols and the two cases (values in milliseconds). Even in this conflict scenario, the demarcation protocol outperforms two-phase commit. In Case 1, the response time for the demarcation protocol is 33% better than the one for two-phase commit. The difference is less marked for setting 2, again since the dominating factor is the transaction length. Even in this case, we get non-trivial improvement (17%).

6. Generalizing the Protocol

In this section, we generalize the demarcation protocol to operate on an arbitrary constraint of the form $c_1A + c_2B \leq \delta$, where $c_1, c_2 \neq 0$. We also discuss how to extend it to more than two data items.

We start by generalizing the predicates SAFE, LIMIT_BEYOND and VALUE_BEYOND for the constraint Φ_j: $c_1A + c_2B \leq \delta$. As before, we use X to refer to either A

or B, and X_{l_j} for its limit. We use the notation c_{x_j} to refer to the corresponding constant in the constraint (either c_1 or c_2).

$$\Phi_j.\texttt{SAFE}(X,\sigma) = \textit{if } c_{xj}\sigma \leq 0 \textit{ then } \text{TRUE } \textit{otherwise } \text{FALSE}.$$

For instance, for the constraint $A \leq B + \delta$, if we wish to increase the limit B_l by $\sigma > 0$, the predicate SAFE would be TRUE.

$$\Phi_j.\texttt{LIMIT\_BEYOND}(X,\sigma) = \textit{if } (X_{l_j} + \sigma - X)c_{X_j} < 0 \\ \textit{then } \text{TRUE } \textit{otherwise } \text{FALSE}.$$

For instance, for the constraint $A \leq B + \delta$, the predicate is true if $A_l + \sigma < A$.

$$\Phi_j.\texttt{VALUE\_BEYOND}(X,\theta) = \textit{if } (X + \theta - X_{l_j})c_{X_j} > 0 \\ \textit{then } \text{TRUE } \textit{otherwise } \text{FALSE}.$$

For the constraint $A \leq B + \delta$, the predicate is true if $A + \theta > A_l$.

Procedure change_value (Section 4.2) is unchanged, except that it uses the new definition of VALUE_BEYOND given here. Procedure accept_change (Section 4.1) is simply generalized for an arbitrary constraint Φ_j (i.e., Y_l is replaced by Y_{l_j}.) Procedure change_limit must be modified as follows. In the original procedure (Section 4.1), when one node changed its limit by σ, the second one would perform a change of the same value later. For the general constraint, however, the sign of the second change depends on the constants c_1 and c_2. For example, for the constraint $A + B \leq \delta$, a positive change in A's limit must be followed by a negative change in B's limit. In the procedure below, we also incorporate *policy4* (Section 4.2).

```
$\Phi_j$.change_limit($X,\sigma$)
    if $\Phi_j$.SAFE($X,\sigma$) is FALSE then
        Send message to $N(Y)$, requesting it to perform
          $\Phi_j$.change_limit($Y, -sign(c_{1_j}) sign(c_{2_j})\sigma$)
    else if $\Phi_j$.LIMIT_BEYOND($X,\sigma$) is TRUE then
        { fire up $\Phi_j$.policy4($X,\sigma$);
        abort this change }
    else
        { $X_{l_j} \leftarrow X_{l_j} + \sigma$;
        send message to $N(Y)$, requesting it to perform
          $\Phi_j$.accept_change($Y, -sign(c_{1_j}) sign(c_{2_j})\sigma$) }.
```

For splitting the slack in the constraint $c_1A + c_2B \leq \delta$, $c_1,c_2 \neq 0$, we generalize Equation 1 as follows:

$$A_l = A + (\delta - c_1A - c_2B)\frac{k}{c_1} \quad B_l = \frac{B - (\delta - c_1A - c_2B)k}{c_2} \qquad (2)$$

Equation 1 can be derived easily from $c_1A + c_2B \leq \delta$, noticing that the slack is equal to $\delta - c_1A - c_2B$.

To conclude this section, let us consider constraints of more than two variables. First, observe that if only two nodes are involved, nothing is different. For example, if we have the constraint $A + B + C \leq \delta$, and the items A, B are stored in one node, while C is stored at another, then A and B can be treated as a single variable as far as the demarcation protocol is concerned. That is, it would be enough to have two limits: AB_l and C_l, and to follow the protocol.

If there are three or more nodes involved, then whatever node performs a safe operation must indicate what other node gets the amount. For unsafe operations, a node must select a particular node for its request to change limits. For instance, consider the inequality $A + B \leq C + \delta$. Say $N(A)$ wants to raise its limit (an unsafe operation). $N(A)$ has a choice: it may ask $N(B)$ to lower its limit, or it may ask $N(C)$ to increase its limit. Suppose that $N(C)$ is selected. If $N(C)$ does raise C_l, a message to perform accept_change is sent only to $N(A)$, and only it consumes the available slack. Now suppose that both $N(A)$ and $N(B)$ want to raise their limits concurrently, and send requests to $N(C)$. $N(C)$ should indicate to each the amount of their changes. For example, if $N(C)$ raises C_l by 10, it may indicate to $N(A)$ that A_l can be raised by 6, and to $N(B)$ that B_l can be raised by 4.

The generalizations we have illustrated here for more than two sites are straightforward. However, the generalization of some of the policies may not be. In particular, to generalize a split_slack policy (Section 4) we need to get three or more nodes that may be concurrently updating their variables to agree on a slack distribution. In this case, a centralized policy (where one site decides on slack distributions) may be easier to implement.

7. Other Constraints

In the introduction we argued that internode constraints tend to be simple. So far we have studied one class of simple constraints: arithmetic inequalities. In this section, we illustrate how the demarcation protocol and its policies can be used to manage other types of simple constraints. The key idea is to convert these other constraints into arithmetic inequalities.

7.1 Referential Integrity

Consider a referential constraint of the form

$$Exist(A,a) \Longrightarrow Exist(B,b) \tag{3}$$

where A,B are data objects, a,b are nodes, and $Exist(A,a)$ is a predicate that is TRUE if object A is stored in node a. $Exist(B,b)$ is TRUE if B is at node b.

To translate this constraint into an arithmetic one, we can define the following function

$$f(X, x) = \text{if } Exist(X, x) \text{ is TRUE } then\ 1\ otherwise\ 0 \tag{4}$$

where X stands for either A or B, and x corresponds to either node a or b. With this equation, the referential constraint becomes

$$f(A,a) \leq f(B,b) \tag{5}$$

This constraint can now be enforced via the demarcation protocol, as long as we interpret arithmetic an operation on $f(X, x)$ to be the appropriate create or delete operation. That is, changing $f(A,a)$ from 1 to 0 means that A is deleted at a; changing it from 0 to 1 means that A is created. We must also define policies that implement the correct semantics for this case. One possibility is to define the following policies. (Policies 2 and 3 are not necessary in this case.)

```
$\Phi_j$.policy1($f(X,x),\delta,T_{code}$)
    *** An invalid change to $f(X,x)$ has been attempted;
      first change limit and then try transaction later ***
    $\Phi_j$.change_limit($f(X, x),\delta$);
    resubmit later $T_{code}$.

$\Phi_j$.policy4($f(X,x),\delta$)
    *** An unsafe limit change has been attempted ***
    $\Phi_j$.change_value($f(X,x),\delta$, null)
    resubmit $\Phi_j$.change_limit($f(X,x)$, $\delta$).
```

To illustrate, assume that A and B are stored in nodes a and b, respectively. Thus, $f(A,a) = 1$, and $f(B,b) = 1$. Equation 5 must be enforced at all times. Therefore, the system establishes two limits $f(A,a)_l$ and $f(B,b)_l$, such that at all times $f(A,a) \leq f(A,a)_l$, $f(B,b) \leq f(B,b)_l$, and $f(A,a)_l \leq f(B,b)_l$. In this initial scenario, these limits can be $f(A,a)_l = f(B,b)_l = 1$. The safe operations are for a to decrease $f(A,a)_l$, and for b to increase to $f(B,b)_l$.

Now assume that there is a transaction T that wants to delete B at b. The transaction will try to change $f(B,b)$ by -1 to 0. However, since the limit $f(B,b)_l = 1$, the transaction will be aborted and *policy 1* will be fired. This policy will force the change of -1 on limit $f(B,b)_l$, invoking change_limit($f(B,b)$,-1). This is not a safe operation for b, so a message will be sent to a, requesting it to change $f(A,a)$ by -1 to 0. In turn, a will invoke change_limit($f(A,a)$,-1). Since the desired new limit $f(A,a)_l = 0$ violates the constraint $1 = f(A,a) \leq f(A,a)_l$, *policy4* will be triggered (and the limit change aborted). This policy will force the change in $f(A,a)$ by -1 to 0, deleting A from a. The policy also resubmits the change in the limit $f(A,a)_l$ to 0. This time, the change will be successful. As a consequence of the change, a message will sent to b, allowing it to change $f(B,b)_l = 0$. When T is resubmitted and attempts the deletion of B again, it will find that the new limit allows it, and will proceed to delete B. Thus, the existential constraint is obeyed at all times. Note that if T is resubmitted early, before $f(B,b)_l$ has had a chance to change, unnecessary (but not harmful) messages will be triggered. One way to avoid this is to have the change in the limit $f(B,b)_l$ trigger the resubmission of the pending transaction.

While this may not be the most efficient way to enforce existential constraints, we believe it is very useful to have a uniform strategy for handling a significant fraction of all distributed constraints. Also, notice that we have assumed no failures in this process. Since the protocol does not guarantee failure atomicity, it is possible for transaction T to abort (because of a crash, for example) and never delete B on site b, after the deletion of A on site a had been completed. This, however, does not violate the constraint.

7.2 Key Constraints

Consider a relation R partitioned over sites $S_1, ..., S_s$. Assume attribute K, over domain $D = \{ v_1, ..., v_n \}$, is a key for R. This key constraint indicates that if a tuple with K value v_i exists at site S_j, then that K value cannot exist for any other tuple, at any site. Enforcing the constraint within each site (i.e., ensuring that locally there are no duplicate keys) is simple. Here, we focus on enforcing it across sites.

To show how the demarcation protocol can be used in this case, we define the function:

```
*if* value $v_i$ exists at site $S_j$ *then*
    $e(i,j) = 1$
    else $e(i,j) = 0$.
```

The key constraint is then equivalent to the following set of arithmetic constraints:

$$e(1,1) + e(1,2) + ... + e(1, s) \leq 1$$

$$...$$

$$e(n,1) + e(n,2) + ... + e(n,s) \leq 1$$

These constraints can be enforced by the demarcation protocol. Note that each site will have n limits, one for each possible key value. If one of the limits at S_j, say the one for value $v_i = 500$, is set to 1, then S_j is free to add this key value to R without consulting the other sites. If, on the other hand, the limit for v_i is 0, and site S_j wishes to insert this key value, then it must first negotiate for a limit change. If some other site has its $v_i = 500$ limit set to 1 and does *not* have that value, then it can accede to the request. Otherwise, it does not and site S_j cannot insert the 500 value. Initially, the limits can be set to 1 at those sites that are likely to want to insert those values.

Of course, keeping track of limits on a per domain value basis requires a lot of bookkeeping. To make this practical we must somehow encode the limits. One possibility is to use a bit vector to represent the limits. That is, a 1 in position i of the limit vector at site S_j indicates that the limit for $e(i,j)$ is one.

If the elements of domain D are ordered, then another encoding scheme is possible. Each site S_j is assigned two bounds, $b_{j,l}$ and $b_{j,u}$. The interpretation is that for all domain values v_i between the bounds ($b_{j,l} \leq v_i \leq b_{j,u}$) the corresponding limit is set to 1. For this to work, the ranges at different sites must not overlap. That is, for all other sites k different than j, either $b_{k,u} < b_{j,l}$ or $b_{j,u} < b_{k,l}$.

This technique is actually used in practice. For example, if a company hires employees at two sites, then the first site may be assigned employee numbers 1 through 1,000 and the second numbers 1,001 through 2,000. The first site can then insert new employees with numbers between 0 and 1,000. If it runs out of numbers, it can ask the second site to "slide" its bounds. (Of course, for this to be practical the second site has to start assigning numbers in a decreasing order, starting from 2,000.) As a variation, we may wish to allow sites to have multiple windows, so that the first site can request numbers 3,001 through 4,000 (from a master site that holds the rest of the key) after it inserts its first 1,000 employees.

As with existential constraints, the enforcement of key constraints needs to be optimized beyond what the basic demarcation protocol provides (e.g., by managing limits in groups bounded by $b_{j,l}$ and $b_{j,u}$.) Our main point here is that key constraints can also be managed within the general framework that the demarcation protocol provides.

7.3 Copy Constraints

As stated in Section 1, approximate equality constraints of the form $|A - B| \leq \varepsilon$ can be implemented via the two constraints $A - B \leq \varepsilon$ and $A - B \leq \varepsilon$. Each constraint can then be enforced by the demarcation protocol.

In this case, the policies for changing limits for the two constraints should be coordinated. To illustrate, let us consider an example, $|A - B| \leq 5$, where A is stored at $N(A)$ and B is stored at $N(B)$. For the first constraint, $B - A \leq 5$, we have limit A_1 at $N(A)$ and B_1 at $N(B)$. The conditions that must be satisfied are $A_1 \leq A$, $B \leq B_1$, and $B_1 - A_1 \leq 5$. Similarly, for the constraint $A - B \leq 5$, we have $A_2 \geq A$ (at $N(A)$), $B \geq B_2$ (at $N(B)$), and $A_2 - B_2 \leq 5$.

Notice that at each site there is a window of allowable values. At $N(A)$, the value for A must be in the range $A_1 \leq A \leq A_2$, while at $N(B)$ the range is $B_2 \leq B \leq B_1$. Let us assume that initially $A = B = 10$, and $N(A)$ is given all the slack (i.e., $A_1 = 5$, $A_2 = 15$, and $B_1 = B_2 = 10$. Figure 3 illustrates this scenario. In this case, $N(A)$ is free to vary A in the 5–15 window, while $N(B)$ cannot change B at all. This may make sense, for instance, if $N(A)$ is the stock market, and $N(B)$ is a site that simply tracks the stock values. As the price of a particular stock A fluctuates in the window 5–15, its remote copy, B, does not need to be updated.

Next assume that A increases to 14, triggering policy 3 (Section 4). Say $N(A)$ then requests an increment of 4 units in A_2 (unsafe), so a message is sent to $N(B)$, requesting an increment of 4 in B_2. At site $N(B)$, it will be impossible to comply because the new value of B_2 (14) is larger than B. (Remember that the constraint $B_2 \leq B \leq B_1$ must be satisfied at $N(B)$. The solution, of course, is to update B to 14, since it is supposed to be a copy of A. Thus, it makes sense that *policy2* should also trigger an update to B.

Figure 3. Initial scenario

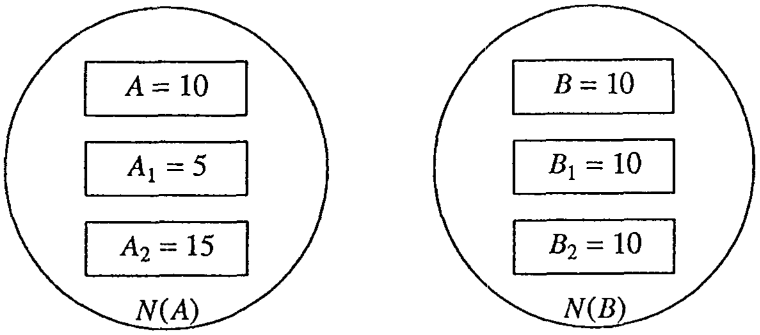
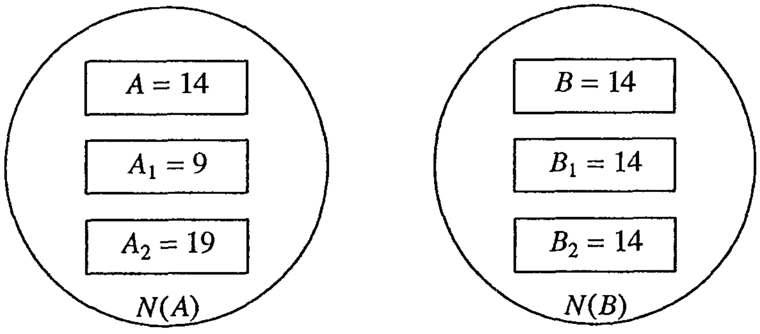


Figure 4. Scenario after increments



However, site $N(B)$ cannot update B to 14 because of the B_1 constraint. Thus, before B is updated, $N(B)$ must request a limit change of B_1 to 14. Since this is an unsafe operation, a change in A_1 must first be requested. Of course, this extra round of negotiation could be avoided if $N(A)$ had changed A_1 in the beginning.

In summary, here is how the change in limits should proceed when A is updated to 14 (*policy2*). Site $N(A)$ should unilaterally increase A_1 by 4 (to 14), and send a message to $N(B)$ to increase B_1 by 4; update B to 14; and increase B_2 by 4. When $N(B)$ replies, site $N(A)$ can increment A_2 by 4. Figure 4 shows the resulting state. Thus, the policies for both constraints need to be coordinated. Instead of moving limits individually, they should be moved as pairs (i.e., the window defined by A_1 and A_2 should be slid.)

Note that the demarcation protocol may also handle a copy constraint based on versions (Section 1). The basic idea is to associate with each object a version number or counter that is incremented at each time period. Constraints (e.g., "the copy of O at a site must be within two versions of the master copy") can be translated into arithmetic inequalities that can be managed via the demarcation protocol.

A limitation of the demarcation protocol for managing approximate copies is that large changes lead to repeated negotiations. For instance, returning to our $|A - B| \leq 5$ example, say that initially $A = B = 10$, and we want to change A to 100. The only way this can be achieved without violating the constraint is to change A to 15, then change B to 15 and adjust the limits, then move B to 20, 25, 30, and so on. In this case it may be preferable to use a blocking strategy: First lock A and B, then change A and B to 100 (violating the constraint momentarily), and then unlock A and B.

Another solution is offered by quasi-copies (Alonso et al., 1990), as discussed in Section 2. With this method, only one of the copies can be modified; the other simply tracks the master copy. Furthermore, the system is given a time window in which to enforce constraints. Thus, the constraint "copies A and B should be within 5 units of each other" is really the constraint

1. Either A and B are within 5 units;
2. the value B is a copy of A that existed at most T time units ago;
3. site $N(B)$ has failed.

Thus, if the value of master copy A fluctuates close to 10 (within 5 units; assuming $B = 10$), no update to B is necessary. If A changes to 100, the system has T time units to propagate the update. If $N(B)$ fails, then A can be changed without notifying B.

In summary, quasi-copies provide a more flexible type of constraint that may be temporarily violated. The demarcation protocol, on the other hand, provides an absolute guarantee that the constraint will be satisfied. It also allows all participants (not just a master site) to modify the variables involved.

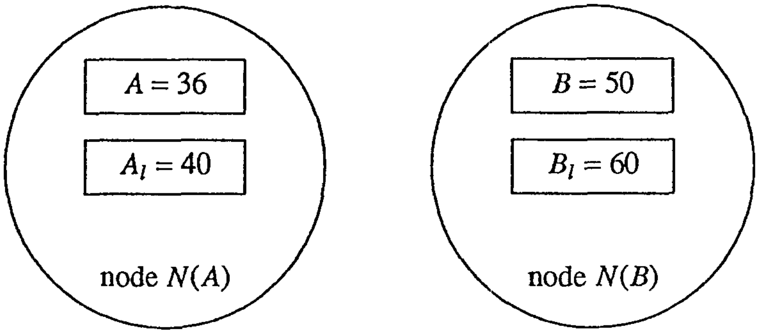
8. Serializability and the Demarcation Protocol

In Section 1 we stated that the demarcation protocol does *not* guarantee global serializable schedules, whether conflict or view serializable. It is clear that the demarcation protocol gives transactions much more flexibility than conventional guaranteeing protocols. At the same time, in the examples presented so far, one does not see any of the anomalies associated with nonserializable schedules (i.e., all final database states that have been shown could have been obtained by some serial schedule.) So, does the demarcation protocol guarantee serializability after all?

For our discussion, let use the constraint $A + B \leq 100$, with A located at site $N(A)$ and B at $N(B)$. Let us assume that limits have been set at each site. Let us look first at an execution that involves *no* limit changes. In this case, it is easy to see that executions are indeed serializable. Transactions at $N(A)$ will only read and write data (A and A_l) locally; the ones at $N(B)$ will only access local data. Hence, there are no conflicts, and schedules are serializable.

The key to achieving serializability without distributed locking (or the equivalent) is to "partition" the original constraint into the constraints $A \leq A_l$, $B \leq B_l$ and A_l

Figure 5. Initial scenario before limit changes



$+ B_l \leq 100$. With the original constraint, a transaction that wanted to modify A had to access data at both sites; now it only has to access data at $N(A)$.

Of course, changing the limits involves actions at both sites and this is where we can lose serializability. Consider the following example. The limits are set to $A_l = 40$ and $B_l = 60$. An update at $N(A)$ brings A to 36, triggering *policy2* because A is now too close to its limit. Call this transaction T_1; it will first ask for a decrement to the B_l limit (at $N(B)$), and then it will add the same amount to A_l at $N(A)$. At the same time, an update brings B to 50, triggering another instance of *policy2.* Call this second transaction T_2. The situation at this time is illustrated in Figure 5.

When T_1 arrives at $N(B)$, requesting a limit decrease, $N(B)$ decides to decrease by *half* of its slack. That is, the local slack is $B_l - B = 60 - 50 = 10$, so $N(B)$ decrements B_l by 5. A message is sent to $N(A)$, instructing it to increment its A_l limit by 5. Concurrently, T_2 is requesting a limit change at $N(A)$, where the same "give half of the slack" policy is in use. The slack is $A_l - A = 40 - 36 = 4$, so A_l is decremented by 2. A message is sent to $N(B)$ to increment B_l by 2. The resulting final state is $A_l = 38 + 5 = 43$, and $B_l = 55 + 2 = 57$.

This state cannot be obtained by a serial execution of T_1 and T_2. If T_1 runs first, $N(B)$ yields 5 points and the limits are $A_l = 45$, $B_l = 55$. If T_2 then follows, $N(A)$ yields half of its current slack (half of 9), resulting in $A_l = 40.5$, $B_l = 59.5$. If T_1 and T_2 run in reverse serial order, the result is $A_l = 44$, $B_l = 56$. Hence, the execution of the two concurrent limit changes is not serializable.

What makes the changes nonserializable is the "give half of the slack" policy, which does no commute with other changes. If we had used a commutative policy (e.g., "always give 1 point"), then the resulting schedule would be (view) serializable (Bernstein et al., 1987). But in general, limit negotiations can lead to nonserializable schedules.

Finally, we point out that nonserializable schedules can also arise if user transactions (as opposed to limit negotiation transactions) access data at more than one site. For example, in our same example, say T_1 reads A, commits at $N(A)$, then updates B, and commits at $N(B)$. Concurrently, assume T_2 reads B, commits at $N(B)$, then updates A and commits at $N(A)$. At $N(A)$ we have dependency $T_1 \rightarrow T_2$, while at $N(B)$ we have $T_2 \rightarrow T_1$ (i.e., a nonserializable schedule). Note that in spite of this, the constraint $A + B < 100$ continues to hold. If multi-site transactions such as T_1 and T_2 are only concerned about the demarcation constraints, then they can operate at nodes freely, without two-phase commit. On the other hand, if global serializability is required (e.g., when accessing objects whose constraints are unknown), then multi-site transactions must follow a two-phase commit protocol or the equivalent (Breitbart et al., 1992).

9. Conclusions

We have presented a strategy for enforcing linear arithmetic inequalities in distributed databases. Limits are defined for each of the participating variables; a node is free to update a variable as long as it stays within its bounds. The demarcation protocol is used to change the limits in a dynamic fashion. We also showed how this strategy can be used for other types of constraints, such as referential constraints and approximate equalities.

Intuitively, we may view a system that uses the demarcation protocol as a "spring." When a constraint has a lot of slack, limits will not be tight, and many transactions will be able to perform their updates locally. This corresponds to a loose spring (the first scenario considered in Section 5). As the slack is reduced, the spring is compressed, and more and more transactions will hit against the limit. The transactions will be more expensive to run, as they will require negotiations with the other node to change limits. When there is no slack (e.g., $A = A_l$ and $B = B_l$), the spring is compressed the most. Even at this point, the system may be more efficient than a conventional one (which requires two-phase commit for every transaction), because transactions that move a variable away from its limit may be done locally.

The demarcation protocol is limited because it only applies to linear arithmetic constraints. However, we have argued that a very large number of distributed constraints fall into this simple category. As a matter of fact, we have difficulty envisioning more complex constraints that would arise in a practical distributed system.

There are two ways to implement the demarcation protocol in a database system. One is to use an existing system, and to provide the user with a library of procedures (e.g., `change_value`), sample policies (e.g., for splitting slack), and definitions (e.g., for limit variables). The user's code could then call these procedures to update values or change limits. The disadvantage of this approach is that a user could circumvent the rules (e.g., by updating a constraint variable directly and not through `change_value`). The other option is to incorporate the procedures into the system itself. The database administrator (or possibly an authorized user) would define the constraints and policies and give them to the system. The variables involved would be tagged. When a transaction updated one of these variables, it would trigger the necessary procedures. With either one of these approaches, there are clearly many issues that still need to be resolved, such as language used for defining the constraints, load-control strategies, and dealing with contradictory constraints.

Acknowledgments

We would like to thank Luis Cova and Ken Salem for helpful discussions, and Naftaly Minsky and Patrick O'Neil for pointing out some useful references. We would also like to thank the referees for many useful suggestions.

References

Alonso, R., Barbará, D., and Garcia-Molina, H. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems,* 15(3):359-384, 1990.

Breitbart, Y., Garcia-Molina, H., and Silberschatz, A. Overview of multidatabase transaction management. *VLDB Journal,* 2(2):181-239, 1992.

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

Carvalho, O.S.F. and Roucariol, G. On the distribution of an assertion. *Proceedings of the ACM-SIGOPS Symposium on Principles of Distributed Computing,* Ottawa, Canada, 1982.

Date, C.J. *An Introduction to Database Systems.* Reading, MA: Addison-Wesley, 1983.

Davidson, S.B. An optimistic protocol for partitioned distributed database sytems. Ph.D. Dissertation, Princeton University. October, 1982.

Du, W. and Elmagarmid, A. Quasi-serializability: A correctness criterion for global concurrency control in InterBase. *Proceedings of the Fifteenth International Conference on Very Large Data Bases,* Amsterdam, 1989.

Fernández, M.F. and Zdonik, S.B. Transaction groups: A model for controlling cooperative work. *Proceedings of the Third International Workshop on Persistent Object Systems.* Queensland, Australia, 1989.

Fischer, J.M., Griffeth, N.D., and Lynch, N.A. Global states of a distributed system. *IEEE transactions on Software Engineering,* 8(3):198-202, 1982.

Garcia-Molina, H. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems,* 8(2):186-213, 1983.

Hammer, M.M. and Shipman, D.W. The reliability mechanisms of SDD-1: A system for distributed databases. Computer Corporation of America Technical Report CCA-80-04, 1980.

Korth, H.F. and Speegle, G.D. Formal model of correctness without serializability. *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Chicago, IL, 1988.

Krishnakumar, N. and Bernstein, A.J. High throughput escrow algorithms for replicated databases. *Proceedings of the Eighteenth International Conference on Very Large Data Bases,* Vancouver, BC, 1992.

Kumar, A. and Stonebraker, M. Semantics-based transaction management techniques for replicated data. *Proceedings of the ACM SIGMOD Conference on Management of Data,* Chicago, IL, 1988.

Lynch, N.A., Blaustein, B., and Siegel, M. Correctness conditions for highly available replicated data. *Proceedings of the Fifth Annual ACM Symposium on the Principles of Distributed Systems,* Calgary, Canada, 1986.

O'Neil, P. The escrow transactional method. *ACM Transactions on Database Systems,* 11(4):405-430, 1986.

Pu, C. and Leff, A. Replica control in distributed systems: An asynchronous approach. *Proceedings of the ACM SIGMOD International Conference on the Management of Data,* Denver, CO, 1991.

Soparkar, N. and Silberschatz, A. Data-value partitioning and virtual messages. *Proceedings of the Conference on the Principles of Database Systems,* Nashville, TN, 1990.