Improving Performance in Distributed Database Systems Using Speculative Transaction Processing

P. Krishna Reddy and Masaru Kitsuregawa
Institute of Industrial Science
The University of Tokyo
7-22-1, Roppongi, Minato-ku
Tokyo 106, Japan
{reddy, kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract

In distributed database systems (DDBSs), a transaction acquires the locks on the data objects during the execution and releases them only after the completion of commit processing. In DDBSs, it can be observed that, even though a transaction produces new data object values on completion of execution and these new data object values are not affected during commit processing, the locks are released only after completion of commit processing. In this paper, we propose speculative distributed transaction processing (SDTP) approach, which allows the transaction to release locks on the data objects on completion of its execution. On completion of transaction's execution, by accessing both original and updated values, the waiting transaction carries out speculative executions and then starts commit processing. Before the end of commit processing, the transaction that has carried out speculative executions retains appropriate execution based on the termination decisions of preceding transactions. Thus, conflicting transactions can be processed in parallel without violating the serializability criteria. In this paper, we have explained the transaction processing strategy employing speculation and then presented SDTP algorithm. This approach is free from cascading aborts. Also, SDTP approach does not require extra number of messages since every message is piggy backed with the messages of commit processing. However, it needs both extra processing power and main memory to support speculative executions. This approach can be tuned according to the resources available in the system. Through simulation experiments, it has been shown that SDTP approach considerably reduces the response time in the case of higher resource conflicts and longer transmission times. This approach suits best for the wide area network environments where the message transmission time dominates the processing time.

Keywords transaction processing, concurrency control, distributed databases, distributed algorithms, non-blocking protocols.

1 Introduction

In distributed database systems (DDBSs), the processing of a transaction can be separated into two stages: the execution stage and the commit stage. During the execution stage, a transaction accesses the data values from various sites and updates these data values in a temporary storage of arrival site. During the commit stage, these updated data values are written in the stable storage at participant sites in an atomic manner. The three popular concurrency control methods are locking [1, 2], timestamping [3], and optimistic algorithms [4], which have been proposed for both centralized and distributed environments. In order to commit a transaction, logging [2], shadow paging [5], and differential files [6] techniques are proposed for centralized database systems, whereas two-phase commit (2PC) protocol [2] and three-phase commit (3PC) protocol [7] are proposed for DDBSs. The general aim of a transaction processing (concurrency control and commit) approach is to ensure the consistency of the database and the correct completion of each transaction initiated in the system. An obvious additional requirement is to improve the performance of a database system by minimizing the average response time. In order to minimize the response time, it is necessary to maximize parallelism of transactions' processing. The major factor in parallelism is the period of time for which the data object values become unavailable to waiting transactions during commit processing of transactions. In DDBSs, especially in wide area network (WAN) environments, the time required to commit a transaction has a significant share as compared to its execution. This is true because, 2PC/3PC protocol requires exchange of communication messages as well as disk accesses at local sites which are involved in computation. During this process, a transaction has to wait for responses from participating sites, which requires substantial amount of waiting time and processing time. Thus, the time spent by a transaction during its commit processing becomes a major factor in degrading the performance of DDBSs.

In DDBSs, it can be observed that, even though a transaction produces the new data values at the end of its execution stage and these new data values are not affected during 2PC/3PC processing of a transaction, the locks on the data objects are released only after the completion of 2PC/3PC processing. In this situation, parallelism can be increased by allowing subsequent transactions to access the final data values produced by the current transaction, immediately at the end of its execution stage. In this paper, we propose speculative distributed transaction processing (SDTP) approach, which allows the transaction to release the locks on the data objects on completion of its execution. By accessing both original values and updated values on completion of transaction's execution, the waiting transaction speculatively carries out alternative executions, starts commit processing. Before the end of commit processing, the transaction that has carried out speculative executions retains appropriate execution based on the termination (commit/abort) decisions of preceding transactions.

Because of its critical performance issue, the transaction processing problem has been well studied in the context of both centralized and DDBSs. There are proposals in the literature to increase the performance by employing hybrid concurrency control algorithms, multi-version algorithms, and semantic based algorithms [8]. Integration of concurrency control protocols and commit mechanisms for centralized environments is explored in reference [9]. Recently commit processing has attracted attention due to its effect on the performance on the transaction processing. It has been shown in [10] that the time to commit accounts for one third of the transaction duration in a general purpose database. In [11], experimental studies have been reported on behavior of concurrency control and commit algorithms in the WAN environments. It has been shown that the time to commit can be as high as 80 per cent of the transaction time in the WAN (Internet) environments. A concurrency control approach for real time environment by employing redundant computations is proposed in [12]. In this approach, whenever a transaction encounters a conflict during its execution, it starts another shadow execution. In this way, a transaction may have k number of speculative executions. By this, the probability of completing the execution of a transaction before the deadline increases. In [13] the speculation is applied to service requests made by clients on the World Wide Web. In this, the server responds to a client's request by sending, in addition to the data (documents) requested, a number of other documents that it speculates will be requested by that client in near future. It has been demonstrated that appropriate speculation reduces both server load and service time, without increasing network bandwidth. In [14], using simulation model, it has been shown that distributed commit processing can have more influence than distributed data processing on the throughput performance. In this, a modified optimistic protocol is proposed, in which a transaction carries out single execution by reading the uncommitted values produced by the conflicting transaction, up to one level. A failure of a transaction may result in aborting of another transaction which has read uncommitted values of the failed transaction. Modified optimistic protocol is proposed for local area network based environments in which the probability of failure during the commit processing of a transaction is less as compared to the WAN environments.

In SDTP approach, a transaction releases locks on completion of its execution. In this paper, we have proposed SDTP approach for a generalized case. That is, a transaction which has carried out speculative executions in turn releases the locks on the data objects immediately after its execution. As such, there is no limitation on the number of levels but depends on the system's resources, such as the size of main memory. Also, SDTP approach is robust with the aborts of transactions. That is, a transaction abort does not lead the other transactions which have carried out speculative executions to abort. Using SDTP approach, conflicting transactions can be processed in parallel without violating the serializability criteria. SDTP approach best suits for the WAN environments where the message transmission time dominates the processing time.

The body of the paper is organized as follows. In the next section, we explain 2PC protocol and define a system model. In section 3, we explain the basic idea of speculative transaction processing without involving commit processing. In section 4, we explain SDTP approach. In section 5, we present SDTP algorithm and present a method to tune SDTP approach according to the available resources in the system. In section 6, we present the results of the performance evaluation for both unlimited as well as limited resources environments. The last section finally consists of summary and conclusions.

2 Two-phase commit protocol and system model

2.1 Two-phase commit protocol

In DDBSs, 2PC protocol extends the affects of local atomic commit actions to distributed transactions by insisting that all sites involved in the execution of a distributed transaction agree to commit the transaction before its effects are made permanent. A brief description of 2PC protocol that does not consider failures is as follows. Initially, the coordinator (originating site of a transaction) writes a **begin-commit** record in the log, sends $PREPARE$ messages to all participating sites, and enters the *wait* state. When a participant receives $PREPARE$ message, it checks if it can commit the transaction. If so, the participant writes a **ready** record in the log, sends $VOTE\_COMMIT$ message to the coordinator, and enters the *ready* state.

Otherwise, the participant writes an **abort** record and sends a $VOTE\_ABORT$ message to the coordinator. If the decision of the site is to abort, it can forget about that transaction. The coordinator aborts the transaction globally, even if the coordinator receives $VOTE\_ABORT$ message from one participant. Then, it writes an **abort** record, sends a $GLOBAL\_ABORT$ message to all participant sites, and enters the *abort* state; Otherwise, it writes a **commit** record, sends a $GLOBAL\_COMMIT$ message to all participants, and enters the *commit* state. The participants either commit or abort the transaction according to the coordinators' instructions and sends back ACK (acknowledgment) message at which point the coordinator terminates the transaction by writing an **end-of-transaction** record in the log.

The 2PC protocol ensures the consistent termination of a transaction even when failures occur [17, 18]. Also, different communication paradigms can be used to implement 2PC protocol. The one described above is called the centralized 2PC, since the communication is between the coordinator and the participants only. In this process, the participants do not communicate among themselves.

2.2 System model

The DDBS consists of a set of data objects. A data object is the smallest accessible unit of data. In this paper, $X, Y, \ldots, Z$ represent data objects. Each data object is stored at one site only. Transactions are represented by $T_i, T_j, \ldots$; and sites are represented by $S_i, S_j, \ldots$; where, $i, j, \ldots$ are integer values. The data objects are stored at database sites connected by a computer network.

Each site supports a transaction manager and a data manager. The transaction manager supervises the processing of the transactions, while the data managers manage individual databases. The network is assumed to detect failures, as and when these occur. When a site fails, it simply stops running and the other sites detect this fact. The communication medium is assumed to provide the facility of message transfer between sites. When a site has to send a message to some other site, it hands over the message to the communication medium, which delivers it to the destination site in finite time. We assume that, for any pair of sites S_i and S_j, the communication medium always delivers the messages to S_j in the same order in which they were handed to the medium by S_i.

We define a transaction as a set of atomic operations on data objects. An operation is either read (returns the value of the data object), or a write (updates the object with a specified new value). For any T_i and data object X, $r_i[X]$ denotes a read executed by T_i on X. Similarly, $w_i[X]$ denotes a write executed by T_i on X. In general, a transaction does not have to be a totally ordered sequence. When two operations are not ordered relative to each other, these can be executed in any order. However, a read and a write on the same element must be ordered. A transaction that has been initiated but has not yet committed or aborted is said to be active. The objects to be locked by the transaction for the purpose of read and write steps are termed as read-set (RS) and write-set (WS), respectively. The union of RS and WS of T_i constitute locking variables. Transactions T_i and T_j are said to have R(read)-W(write), W-R or W-W conflict, if $RS(T_i) \cap WS(T_j) \neq \phi$; $WS(T_i) \cap RS(T_j) \neq \phi$; or $WS(T_i) \cap WS(T_j) \neq \phi$, respectively. Also, T_i and T_j are said to be in conflict, if at least one of the above conflicts exists.

It is assumed that two-phase locking algorithm is employed for concurrency control and 2PC protocol with centralized communication paradigm is employed for commit processing. Also, deadlock detection and removal [15, 16] is carried out whenever a deadlock occurs in the system.

Figure 1: Processing of a T_i in the DDBSs

The processing of a T_i, in DDBSs is depicted in Figure 1. In this Figure, s_i denotes the starting of execution and e_i denotes the completion of execution. The data values produced at e_i are termed as uncommitted (new) values. The completion of first and second phases of 2PC protocol are denoted by f_i and c_i, respectively.

In SDTP approach, a transaction starts its processing in the normal mode. During its execution, if a transaction reads at least one uncommitted value produced by another transaction, it enters into speculative mode. In speculative mode, a number of speculative executions are carried out on behalf of T_i. We use the notation T_{ij} or (i,j) to represent j^{th} $(j \geq 1)$ execution of T_i. The notation T_{i1} is used to represent the initial execution of T_i. Also, RS and WS of T_{ij} are represented by RS_{ij} and WS_{ij} respectively. A data object may have more than one uncommitted versions. The notation $X_q (q \geq 1)$ is used to represent q^{th} version of X. The notation X_1 represents initial value of X when no transaction is accessing it.

3 Speculative transaction processing

In this section, we first explain about the version tree of the data object. Next, we explain speculative transaction processing without involving commit processing. Subsequently, we present the execution complexity of speculative transaction processing.

3.1 Version tree of the data object

We employ tree data structure to organize the uncommitted versions of a data object produced in speculative transaction processing. For a data object X, its version tree is denoted by $tree_X$. It is a tree with X_1 (committed version) as the root and its uncommitted versions as the rest of nodes. Initially, X is represented as a tree with X_1 as the root node. A transaction accesses $tree_X$ in place of X. Also, $tree_X$ is updated when a transaction that accesses X completes execution.

Consider that T_i carries out n ($T_{ik}, k = 1 \ldots n$) number of executions, by accessing the set of objects, say $objects_i$. Let $X \in objects_i$. On completion of $T_i's$ execution, trees of data objects that belong to $objects_i$ are updated as follows. For each $X \in objects_i$, each new version of $X \in WS_{ik}$, $k = 1 \ldots n$, is included as a child to the corresponding read version node of $tree_X$.

3.2 Speculative transaction processing approach

We first explain speculative transaction processing approach by considering a simple case where multiple transactions access single data object. Next, we extend to a generalized case where a transaction accesses multiple data objects.

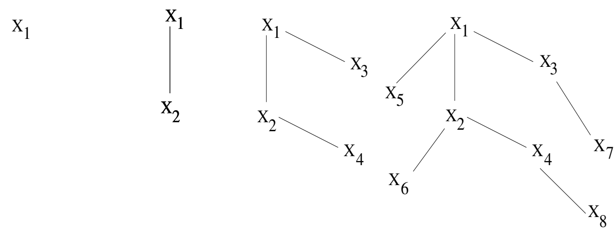
3.2.1 Simple case

Consider that X is stored at S_1 and both T_1 and T_2 arrive simultaneously into the system to update it. In order to explain the approach it is assumed that a long time is required to commit a transaction as compared to its execution. Using speculation, transactions are processed as follows.

We employ nested parentheses notation to represent $tree_X$. Initially, $tree_X$ is (X_1). Consider that T_1 starts first execution T_{11}, with $RS_{11} = \{X_1\}$. T_{11} completes with $WS_{11} = \{X_2\}$. X_2 is then included in $tree_X$ as $(X_1(X_2))$ as shown in Figure 2(b). Afterwards, T_1 proceeds with its commit processing. The proposed approach allows T_2 to access $tree_X$. T_2 carries out two speculative executions: one is T_{21} with $RS_{21} = \{X_1\}$ and other is T_{22}, with $RS_{22} = \{X_2\}$. T_{21} completes with $WS_{21} = \{X_3\}$, and T_{22} completes with $WS_{22} = \{X_4\}$. Afterwards, both X_3 and X_4 are included in $tree_X$ as $(X_1(X_2(X_4), X_3))$ as shown in Figure 2(c). If T_1 terminates with successful commit, T_{22} is retained. Otherwise, T_{21} is retained.

Now consider that T_3 enters into the system to access X. By accessing $tree_X$, T_3 carries out four speculative executions. These are T_{31}, T_{32}, T_{33}, and T_{34} with $RS_{31} = \{X_1\}$, $RS_{32} = \{X_2\}$, $RS_{33} = \{X_3\}$, and $RS_{34} = \{X_4\}$, respectively. T_3 completes the speculative executions, with $WS_{31} = \{X_5\}$, $WS_{32} = \{X_6\}$, $WS_{33} = \{X_7\}$, and $WS_{34} = \{X_8\}$, respectively. When, T_3 completes the execution, $tree_X$ is updated by including new versions X_4, X_5, X_6, and X_7 as $(X_1(X_2(X_4(X_8), X_6), (X_3(X_7)), X_5))$ as shown in Figure 2(d). Before the end of its commit processing, T_3 retains appropriate execution based on the commit/abort decisions of both T_1 and T_2.

When a transaction terminates, the trees of corresponding data objects are updated. For example, if T_1 commits, $tree_X$ is updated with X_2 as the root. Otherwise, if T_1 aborts, subtree of X_2 (including the node X_2) is deleted from $tree_X$.



(a)Initial state (b) After T_1's execution (c) After T_2's execution (d) After T_3's execution

Figure 2: State of $tree_X$

3.2.2 Generalized case

Now, we consider a generalized case where a transaction accesses multiple data objects. Before accessing any object, a transaction may be executing either in the normal mode or in the speculative mode.

A normal mode transaction has only one execution. When it accesses data object X having n nodes in its $tree_X$, it carries out n speculative executions and enters into speculative mode. Consider that T_i is executing in speculative mode with m executions. Next, it accesses X. There exist two cases: $tree_X$ contains only one node and $tree_X$ contains n nodes. If $tree_X$ contains only one node, each one of $T_i's$ speculative executions accesses $tree_X$ and continues its execution. Otherwise, if $tree_X$ contains n nodes, the process is extended to all the nodes of $tree_X$. That is, for each T_{iq} (q=1 ...m), n speculative executions are carried out (one for each node of $tree_X$).

3.3 Complexity of speculative transaction processing

In SDTP approach, the number of speculative executions produced by T_i depends on the number of objects it accessed and the number of uncommitted versions in the tree of each object.

Theorem : Consider that T_i has completed execution by accessing m data objects. Let v_k be the number of uncommitted versions in the tree of k^{th} $(1 \leq k \leq m)$ data object and N_i be the total number of executions produced by T_i. Then, $N_i = \prod_{k=1}^{m} v_k$.

Proof: T_i starts execution by accessing the first data object having v_1 nodes in its tree and carries out v_1 executions. When it accesses the second object having v_2 nodes in its tree, each one of the v_1 executions carries out v_2 executions. Following this, after accessing all m objects, $N_i = \prod_{k=1}^{m} v_k$.

4 Speculative distributed transaction processing

In this section, we first explain the lock compatibility matrix. Next, through example we explain the processing of transactions employing SDTP approach.

4.1 Lock compatibility matrix

In Figure 3, *yes* indicates that corresponding lock requests are compatible and *no* indicates that corresponding lock requests are not compatible.The S and X denotes shared (read) and exclusive (read-write) locks respectively. Lock compatibility matrix in locking based approach is shown in Figure 3(a).

In SDTP approach, S lock is partitioned into execution-shared (ES) lock and commit-shared (CS) lock. Also, X lock is partitioned into execution-exclusive (EX) lock and commit-exclusive (CX) lock. The transactions request only ES and EX locks. On completion of a transaction's execution, after including the new data values in the respective trees of data objects, ES/EX locks are changed to CS/CX locks. Lock compatibility matrix for SDTP approach is shown in Figure 3(b).

Lock requested by T_j	**Lock held by T_i**	
	S	**X**
S	yes	no
X	no	no

(a)

Lock requested by T_j	**Lock held by T_i**			
	ES	**CS**	**EX**	**CX**
ES	yes	yes	no	yes
EX	no	yes	no	yes

(b)

Figure 3. Lock compatibility matrix (a) Locking (b) SDTP approach

In SDTP approach, multiple transactions can have CS/CX locks on a data object at the same time. To ensure consistency, SDTP approach follows the below rules.

1. If T_i obtains EX lock, while T_j is holding either CS lock or CX lock on the data object, T_i is committed only after the termination of T_j.
2. If T_i obtains ES lock, while T_j is holding CX lock on the data object, T_i is committed only after the termination of T_j.

4.2 Example using SDTP approach

We explain the processing of transactions in DDBSs by considering the example. Consider that the data objects X, Y, and Z are stored at sites S_1, S_2, and S_3 respectively. Both transactions T_1 and T_2 arrive at S_2 to (read and write) access (read and write) X,Y and Y, Z respectively. The processing without SDTP approach is depicted in Figure 4(a). T_1 completes the execution by producing the uncommitted data values of both X and Y. After the completion of its 2PC processing, T_1 releases the locks on both X and Y. Next, T_2 obtains the lock on Y.

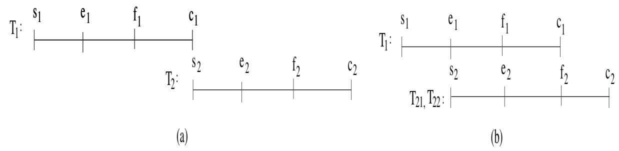


Figure 4. Processing: (a) without SDTP (b) with SDTP

Now, we explain the processing employing SDTP approach. Initially, $tree_X = (X_1)$, $tree_Y = (Y_1)$ and $tree_Z = (Z_1)$. First, T_1 obtains EX locks on both X and Y. It completes the execution T_{11}, with $RS_{11} = \{X_1, Y_1\}$ and $WS_{11} = \{X_2, Y_2\}$. On completion of T_{11}, X_2 is included in $tree_X$ as $(X_1(X_2))$ and Y_2 is included in $tree_Y$ as $(Y_1(Y_2))$. Afterwards, EX lock on both X and Y are changed to CX lock. (It can be noted that $tree_Y$ can be updated on completion of T_1's execution as Y resides at T_1's home site. However, $tree_X$ is updated on receiving the PREPARE message of 2PC protocol on behalf of T_1.) Next, T_2 starts two executions by taking Y_1 and Y_2 as inputs at e_1. The processing is depicted in Figure 4(b). The first execution of T_2 is T_{21} with $RS_{21} = \{Y_1, Z_1\}$ and $WS_{21} = \{Y_3, Z_2\}$ and the second execution of T_2 is T_{22} with $RS_{22} = \{Y_2, Z_1\}$ and $WS_{22} = \{Y_4, Z_3\}$. After completion of execution, Y_3 and Y_4 are included in $tree_Y$ as $(Y_1(Y_2(Y_4), Y_3))$ and $Z_2$and Z_3 are included in $tree_Z$ as $(Z_1(Z_2, Z_3))$. Now, without waiting for the completion of 2PC processing of T_1, T_2 proceeds with first phase of 2PC processing by sending the effect (RS and WS) of both T_{21} and T_{22} to the participating sites as part of $PREPARE$ message. When a participant site receives the $PREPARE$ message, it checks if it can commit a transaction. If so, after writing the $PREPARE$ message in the stable storage, it sends $VOTE\_COMMIT$ message to S_2. Otherwise, it sends $VOTE\_ABORT$ message to S_2. If all the participant sites respond with $VOTE\_COMMIT$ messages, S_2 takes the decision to retain one of the two executions as follows. After the completion of first phase of 2PC of T_2, the coordinator checks for ABORT/COMMIT decision of T_1. If T_1 commits, then the coordinator of T_2 takes the decision to retain the execution T_{21} as it follows the serial order $< T_1, T_2 >$. Otherwise, if T_1 aborts, the coordinator takes the decision to retain T_{22} as it follows the serial order $< T_2, T_1 >$ (This is under the assumption that T_1 is resubmitted after its abort and that no other transaction enters into the system.) Next, the coordinator sends this decision as a part of GLOBAL_COMMIT message to all the participants. In this way, both T_1 and T_2 can be processed in parallel without violating the serializability criteria.

5 Description of SDTP Algorithm

In this section, we first explain the terms used in the algorithm. Next, the algorithm is presented. Subsequently, we present a method to tune SDTP approach according to the available resources in the system. At the end of the section, we briefly discuss the behavior of SDTP approach against failures.

5.1 Terms used to present the algorithm

- Home site (HS_i):
  The site of origin of T_i is referred to as the home site (HS_i).
- Number of speculative executions of T_i (N_i) :
  It is an integer ($N_i \geq 1$), which represents the number of speculative executions carried out by T_i.
- Status of a transaction (ST_i):
  The $ST_i(X)$ is maintained for each data object X that is accessed by T_i. It takes one of the following values.

$ST_i(X)$	= waiting	: T_i is waiting for X or executing.
	= begin_commit	: T_i has started 2PC processing.
	= committed	: T_i has committed.
	= aborted	: T_i has aborted.

- Lock request (LR_{ik}):
  During the execution of T_i, HS_i sends LR_{ik} to S_k, whenever a data object is required at S_k.
- Active_queue of X:
  It is denoted by a_queue_X which is maintained at the resident site of each data object X. Consider that T_i accesses X. Then, triple $< T_i, TREE_X, status >$ is stored in a_queue_X. In this, T_i denotes transactions' identity. When T_i accesses X, $tree_X$ is copied to $TREE_X$. The status takes one of the two values: true or false. Suppose T_i accesses X. If $tree_X$ contains only one version, the status is assigned to true. Otherwise, if $tree_X$ contains more than one version, initially the status is assigned to false. It is changed to true whenever the preceding transaction terminates.
- Wait_queue of X: It is denoted by w_queue_X which is maintained for each object. It is used to store the lock requests that are waiting to access X.
- RES_{ik}:
  In response to LR_{ik}, S_k prepares the message RES_{ik} with two fields: type and $data\_values$. The type field takes one of the two values: true or false. The data values are assigned to $data\_values$ field.
- $confirm_i$: It is a set of data values which is maintained for a T_i at HS_i.

5.2 SDTP algorithm

We present SDTP algorithm, by employing dynamic two-phase locking algorithm for concurrency control. In this, the lock request is issued to a data object whenever it is required. Initially, each data object X is organized as a tree with X_1 as the root node. Also, w_queue and a_queue of all the data objects are empty. For a data object, arriving lock requests are stored in its w_queue. The algorithm exchanges the different kinds of messages. At any site, depending on the message received, a specific action is invoked.

- New_transaction /* HS_i starts new T_i */
  When T_i (T_{i1}) starts at HS_i, the following variables are initialized: $N_i := 1$; $confirm_i := \phi$, $RS_{i1} := \phi$. A lock request message $LR_{ik} = (T_i, X)$ is sent to S_k where X resides.
- LR_{ik} /* S_k receives it from HS_i */
  1. If both a_queue_X and w_queue_X are empty, the following actions are performed. Otherwise, step (2) is followed. X_1 is locked. The information "$ST_i(X) := waiting$" is stored at S_k. The triple $< T_i, tree_X, true >$ is stored in a_queue_X. The message RES_{ik} is prepared with "$RES_{ik}.type := true$", and "RES_{ik}.data_values := value of X_1". Next, RES_{ik} is sent to HS_i.
  2. If LR_{ik} is in conflict with preceding LR_{jk} which is already waiting for X in w_queue_X, LR_{ik} is stored in w_queue. Otherwise, if preceding LR_{jk} has accessed X in a_queue_X, the steps given below are followed.
     - 2.1 If $ST_j(X) = waiting$, LR_{ik} waits until the value of $ST_j(X)$ becomes "$begin\_commit$".
     - 2.2 If ($ST_j(X) = begin\_commit$), the following actions are followed. LR_{ik} is removed from w_queue_X. The triple $< T_i, tree_X, false >$ is stored in the a_queue_X. The information "$ST_i(X) = waiting$" is stored at S_k. The message RES_{ik} is prepared with, "$RES_{ik}.type := false$"; "$RES_{ik}.data\_values$:=node values of $tree_X$". Next, RES_{ik} is sent to HS_i.

- RES_{ik} /* HS_i receives it from S_k*/

  1 If RES_i.type=true, all N_i executions are continued, with X_1. For $q = 1 \ldots N_i$, RS_{iq} is updated as $\{X_1\} \cup RS_{iq}$. Otherwise, if RES_i.type=false, step (2) is followed.

  2 Consider that, RES_{ik}.data_values has n values in it. Each execution T_{iq} (q=1 $\ldots N_i$) is replaced by n executions taking each X_p (p=1 $\ldots$ n) as input. The actions in this step are presented in the following module.

  ```
  begin
    p:=0;  /* k, p and q are integer variables */
    for q:=1 to N_i do
    for k:=1 to n do
        begin
          p:=p+1;
          T_ip:=(i,p);
          RS_ip := {X_k} ∪ RS_iq;
  /*The execution T_ip is carried out with RS_ip */
        end;
        N_i :=p;
  end.
  ```

  3 If T_i completes execution, the following actions are performed. The message $PREPARE_{ik}$ is prepared with RS_{ij} and WS_{ij} of all N_i executions along with respective execution number (i,j) for data objects residing at S_k. Next, as a part of the first phase of 2PC, $PREPARE_{ik}$ is sent to each participant site S_k.

- $PREPARE_{ik}$ /*S_k receives it from HS_i */

  1 The message $PREPARE_{ik}$ is stored in the stable storage. Next, S_k checks if it can commit T_i. If so, step (2) is followed. Otherwise, step(3) is followed.

  2 The new versions received with $PREPARE_{ik}$ are included in the respective trees as the children of corresponding version in RS_{iq} ($1 \leq q \leq N_i$). The ES/EX locks for the data objects accessed by T_i at S_k are changed to CS/CX locks. For each data object X that is accessed by T_i at S_k, $ST_i(X)$ is changed to *begin_commit*. (The waiting transactions are granted locks on these data objects.) Next, for each data object X that is accessed by T_i at S_k, the status of T_i is checked in a_queue_X. If status=true, the corresponding node value in $TREE_X$ is sent along with $VOTE\_COMMIT_{ik}$ to HS_i. Otherwise, if status=false, it is put to wait either for time-out, or to get terminating decision of preceding transaction.

  3 The message $VOTE\_ABORT_{ik}$ is sent to HS_i.

- $VOTE\_COMMIT_{ik}$ /* HS_i receives it from S_k*/

  1 If node value X_q is received with $VOTE\_COMMIT_{ik}$, $confirm_i := \{X_q\} \cup confirm_i$. If HS_i receives $VOTE\_COMMIT_{ik}$ messages from all participants, step (2) is followed. Otherwise, it waits for the time-out.

  2 From all RS_{iq} ($1 \leq q \leq N_i$) of T_i, some q is selected such that $confirm_i \subseteq RS_{iq}$. Next, message $GLOBAL\_COMMIT_{ik}$ with execution number (i,q) is sent to all the participants.

- $VOTE\_ABORT_{ik}$ /* HS_i receives it from S_k*/
  The message $GLOBAL\_ABORT_{ik}$ is sent to all the participants.

- $GLOBAL\_COMMIT_{ik}$ /* S_k receives it from HS_i*/
  From the stable storage of S_k, message $PREPARE_{ik}$ is retrieved. From this, execution number (i,q), which is received with $GLOBAL\_COMMIT_{ik}$ is retained. Suppose some $X_q \in WS_{iq}$. Then, $tree_X$ is updated with X_q as the root. Also, TREEs of all subsequent entries in a_queue_X is updated with X_q as the root. Next, for each data object X that is accessed by T_i at S_k, status of subsequent lock request in the a_queue_X is changed to true. $ST_i(X)$ is changed to committed. Next,"ACK_{ik}" is sent to HS_i.

- $GLOBAL\_ABORT_{ik}$ /* S_k receives it from HS_i*/
  The transaction is aborted. $ST_i(X)$ is changed to aborted. The new nodes included by T_i in $tree_X$ are removed along with their subtrees. Also, the nodes included by T_i in TREEs of all subsequent entries of a_queue_X are deleted along with their subtrees. Next, for each data object X that is accessed by T_i at S_k, the status of subsequent lock request in a_queue_X is changed to true. The message "ACK_{ik}" is sent to HS_i.

- ACK_{ik} /* HS_i receives acknowledgment message from S_k */
  After receiving ACK_{ik} messages from all the participants, HS_i terminates T_i by writing the *end_of_transaction* record in the log.

5.3 SDTP approach : limited resources

It can be observed that, SDTP approach does not require extra number of messages as every message is sent with the messages of 2PC protocol. But, it needs extra memory and processing power to carry out speculative executions. In SDTP approach, the number of speculative executions that a transaction has to carry out increases exponentially with the increase in data contention. Since each speculative execution needs separate work space, the size of main memory available in the system can limit the number of speculative executions. With this limitation, processing cost may not be considered as a considerable overhead as current technology provides high speed parallel computers at low cost. To keep number of executions within the manageable limit according to the affordable size of the main memory, the number of speculative executions carried out by a transaction can be controlled with two variables : *versions_limit* and *executions_limit*.

For a data object, *versions_limit* variable limits the maximum number of versions allowed in its tree. If number of versions in the tree of a data object is more than the *versions_limit*, the lock request is put to wait. That is, the lock request waits for the termination of earlier transaction. On termination of earlier transaction, the tree is updated. Afterwards, if the number of nodes equals or less than *versions_limit*, the lock is granted to waiting lock request.

Let the amount of memory to carry out single execution be one unit. Then, N denotes the number of executions that can be executed by the system in parallel. The *executions_limit* variable is set to maximum number of speculative executions that can be carried out by the system at any time. After getting responses to the lock requests, if the number of speculative executions crosses *executions_limit*, the transaction is rejected. The rejected transaction is resubmitted.

5.4 Behavior against failures

The termination and recovery protocols for 2PC protocol against failures such as coordinator timeouts, participant timeouts, coordinator failures, and participant failures have been explained in references [17, 18]. The same termination and recovery protocols can be employed in the case of SDTP approach with appropriate modifications.

6 Performance study

6.1 Simulation settings

The meaning of each model parameter for simulation is given in Table 1. The size of the database is assumed to be *db_size* data objects. The database is uniformly distributed across the all *num_sites* sites. The new transaction is assigned the arrival site which is chosen randomly over *num_sites*. The parameter *trans_size* is the average number of data objects requested by the transaction. It is computed as the mean of a uniform distribution between *max_size* and *min_size* (inclusive). The probability that an object read by a transaction will also be written is determined by the parameter *write_prob*. The parameter *trans_time* is the time required to transmit a message between sites. The parameter *local_to_total* is the ratio of the number of local requests to the number of total requests for a transaction. The parameter *res_io* is the amount of time taken to carry out i/o request and the parameter *res_cpu* is the amount of time taken to carriyout CPU request. Accessing a data object requires *res_cpu* and *res_io*. Also, each message of 2PC protocol requires *res_cpu* and *res_io* for processing. The total number of concurrent transactions active in the system at any time is specified by multiprogramming level (MPL).

The communication network is simply modeled as a fully-connected network. Any site can send messages to all the sites at the same time. The WAN behavior is realized by varying *trans_time*. This is true because, in DDBSs, even though the difference in the delay to receive responses for remote requests varies considerably, the transaction does not complete its execution, unless it receives responses from all the remote sites.

In the simulation we employ static two phase locking approach. In this, a transaction starts the execution only after receiving the locks on all the required data objects. The 2PC protocol is employed for the commit processing.

Parameter	Meaning	Value
db_size	Number of objects in the database	1000 objects
num_sites	Number of sites in the system	5 sites
trans_size	Mean_size of transaction	8 objects
max_size	Size of largest transaction	12 objects
min_size	Size of smallest transaction	4 objects
write_prob	Pr (write X/read X)	1
local_to _total	Ratio of local requests to total requests	0.6
res_cpu	Time required to carry out CPU request	15 msec
res_io	Time required to carry out I/O request	35 msec
MPL	Multiprogramming level	
trans_time	Transmission time between two sites	

Table 1. Model parameters with settings

The setting of *res_io*, *res_cpu*, *db_size*, *trans_size*, *min_size*, and *max_size* parameters are given in the Table 1 [19]. The *local_to_total* ratio for a transaction is fixed at 0.6 [20]. Thus, 60% of the data objects are randomly chosen from the local database and the 40% of the data objects are randomly chosen from the remaining database sites. We have set *write_prob* parameter to 1; i.e., a transaction writes all the data objects it reads. With these settings, by varying MPL values, sufficient variation in the data contention is realized.

In the simulation experiments, we have evaluated the response time metric. The response time is the time spent by the transaction in the system. This is the difference between when the transaction first submitted and when the transaction decides to commit.

6.2 Unlimited resources

In this section, we first evaluated the performance without SDTP (locking and 2PC protocol) approach. In this locks are released only after completion of commit processing. Next, the experiment is repeated with SDTP approach. The simulation is carried out for a variety of MPL values and transmission time values assuming unlimited resources.

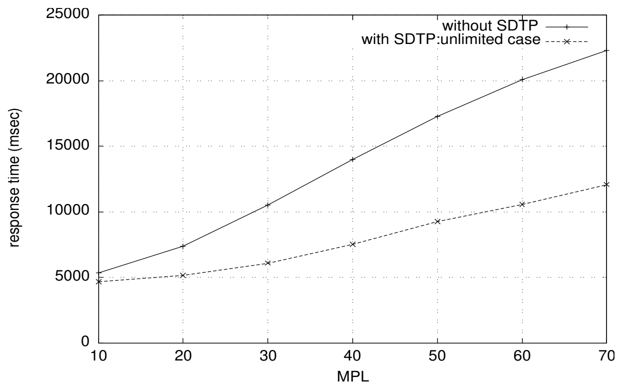


Figure 5. MPL versus response time at transmission time=1000 msec

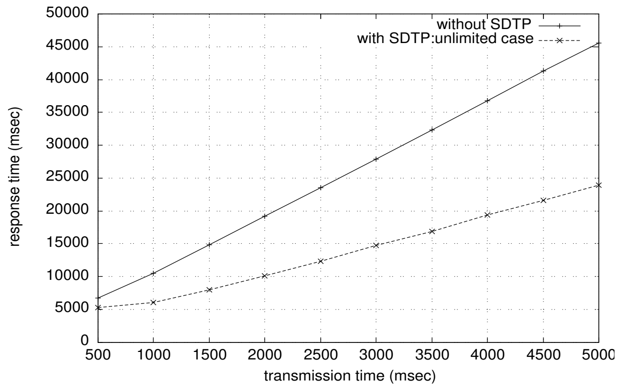


Figure 6. Transmission time versus response time at MPL=30

At different MPL values, Figure 5 shows the response time results. In DDBS, as we increase the MPL, both data and resource contention also increases. As a result, more number of transactions wait for the data objects. Therefore, more number of transactions carry out their executions in speculative mode. Consequently, increase in the parallelism reduces the response time of the transactions. As a result, SDTP approach exhibits better performance.

At different transmission time values, Figure 6 shows the response time results. In DDBSs, as transmission time increases, the transaction spends longer duration in the commit processing. As a result, more number of transactions spend longer time in waiting for the data objects. In SDTP approach, performance is improved as more number of transactions carry out their executions in speculative mode.

6.3 Limited resources situation

In this section, we have analyzed the impact of size of the main memory that will limit the number of speculative executions carried out by transactions. For these experiments, we have fixed the *execution_limit* value at 1000 and observed the performance at different *versions_limit* values.

At different MPL values and *versions_limit* values, Figure 7 shows the number of rejected transactions (resubmissions) due to the crossing of *executions_limit* and Figure 8 shows the corresponding response time results.

In SDTP approach, the number of speculative executions produced by a transaction depends on the number of objects it has accessed and the number of uncommitted versions in the tree of each object. From Figure 7, it can be observed that, by fixing the *versions_limit* value, if we increase MPL, the number of rejections keep increasing. This is true because as we increase MPL, the data contention increases. Due to this, a transaction conflicts on more number of data objects. As a result, more number of transactions cross *executions_limit* value. Similarly, it can be observed that higher *versions_limit* results more rejections.

From Figure 8, it can be observed that by increasing *versions_limit*, the response time is improved considerably towards the unlimited case. This is true because, the increase in the *versions_limit* value results more number of transactions to carry out their executions in speculative mode. Also, the rejections have little effect on the response time performance of the system. Because when *versions_limit* increases, transactions are rejected and resubmitted again. It is obvious that the rejection probability for those transactions which conflict on large number of data objects is high. When a transaction is rejected its locks are released. This leads to the execution of waiting transactions which conflict on lesser number data objects.

At different transmission time values and *versions_limit* values, Figure 9 shows the rejected transactions due to crossing of *execution_limt* and Figure 10 shows corresponding response time results.

From Figure 9, it can be observed that with increasing *versions_limit*, more number of transactions are rejected. However, at the fixed *versions_limit* and MPL, the number of rejections is same at different transmission times. Because, by increasing the transmission time, the waiting for the data objects is increased but the contention for the data objects is same.

From Figure 10, it can be observed that as we increase *versions_limit*, the response time decreases significantly towards unlimited case. Because, increasing *version_limit* causes more transactions to explore speculation.

From above experiments, it can be observed that by selecting appropriate *versions_limit* and *executions_limit* values we can achieve considerable improvement in the response time metric at higher resource conflicts and longer transmission times.

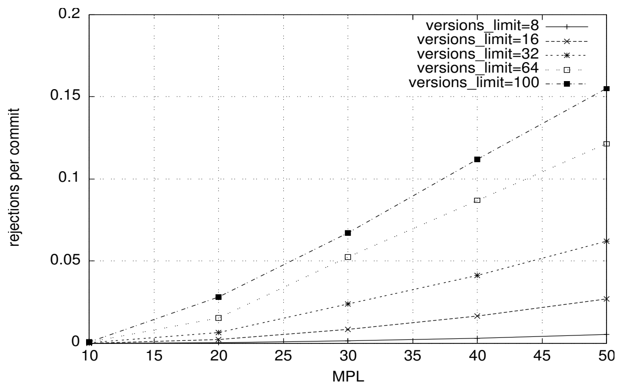


Figure 7. MPL versus rejections: transmission time=1000 msec *executions_limit*=1000

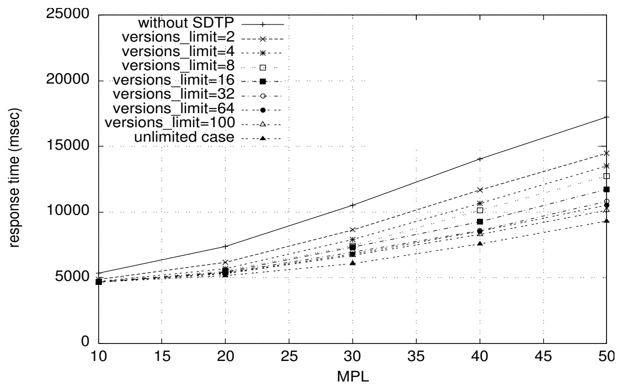


Figure 8. MPL versus response time: transmission time = 1000 msec, *executions_limit*=1000

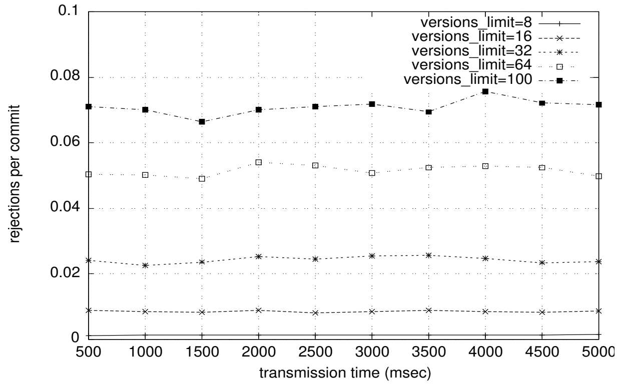


Figure 9. transmission time versus rejections:MPL=30, *executions_limit*=1000

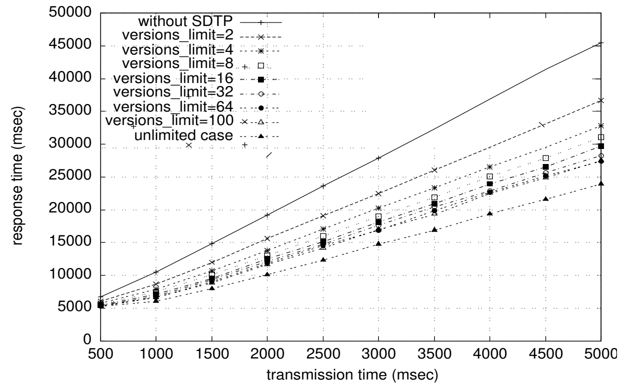


Figure 10. transmission time versus response time: MPL=30, *executions_limit*=1000

7 Summary and conclusions

In this paper, we have proposed speculative distributed transaction processing approach in which a transaction releases the locks on the data objects on completion of its execution. By this, the waiting transaction accesses the required data objects and carries out speculative executions by reading the data object values read and written by preceding conflicting transactions. Before the end of commit processing, it retains the appropriate execution based on the abort/commit decisions of preceding conflicting transactions. Using this approach, conflicting transactions can be processed in parallel without violating the serializability criteria. This approach is free from the cascading aborts. The SDTP approach requires extra processing and memory resources to carry out speculative executions. Through the simulation study, we have evaluated the performance of SDTP approach. It has been found out that even with limited resources, SDTP approach significantly minimizes the response time in the case of higher resource conflicts and the longer transmission times. The proposed approach suits best for the WAN environments where the message transmission time dominates the processing time. In this paper, we have presented SDTP approach by employing 2PC protocol. In case of 3PC protocol, SDTP approach exhibits an improved performance as an extra round of message transmission is involved. As a part of future work we will conduct extensive performance study.

Recent advances in communication technology improves the transmission bandwidth significantly. However, we are still suffering from message latency and it will be very difficult to resolve this problem in near future. The SDTP plays an important role under such latency sensitive database applications.

Acknowledgments

This work is (partially) supported by Grant-in-Aid for Creative Basic Research # 09NP1401: "Research on Multimedia Mediation Mechanism for Realization of Humanoriented Information Environments" by the Ministry of Education, Science, Sports and Culture, Japan.

References

[1] K.R.Eswaran, J.N.Gray, R.A.Lorie, and I.L. Traiger, The notions of consistency and predicate locks in a database system, *Communications of ACM*, November 1976.

[2] J.N.Gray, Notes on database operating systems: in operating systems an advanced course, *Volume 60 of Lecture Notes in Computer Science*, 1978, 393-481.

[3] P.A.Bernstein, and N.Goodman, Concurrency control in distributed database system, *ACM Computing Surveys*, 13(2), June 1981, 183-221.

[4] H.T.Kung, and J.T.Robinson, On optimistic methods for concurrency control, *ACM Transactions on Database Systems*, 6(2), June 1981, 213-226.

[5] R.A.Lorie, Physical integrity in a large segmented database, *ACM Transactions on Database Systems*, 2(1), March 1977, 91-104.

[6] D.G.Severance, and G.M.Lohman, Differential files: Their application to the maintenance of large databases, *ACM Transactions on Database Systems*,1(3), September 1976, 256-267.

[7] D.Skeen, Nonblocking commit protocols, *proc. of ACM SIGMOD*, June 1981.

[8] P.A.Bernstein, V.Hadzilacos and N.Goodman, *Concurrency control and recovery in database systems*(Addison-Wesley, 1987).

[9] R.Agrawal and D.J.Dewit, Integrated concurrency control and recovery mechanisms: design and performance evaluation, *ACM Transactions on Database Systems*, 10(4), December 1985, 529-564.

[10] P.Spiro, A.Joshi, and T.K.Rangarajan, Designing an optimized transaction commit protocol, *Digital Technical Journal*, 3(1), Winter 1991.

[11] B.Bhargava, Y.Zhang, S. Goel, A study of distributed transaction processing in an internetwork, *Volume 1006 of Lecture Notes in Computer Science*, 1995, 135-152.

[12] Azer Bestavros and Spyridon Braoudakis, Value-cognizant speculative concurrency control, *proc. of the 21th VLDB Conference, 1995*, 122-133.

[13] Azer Bestavros, Using speculation to reduce server load and service time on the WWW, *proc. of ACM Fourth International Conference 0n Information and Knowledge Management (CKIM'95)*, 1995, 403-410.

[14] Ramesh Gupta, Jayant Haritsa and Kirti Ramamritam, *Revisiting commit processing in distributed database systems*, ACM SIGMOD, 1997, 486-497.

[15] A.K.Elmagarmid, A survey of distributed deadlock detection algorithms, *ACM SIGMOD RECORDS*, 15(3), September 1986, 37-45.

[16] E.Knapp. Deadlock detection in distributed databases, *ACM Computing Surveys*, 19(4), December 1987, pp.303-328.

[17] M.Tamer Ozsu and Patrick Valduriez, *Principles of distributed database systems*, Prentice-Hall, 1991.

[18] S.Ceri and P.Pelagatti, *Distributed databases: principles and systems*(New York:McGraw-Hill, 1984).

[19] R.Agrawal, M.J.Carey and M.Livny. Concurrency control performance modeling: alternatives and implications, *ACM Transactions on Database Systems*, 12(4), December 1987, 609-654.

[20] Alok N.Choudhary, Cost of distributed deadlock detection:A performance study, *in 1990 proc. of IEEE Conference on Data Engineering*, 1990, 174-181.

[21] W.H.Kohler, A survey of techniques for synchronization and recovery in decentralized computer systems, *ACM Computing Surveys*, June 1981, 149-183.

[22] D.Skeen and M.Stonebraker, A formal model of crash recovery in a distributed system, *IEEE Transactions on Software Engineering*, SE-9(3), 1983, 219-227.