August 1983

Report No. STAN-CS-85- 1053

Transaction Classification to Survive a Network Partition

by

Peter M. G. Apers

Gio Wiederhold

Department of Computer Science

Stanford University
Stanford, CA 94305

Transaction Classification to Survive a Network Partition

Peter M. G. Apers †
Gio Wiederhold
Stanford University

ABSTRACT

When comparing centralized and distributed databascs onc of the advantages of distributed databases is said to be the grcater availability of the data. . Availability is dcfined as having access to the storcd data for update and retricva!, cven when some distributed sites are down due to hardware failures. We will investigate thc fimctioning of a distributed database of which the underlying computer network may fail. A classification of transactions is given to allow an implcmentation of different levels of operatability. Some transactions can be guaranteed to commit in spite of a network partition, while others have to wait until the state of potential transactions in the other partitions is also known. An algorithm is given to compute a classification. Based on historics of transactions kept in the differcnt partitions a merge of histories is computed, generating the new values for some data items when communication is re-established. The algorithm to compute the merge of the histories makes usc of a knowledge base containing knowledgc about the transactions, to decide whcther to merge, dclcte, or dclay a transaction.

1. Introduction

When comparing centralized and distributed databases one of the advantages of distributed databases is said to be the greater availability of the data [1, 10]. Availability is defined as having access to the storcd data for update and retrieval, cven whcn distributed sites arc down due to hardwarc failures. The reason for improved availability is that data can be stored redundantly; increasing the probability that data is available on a site which is up.

One disadvantage of distributed systems is the possibility of a network partition. A partition

This work was supported in part by the Netherlands Organization for the Advancement of Pure Research (Z.W.O.) and in part by thc Knowledge Base Management System Project from the Defense Advanced Research Projects Agency, contract N39-84 C 0211. This research is part of a joint projcct callcd Distribution Expertise for Database Applications (DEDA-project).

† Current address: Dept. of Mathematics and Computer Scicnce, Vrijc Univcrsitcit, De Boclelaan 1081. 1081 HV Amsterdam.

is defined to be a split of a computer network into several subncts, each being ablc to operate without conununication among them. Depcnding on thc nctwork topology it is possible that a crash of only one site may causc a partition of the nctwork. Now updates to rcplicatcd data bccomc problematical, since consistcncy cannot be maintaincd betwecn thc partitions. One rather ad hoc solution that has bcen suggested in litcraturc is to count thc numbcr of sites in a partition and if thc partition contains morc than half the numbcr of sitcs of the original network the database in this partition of the nctwork may proccss updatcs as normal [11]; the copies of thc database in the other partitions may only be used for retrieval. When the partition is repaired the updates madc are propagated to the other partition. Obviously, this solution provides only limited availability, and perhaps no availability for updates.

By means of an example we will point out the specific problcms of updating thc database during network partition. Image we have a database for a bank containing the balances of the accounts. For simplicity, we assume that at every site a complete copy of the database is available. Let the balance of account *A* before network partition bc $500. During network partition the bank still wants to allow banking transactions to be executed. A deposit is made to account A of $100 in one partition and a withdrawal of $200 in another partition. Because there is no communication between the two partitions we end up with two balances: $600 and $300. After communication has been re-established the bank Wants to have one balance for account A. Obviously, from the two balances alone we cannot compute a new balance that reflects the banking transactions madc in the two partitions. So, keeping the balances in the two partitions is not sufficient.

Another problem is caused by integrity constraints. For example, two withdrawals of $400 and $300 take place in the two partitions. Both are valid because the two partition balances remain non-negative. However, after merging the transactions in the two partitions a negative balance of $200 CR ($500 – $300 – $400) results. Prohibiting the eventual possibility of a final credit balance will severely restrict the handling of banking transactions. So, strict maintenance of global integrity constraints may be reasonable during normal operation but may disable the functioning of the database during partition.

A third problem relates to the interaction between the database and its environment. For example, a manager inquircs about thc balances of current, savings, and trust accounts during network partition to decide about a loan. Although the balances of thc accounts in the local partition may be sufficient to obtain the loan, it docs not mean that after merging the deposits and withdrawals from all partitions the balances are the same. So, thc question is what kind of decisions can be madc based on the data during partition and, also, whether thc user must be informcd that thc answer to his qucstion has changed after merging thc transactions from the partitions.

In the approach we will discuss in this paper we will providc a number of diffcrent lcvels of service for the transactions of the uscrs during network partition. Some transactions will be guaranteed to commit aftcr communication has bccn rc-cstablishcd and othcrs may commit dcpcnding on constraints. I lcncc wc scparatc thc notions of committing a transaction from making thc changes of thc database causcd by thc transaction permanent. This mcans that during the partition we actually have scvcral databases, which havc to bc mergcd again into onc database aftcr communication has bccn rc-cstablishcd. ‘1’0 classify thc transactions a knowlcdge basc is uscd containing knowledge about the transactions, thcir interaction and their cffects on the real world.

A relatcd problem is found in a database which is distributed ovcr a loosely coupled network, where the nodcs, typically persona1 computers, have a high degree of autonomy. In this case, copics of part of a database are availablc on pcrsonal computers that can be hooked onto the network at will. At that time changes madc to thc local copy of thc database will havc to be merged with changes madc to other copics.

Related research can be found in [4] whcrc the problem of synchronization after communication has bccn re-cstablishcd, is discussed. In [6] a Highly Available System is discussed which ensures the robustness of partitions. Systcm R* [12] emphasizcs autonomy for each partitioq. In [8] a technique is introduccd for dynamic allocation of primary copies and recovery if the database becomcs partitioned duc to a network partition. Data-Patch [7] is a technique to generate one big transaction to account for all changes made to the database in one partition; the goal is to find a scrialization of thcsc transactions. Another technique, prcscnted in [3], tries to minimize thc amount of work to merge the transactions from databases in different partitions. In [S] an algorithm is prcscntcd to detect conflicts between transactions executed in diffcrcnt partitions. In [2] a system which permits distributed rcsourccs to be sharcd in a resilient manner, is discussed. Finally, in [9] automatic conflict resolving in a distributed file system is discussed.

The paper is organized as follows. In Section 2 we will introduce somc notions and define our goal of surviving a network partition more formally. In Section 3 we discuss the history of data items in the database. In Section 4 a classification of the databasc transactions is introduced. In Section 5 an algorithm will be presentcd to compute the merge of the transactions of the different partitions. In Scction 6 it is shown that even though system-wide communication may not occur, WC are still able to commit or undo transactions. In Scction 7 implcmentation issues are discussed. Finally, we end with a summary and a conclusion.

2. Notions and Goal Statement

In this section we will introduce some notions and formalize our goal to survive a network partition.

A *(logical) databasc* is a collection of logical data items. ***A logical data item*** will bc thc unit of access at thc logical level of a databasc; for cxample, thc database itself, a relation, or a tuple. 'I'hc granularity of a logical data itcm dcpcnds on thc transactions acccssing it. A logical data item can be represented by multiple *physical data items.* So, we allow for replicated data.

A *network partition* is a situation in which a certain site is not able to communicate with another site, which is not necessarily down. We call the subnetworks that are caused by the network partition *partitions.* Due to a network partition the set of physical data items belonging to one logical data item will be partitioned into a number of subsets, where each subset is contained in a partition. For each such subset of physical data items there will be a *partition data item.* Obviously, there are no more partition data items for one logical data item than there are partitions. A *partition database* is the set of partition data items of all physical data items stored in a particular partition of the network. We will use the term *data item* if we either mean a logical data item or a partition data item.

Figure 1 shows the relationship between a logical data item, its partition data items and its physical data items during a network partition. We assume that the network partition causes the set of physical data items to be split into three subsets.

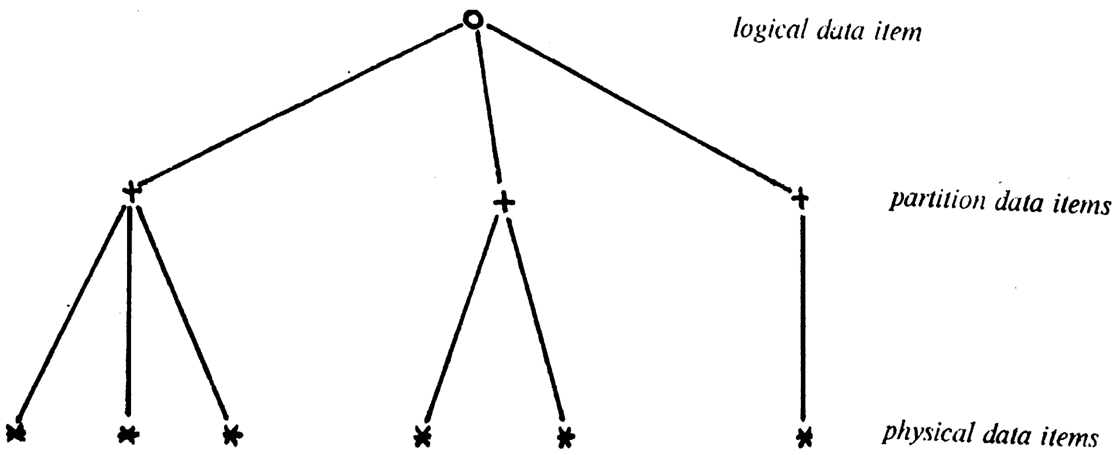

Fig. 1. Relationship logical, partition and physical data items

A *transaction* is a sequence of database operations that transform the database from one consistent state to another. A *database operation* consists of either a read or write of a data item. A database operation accessing a data item actually refers to physical data items for reading or writing. If a change is made to a data item then all physical data items of that data item have to be changed. A *concurrency control algorithm* is a scheduler which interleaves the execution of database operations in such a way that the resulting values of the data items are the same as if the transactions were executed serially.

For a transaction we define an ***intentional read-set** IR* and ***intentional write-set** IW* as being a description of the set of data items it will read, respectively write. For simplicity we assume that thc intentional rcad-sct includes thc intentional writc-sct. We assumc that thc concurrency algorithm makes usc of *time stamps.* Whcn a transaction is given to thc databasc for execution it is labclcd with a timc stamp, which is unique for thc partition in which it runs, plus its site numbcr. A transaction is callcd *committed* if its effects on thc logical databasc havc bccn madc pcrmancnt.

The goal for surviving a nctwork partition is to givc thc users as much as possiblc thc impression that thcir transactions acccss the logical databasc instead of thc partition databasc. To do so thc values of thc physical data itcms are saved upon discovery of a network partition. Transactions in each of the partitions executc bascd on thc locally availablc data items. After communication has bccn re-cstablishcd betwccn two or more partitions, thc transactions executed in thc different partitions are redonc on the values of the physical data items savcd bcforc nctwork partition. It may happen that somc of thc transactions cannot bc rcdonc because somc constraint has bccn violated. For thc uscrs this means that the transactions and their consequences havc to be undonc. Because undoing certain transactions may be impossible, because it would createc an inconsistcncy with the outside world, we will classify the transactions into a guaranteed to commit class, a conditionally committable class, and a class of transactions that should not be executed during network partition. To guarantce the commit of ccrtain transactions thc database administrator has to determine which contraints havc to bc relaxed, and to weigh whether the constraints are more important than the inability to commit thc transactions.

3. History of Data Itcms

In this scction WC will introduce thc notion of thc history of a set of data itcms and USC it to dctcrmine the diffcrent levels of opcratability that can be offcred to thc transactions.

A simple definition of the ***history*** of a data item is a scqucnce of triples:

<item_id, old-value, new-value),

wherc *item_id* stands for the idcntification of the logical data itcm, ***old-value*** stands for the value of the data itcm bcforc the transaction, and ***new-value*** for thc valuc aftcr execution of thc transaction.

During nctwork partition each partition data item will dcvclop its own history. Thc problcm is that these histories will have to bc mergcd to define the logical data item history after communication has been re-established. A *merge* of two histories is dcfincd as the result of interleaving the two histories such that thc result is a valid history, while prcscrving the partial ordcrs of both. By a ***valid history*** of data item *i* we mean a scqucnce of ***<i, old-value, new_value>*** triples such that the ***old-value*** of a triplc equals thc ***new-value*** of thc prcvious tuplc. Obviously, with this dcfinition of valid history we are not able to tncrgc thc two histories. For examplc, Fig. 2 shows two historics of thc balance of account A. The balance of thc account bcforc network partition was ***$500.*** In history 1 there are two transactions: a withdrawal of $300 and a deposit of $400; and in history 2: a deposit of $200 and a withdrawal of $600. Based on these tuples we are not able to construct a valid history because the value $500 (the *old_value* of the first transaction in History 2) never returns as a *new_value*.

History 1	*History 2*
<*A*, $500. $200>	<*A*, $500, $700>
<*A*, $200, $600>	<*A*, $700, $100>

Fig. 2. Histories of two partition data items

A more complete *history* of a data item is a sequence of sextuples:

<item_id, transaction type, values of input variables, IR, IW, pre-condition>,

where *pre-condition* is a constraint which has to be satisfied in order to execute a transaction instance of *transaction type* with the given values of the input variables. The *IR* is included to easily determine which data items were accessed. A transaction may access a data item either by its ID or by contents. In the first case the set consists of ID and in the latter it is a set described by a condition that has to be fulfilled by the data items to be accessed. The reason that we distinguish between accessing data items by ID and by value is to allow some flexibility. If a transaction accesses data items by value then in the merge of the histories it cannot be guaranteed that the transaction accesses the same data items because their values may have changed. If in the final merge the transaction should access the same data items they should be accessed by their IDs. ***A valid merge of histories*** is now defined as a sequence of sextuples such that the pre-conditions of each transaction are satisfied, and that the order of the transactions in the original histories are maintained.

Keeping a history of the kind definad above gives a better chance of merging the two histories. For example, the two histories in Fig. 3, which correspond to the above example, are mergeable into a valid history. The quintuples are labeled for future reference. The superscript refers to the history.

A merge of the two histories may consist of: W^1, D^2, D^1, W^2. Not every merge is, of course, allowable. The pre-conditions may cause a problem in determining the merge of two histories. For example, the sequence W^1, D^2, W^2, D^1 violates a precondition, because after the execution of D^2 the balance is $400, which is not sufficient to withdraw $600. To make a merge always possible

History 1

W^1 <*A*, *withdrawal*, (*A*,300), (*A*,*balance*), (*A*,*balance*), exist(*A*) and GE(balance(*A*),300)>

D^1 <*A*, ***deposit.*** (*A*,400), (*A*,*balance*), (*A*,*balance*), exist(*A*)>

History 2

D^2 <***A, deposit.*** (*A*,200), (*A*,*balance*), (*A*,*balance*), exist(*A*)>

W^2 <***A, withdrawal,*** (*A*,600), (*A*,*balance*), (*A*,*balance*), exist(*A*) and GE(balance(*A*),600)>

Fig. 3. Histories consisting of quintuples

and to give the users some kind of guarantee about the execution of their transactions we will classify them, relax constraints and/or adjust transactions.

Another constraint on the validity of a merge of histories is the ***external time.*** As far as the database is concerned, any merge of histories satisfying the definition of validity is fine. However, if the database has to report to the outside world, for instance, to compute service charges and produce monthly statements, the order of the merge must coincide with an order that is acceptable in the real world. For example, banking transactions that were handled on the same day should appear in the monthly statement on the same day. Therefore, we assume that for each transaction besides the quintuple also a time stamp is kept. During partition the clocks at the different sites can not be synchronized and may drift apart. Therefore, and also to allow for some flexibility in merging histories, the time stamps of the transactions are only used to secondarily order the transactions from different partitions.

In the next section we will classify the transactions.

4. Classification of Transactions

In this section we will discuss the interaction of transactions and their effects on the real world. A transaction may access data items either by key or value. If during network partition a data item is accessed by key this data item should also be available after the communication has been re-established. Therefore, the existence of the data items should be part of the pre-condition of the transaction. If, on the other hand, data items are accessed by value it might happen that the data items accessed in the partition differ from the ones accessed by the transaction in the merged history. The latter may have been caused by transactions in other partitions.

In the next two subsection we will propose a classification for predefined updates and discuss different ways of using retrievals.

4.1. Update Transactions

In this subsection we will provide the database administrator with tools to put the updates into different classes for which different levels of service are provided. In the following we will use the word transaction instead of update because as we will see in the next subsection some of the retrievals will be turned into updates.

The classification consists of the following classes of transactions:

1. *Unconditionally Committable Class* (*UCC*): a set of transactions belongs to the *UCC* if the transactions can be committed as long as the database in the partition is the whole database, i.e. the execution of a *UCC*-transaction in one partition cannot violate the precondition of a KC-transaction in another partition,
2. *Conditionally Committable Class* (CCC): a transaction belongs to the CCC if commitment of the transaction on the logical database cannot be guaranteed and undoing the transactions does not lead to inconsistencies with the real world,
3. *Non-Committable Class* (NCC): a transaction belongs to the NCC if undoing the transaction will lead to an inconsistency with the real world that cannot be resolved from within the database.

A simple approach to the network partition problem would be to place all transactions in the conditionally committable class, and determine which transactions have to be undone after communication has been re-established. Some transactions cannot be undone because undoing them would result in an inconsistency with the real world. Therefore, they should either be put in the Unconditionally Committable Class or the Non-Committable Class.

Our approach is to combine the service that can be provided by the DBMS with the service that is required by the organization to continue functioning as well as possible. The way to do this is by relaxing integrity constraints and thereby making it possible to guarantee the commit of certain transactions. For example, a bank may have as a policy *never* to allow a withdrawal from an account with a negative balance. During a network partition a customer may withdraw from different partitions leading to a negative balance after merging the histories on this account. If the bank would not allow this, *the withdrawal* would fall into the class of non-committable transactions, because the DBMS is not capable of forcing the customer to return the money. But this may be an unacceptable situation. Therefore, the bank may change its policy by dropping the constraint concerning the negative balance.

To classify the transactions operating on a set of data items we need to know their pre-conditions and their post-conditions. The *pre-condition* and the *post-condition* of a transaction on the database are first order logic expressions. The pre-condition specifies a condition that is required for normal execution of a transaction, i.e. the transaction does not violate any integrity constraints. The post-condition of a transaction expresses what happened to the data items specified in the pre-condition or created by the transaction. To do so it makes use of the following predicates:

EX(D) means that data item D is accessible,

INC(D,A) (DEC(D,A)) that the value of attribite A of data item D has increased (decreased),

CH(D, A) means that attribute A of data item D has changed its value.

This information will be kept in ***the** Transaction Interaction* of the transactions on a set of data items. Let *Pre*(T) and *Post*(T) denote the pre- and post-condition of transaction T. These pre- and post-conditions may make me of comparison operations such as GE(x,y) with the obvious meaning. *Post*(S) *violates* *Pre*(T) if we are not able to prove that *Post*(S) $\supset \neg$ *Pre*(T) is false. With a simple, but specialized theorem prover we try to prove that *Post*(S)$\supset\neg$*Pre*(T) is false. If theorem prover is not able to do so either by lack of information or insufficient time it concludes that ***Post(S)*** violates *Pre*(T). This specialized theorem prover needs to know about the predicates that are used and how they interact. For example, if the Post(S) is DEC(D,A) and the *Pre*(T) is GE(A(D),x) then it concludes that *Post*(s) violates *Pre*(T) because it knows that (DEC(D,A) $\supset \neg$GE(A (D),x)) $\equiv$ true.

transaction	input	pre-condi tion	post-condition
open_account (O)	A	**not** EX(A)	EX(A)
close_account **(C)**	A	EX(A) and GE(balance(A),0)	**not** EX(**A)**
withdrawal (W)	(A,x)	EX(A) **and** GE(balance(A,x)	EX(A) and DEC(A,*balance*)
deposit (D)	(A,x)	EX(A)	EX(A) **and** INC(A,*balance*)
money_transfer (M)	(A,B,x)	EX(A) **and** EX(B) **and** GE(balance(A),x)	EX(**A) and EX(** B) **and** DEC(A,*balance*) and INC(B,*balance*)

Fig. 4. Transaction Interaction for transactions on accounts

Figure 3 displays the Transaction Interaction for transactions on accounts: it shows that the transactions on the account A can only take place after the creation of the account by the *open_account*; we assume that *open_account* is given an account number. Also, all transactions on A should have finished before executing the *close_account*. Jn between these two transactions, the transactions *withdrawal*, *deposits*, and *money_transfer* may take place under the condition that their pre-conditions are satisfied. Their effects are indicated by the predicates EX, INC, DEC, and CH.

Before presenting the algorithm, which computes the classification, we need to introduce one more notion. A transaction T is *execution-dependent* on transaction S if T cannot be committed in case the execution of S is undone. E.g., S creates a new data item and T operates on it. The execution of T depends on the execution of S in the sense that if S is undone T has to be undone as well.

Based on the Transaction Interaction of a set of transactions operating on a set of data items we are able to determine into what classes the transactions fall. The classification is computed by the algorithm CLASSIFY shown in Fig. 5. It has two input parameters, ***ST,*** the set of transactions working on the set of data items under consideration and, ***RT,*** the set of transactions that should belong to the *UCC* according to the database administrator. Because the transactions in *RT* will be execution-dependent on transactions that create data items these will be added to the set ***RT.*** To put the transactions of ***RT*** in the set *UCC* violations of pre-conditions have to be removed. The part where these pre-conditions are relaxed and/or transactions are adjusted form an interactive session with the database administrator. Then the transactions in *UCC* are made execution-independent of transactions outside *UCC* either by relaxing preconditions, adding these transactions to UCC as well, or by prohibiting the execution of these transactions during network partition. . The choice is up to the database administrator.

```
procedure CLASSIFY=(set of transaction ST,RT)void:
begin
    procedure UNITE=(set of transaction X, Y)set of transaction:
    begin
        relax some of  the pre-conditions and/or adjust transactions in the sets X and Y to remove
        violations of pre-conditions:
        update the pre-condition entries in the Transaction Interaction;
        UNITE: = $X \cup Y$
    end:
1: add transactions that create data items to RT;
2: UCC: = 0:
    for $T \in RT$ do UCC: = UNITE(UCC,{T}) od;
3: while $T \in UCC$ is execution-dependent on $S \notin WCC$
    do
        case let database administrator decide what to do of
        relax:       relax precondition of T such that T no longer execution-dependent is on S;
        unite:       UCC: = UNITE(UCC,{S});
        non-comm: put S in the set NCC
        esac
    od:
4: put the transactions in ST that are not in UCC or NCC in the sets CCC or NCC depending on
   whether non-committing may lead to inconsistencies
end
```

Fig. 5. Procedure *CLASSIFY*

Theorem 1 The set *UCC* produced by procedure *CLASSIFY* contains only unconditionally committable transactions.

Proof A transaction T in thc sct UCC can bc committcd on thc logical databasc bccause thc othct transactions in that sct cannot violatc its prc-condition. and the transactions on which it is cxccution-dcpcndcnt arc cithct in UCC or in NCC. I Icncc, T is an unconditionally committablc transaction.

□

Now an examplc. Assumc that procedurc $CLASSIFY$ of Fig. 5 is callcd with as first parameter the sct shown in Fig. 4 { O, C, W, D, M } and as sccond parameter thc set { W, D }. The execution of thc statcmcnt at the linc labclcd 1 adds transaction O to the set RT. During cxecution of the whilc-statcmcnt at thc line labclcd 2 wc have to unitc {O} and {D }. This can be done by dropping thc precondition that A should be uniquc. A simple way to do this is to lct *opcn_account* itsclf gcneratc a unique number by prcfixing a locally unique numbcr with the branch numbcr whcre the account will reside. From now on we assume that O stands for this adjusted transaction. In thc sccond cxecution of this whilc-statcmcnt wc have to unitc (0, D} and {W}. This can be achicvcd by dropping the constraint "GE(balancc(A),x)" in the prc-condition of *withdrawal.* In thc execution of the whilc-statcmcnt labclcd 3 we notice that none of the transactions in UCC are execution-dependent on C. So, in the statcmcnt at thc line labclcd 4 C is either put in the set CCC or NCC depending on whcther non-committing may lcad to an inconsistency with the rcal world.

4.2. Retriev al transactions

The diffcrcnce bctwecn a rctricval transaction and an updatc transaction is that a rctricval does not changc thc contcnts of the databasc. Howevcr, during its cxecution it may writc on output dcviccs such as a terminal or a printer. This output is a function of the contcnts of thc databasc, which is not up-to-date. The knowledge basc may be uscd to indicate whcthcr the result is reliable. This may be done bascd on the frequency with which data items have bccn changcd in the past. Rut, in gcncral, the result of a rctrieval during partition is not reliable.

.For somc applications this unreliability is acceptable. If a pcrson inquires about the balance of his account and the databasc answcrs: "Thc balancc of your account shows $500 but due to hardware problems not all banking transactions havc bccn proccsscd," the person can bc pcrfcctly happy with that rcsponse. If, on thc other hand, a managcr inquires about ncgativc balances of accounts to evaluatc thc crcditworthincss of their ownc1s the conscqucnccs might bc more sevcrc.

If the result of a rctrieval has to havc crcditablc validity it is best to turn it into an update. Then thc output dcviccs arc considcrcd to be part of thc database. Thc output mcssagcs will form just a privatc workspacc of thc databasc. to bc uscd for thc rctricval. Bccausc nonc of thc other retrievals and updntcs bclonging to UCC can violate thc precondition of thc rctricval thc rctricval transaction bclongs automatically to thc CCC. During the timc that thc mcrgc of thc histories is computed the previously computed output, which was kept in the private workspace, is compared with the newly computed output. If there is any difference the user is notified. Retrievals that are not turned into an update need not be recorded in any of the histories.

To reconsider the inquiry of the manager, if the manager had turned the retrieval into an update an updated list of accounts having a negative balance would be produced at the point in the merged history where his retrieval was executed.

In the next section we compute a merge of the histories of data items in different partitions using the classification proposed in this section.

5. **Computing** a **Merge** of Histories

In this section we will discuss the problem of merging the histories of data items from different partitions. An algorithm will be proposed, which on the one hand uses the classification of the transactions and on the other hand uses a knowledge base containing knowledge about the transactions, their interaction, and their effects on the real world. This algorithm will later be used within an algorithm to determine whether transactions can be comtnitted or have to be undone.

After communication has been re-established between two partitions the transactions executed in these partitions have to be merged. The merge is done based on the histories that were kept for the sets of partition data items in the different partitions and the ‘classification of the transactions. In general, these histories from the two partitions consist of sequences of conditionally and unconditionally committable transactions. If the re-establishment of the communication is system-wide the resulting database is the logical database and all transactions in the final history can be committed on the logical data items. Otherwise, the history is kept for future merging.

To be able to operate on the histories we will represent them in a graph called the History Graph. ***A History Graph*** consists of nodes representing the transactions and directed edges representing a dependency among the transactions. Transaction T_i *depends* on T_j if $TS(T_i)$>$TS($ 7”) and also if $TS(T_i)= TS(T_j)$ but T_i appears after T_j in the history of the data items, and $IR\,(T_i)$ intersected with $IW(T_j)$ is not empty. In this case there will be an edge from T_i to T_j. T_j is called a *ancestor* of T_i and T_i is a *dependent* of T_j. The case of identical time stamps has to be considered because in computing a valid merge of histories some of the titne stamps may be changed. In that case the order of the transactions in the history determines the dependency. The “roots” of the History Graph are the ***last committed*** *transactions* on the logical data items before network partition, indicated by *LCT*. The post-conditions of all *LCT*s indicate that all data items have changed.

Figure 6 shows a History Graph for two accounts *A* and *B* with transactions for *deposits* (*D*), *withdrawals* (*W*) and *money_transfers* (*M*) from one account to another. The roots of the graph are the last committed transactions on the accounts, indicated by *LCT(A)* and *LCT(B)*.

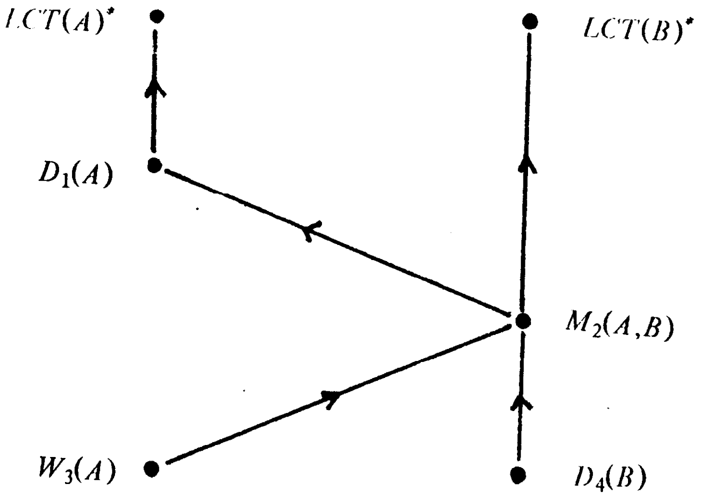

Fig. 6. A History Graph for accounts ***A*** and *B*

The procedure we propose to compute a merge of histories is called *MERGE* and is shown in Fig. 7. The knowledge base required is shown in Fig. 8. The procedure starts with creating one graph, called the ***RESULT,*** by identifying the last committed transactions of partition data items of the same logical data item. Then it goes through the histories to determine which transaction will appear in the merge. Every time it executes the while-statement it looks for a *candidate mergeable transaction*, which is a transaction whose ancestors have already been merged and which has the smallest time stamp. Procedure *CMT* computes this transaction and while doing so it might delete some transactions from the CCC and reconnect their dependents to the last committed transactions of the corresponding data items. The rules concerning the order in which the transactions from the CCC are deleted from the cycle are not shown in the knowledge base. Procedure *CMT* throws away identical transactions to allow procedure *MERGE* to merge two histories whose beginnings are identical because they concern transactions executed in the same partition before another network partition occurred.

The procedure *MERGE* accesses the knowledge base to determine which action to take. The actions it can take on the T_1 being the *CMT* are merge, delete, and delay. ***Merging*** a transaction T_1 into the final history means that the transaction will be part of the history of the data items on which it operates. Every dependent ***T*** of a ancestor of T_1 is made a dependent of T_1 if the intentional read set of T has a non-empty intersection with the intentional write set of T_1. ***Deleting*** a transaction T_1 means that it is removed from the history. All dependents of T_1 arc made dependents of the ancestors of T_1 depending on the intersections of the intentional read sets and intentional write sets. ***Delaying*** a transaction T_1 means that the order of execution between transactions

```
procedure MERGE=(history H1,H2;ref history FH;ref decision DL)void:
begin
    history graph HG1, HG2, FinalHist;
    ref decision DecList: = nit;
    transaction T1: = null transaction;

    procedure CMT= (history graph R)transaction:
    begin
        transaction T;
        repeat
            let T be the candidate mergeable transaction of R;
            if no such T exists
                then
                    there is a cycle in the history graph R;
                    access knowledge base to determine which conditionally committable transaction
                    should be deleted
                else if T is the same as the previous candidate mergeable transaction
                    then T: = null transaction
            fi
        until $T \neq$ null transaction;
        CMT:=T
    end;

    let HGI and HG2 be the history graphs of HI and H2, respectively;
    let RESULT be the history graph that results from identifying the LCTs of HGl and HG2;
    white there are transactions which are not marked merged in RESULT
    do
        T1:=CMT(RESULT);
        access knowledge base ( RESULT, T1, ACTION);
        case ACTION in
        merge:      (make every transaction T whose ancestor is also a ancestor of Tl and $IR(T) \cap$
                    $IW(T1) \neq 0$, a dependent of Tl;
                    foreach dependent S of Tl
                    do
                        if $TS(S) < TS(T1)$ then T&S ):= TS(T1) fi
                    od;
                    mark Tl as being merged;
                    add decision to merge Tl to DecList)
        delete:     (connect a dependent T of Tl to a ancestor Ai of Tl, if $IR(T) \cap IW(Ai) \neq 0$;
                    remove Tl from RESULT;
                    add decision to delete Tl to DecList);
        delay:      (make Tl a dependent of the last UCC-transaction, whose pre-condition might
                    be violated by the execution of Tl, say T, and remove edge from Tl to Ai,
                    where Ai is an ancestor of both Tl and T)
        esac
    od;
    FH: = FinalHist;
    DL: = DecList
end
```

Fig. 7. Yroccdure *MERGE*

from different partitions will deviate from the time stamp ordering. T_1 is made a dependent of the last *UCC* transaction *T* whose precondition is violated by the post-condition of T_1. The dependency of T_1 on the mutual ancestor of T_1 and 7' is removed from thc history graph.

Thc rules in the knowledge base should be such that a transaction belonging to the *UCC* is merged into thc result history. The other rules concerning the transactions from the CCC should be such that most of them are merged into the resulting history. Note that a transaction belonging to the CCC may only be delayed when there is system-wide communication. The knowledge base is established at database design time and is fully replicated. Figure 8 only shows a minimum set of rules. A possible extension is to test in rule 2 whether the transaction violates the pre-condition of another transactions belonging to the CCC and then to decide to delete the first transaction, thus selectively backing out transactions [5].

```
Rule I:
if T belongs to UCC then ACTION:=merge fi;

Rule 2:
if T belongs to the CCC and its pre-condition is not violated and
   T does not violate the pre-condition of a UCC transaction
     then ACTION: = merge
fi;

Rule 3:
if T belongs to the CCC and its pre-condition is violated
     then ACTION: = delete
fi;

Rule 4:
if T belongs to the CCC and its pre-condition is not violated and
   T violates the pre-condition of a UCC-transaction
     then ACTION: = if there is system- wide communication and
                          time stamp ordering is important then delete else delay
                    f          i
fi;
```

Fig. 8. Rules of a knowledge base

Theorem 2 Procedure MERGE together with the knowledge base as shown in Fig. 8 computes a valid history.

Proof A history is valid if the partial order imposed by the time stamp ordering in the different partitions is retained and if the pre-conditions of all transactions in the resulting history are fulfilled.

Because the transactions in the different partitions are considered in time stamp ordering the partial order is maintained. The pre-condition of a *UCC* transaction cannot be violated by other *UCC* transactions by definition and neither by CCC transactions due to the fourth rule of the knowledge base, and the fact that the *UCC* transaction is not execution-dependent on CCC transactions. Neither can a combination of *CCC* transactions and *UCC* transactions violate the pre-condition of a *UCC* transaction, because of the same reasons as above. Also, a CCC transaction whose pre-condition is violated is deleted from the resulting history graph.

□

Now we will give an example. Assumc that ***the*** History Graph *RESULT* aftcr identification of the *LCT*s looks as shown in Fig. 9. Furthermore, we assume that the transactions D, W, and M belong to the UCC and C bclongs to the CCC. Thc superscripts of the transactions indicate the partition in which they were executed and thc subscripts indicate the local time stamp. Transactions which are merged in the final history are labeled with an asterisk. First, M_1^2 is merged and D_2^1 and W_5^1 become its dependents. The result is shown in Fig. 10. Then D_2^1 becomes the *CMT* and is merged; W_3^2 becomes its dcpendcnt. Aftcr W_3^2 has been merged it is thc turn of C_4^2. Execution of C_4^2 violates the precondition of W_5^1 and therefore it is delayed. The rcsult is shown in Fig. 11. Now both W_5^1 and C_4^2 can be merged and the final history is obtained.

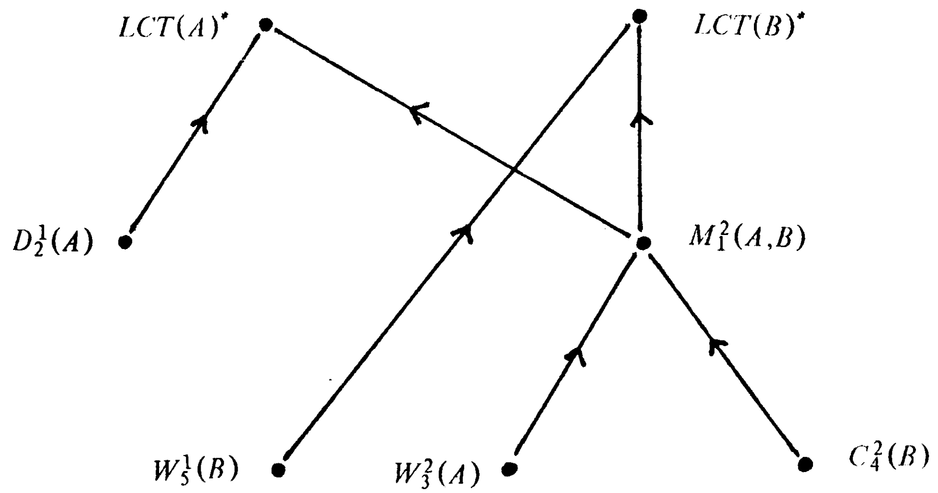

Fig. 9. History Graph ***RESULT*** after identification of *LCT*s

6. Merging and Committing Transactions

In the previous section we presented algorithm *MERGE* to mcrgc the histories of different partitions.. In this section we will show how this algorithm is used by the sites in the network after communication is re-established between all or sevcral partitions, and when the sites decidc about committing transactions. WC makc a distinction bctwccn merging and committing during system-wide communication and during network partition. The former can be used in environments where partitions are very unlikely, and the lattcr where simultaneous complete system-wide communication hardly ever occurs. An example of the latter environment is a loosely coupled system, e.g., a network consisting of personal computers, which have a high dcgrec of autonomy.

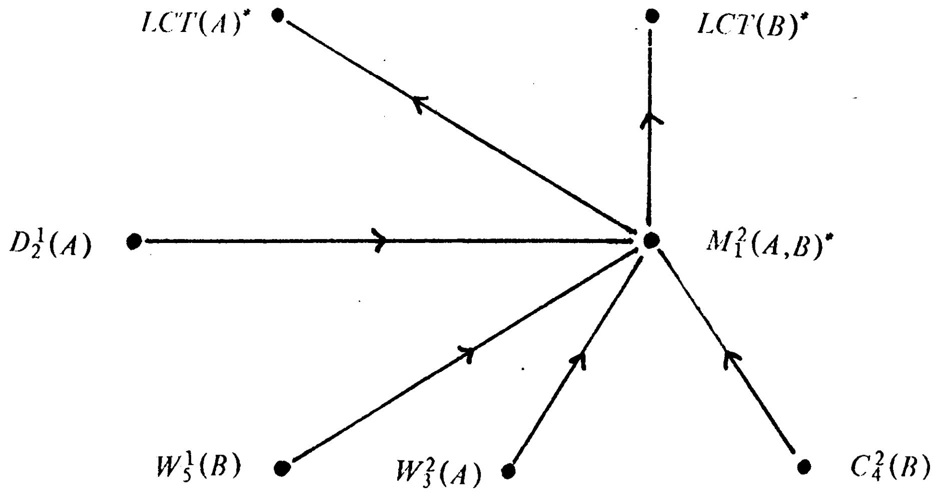

Fig. 10. History Graph ***RESULT*** after merging M_1^2

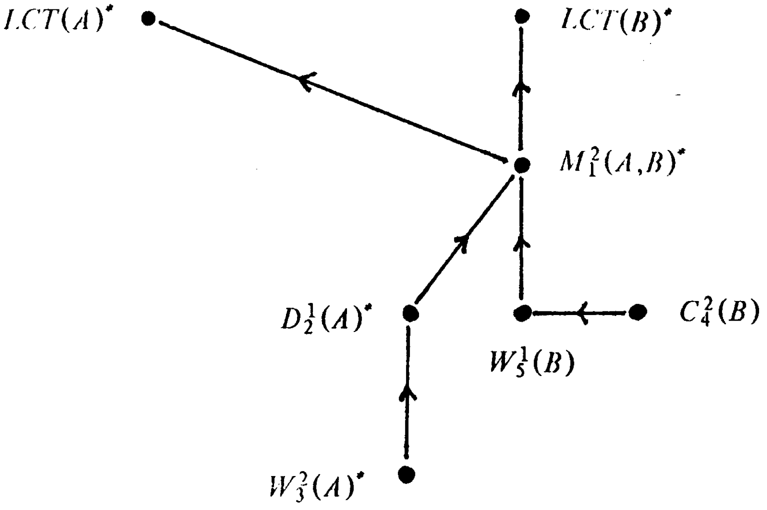

Fig. 11. History Graph ***RESULT*** after delaying C_4^2

6.1. Merging and Committing during Systems-Wide Communication

In an environment where a partition hardly ever occurs the best strategy is to wait with merging and committing until system-wide communication has been established again. Then algorithm ***MERGE*** can be executed to merge the histories of the different partitions. Because there is system-wide communication the algorithm may also decide to delay certain transactions to increase the number of transactions that will finally bc committed. After that, the list of decisions is scanned to 1) commit the transactions that arc merged, and 2) undo the transactions that are dcletcd. Undoing a transaction means that the user will bc notified of the fact that his transaction cannot bc committed and that appropriate actions should be taken. How committing and undoing affect the data at the different sites in the network will be discussed in the next section on implementation issues.

6.2. Merging and Committing during Network Yartition

In a loosely coupled network there will hardly ever be system-wide communication. So, merging and committing transactions cannot be postponed until these system-wide communications occur. In this section we will present an algorithm, which is executed when two or more partitions have communication again, to decide to commit or undo certain transactions.

Before presenting the algorithm we will introduce the notion of a partition-scenario graph. A ***partitiorl-scenario*** *graph* consist of nodes and directed edges. A node represents a partition and is labeled with the numbers of the sites in the corresponding partition. A directed edge from node P_i to node P_j means either that 1) P_i was part of P_j but that a partition caused P_j to split into P_i and some other partitions (P_i is called a ***phrtitiorz-node),*** or that 2) communication has been re-established between P_j and some other partition(s) to form P_i ***(Pi*** is called a ***merge-node).*** P_j is called **the** ***predecessor*** **of** ***Pi.***

All sites will keep track of their own partition-scenario graph. Because sites within one partition are able to communicate with each other they will have the same graph. Furthermore, because there is no system-wide communication none of the sites will have a partition-scenario graph that completely describes the current partitions. 'Therefore, the graph is only used .to keep track of the way partitions and reconnections occurred. Figure 12 gives an example of a. pa&on-scenario graph. For example, the node labeled 23456 is a merge-node and the node labeled 23 a partition-node. Note, the sites in the partition corresponding to node 1234 do not know anything about what happened to sites 5 and 6.

As the partition-scenario graph shows, two things might happen: a network partition or the re-establishment of communication. In the both cases the sites in each partition will update their partition-scenario graph. In the latter case algorithm *COMMIT_UNDO* is executed in the newly formed partition. Figure 13 shows algorithm ***COMMIT-UNDO.*** The algorithm first executes algorithm *MERGE* to compute the merge of the histories and the decision list ***DL*** of the partitions between which communication has been re-established again. A decision is either the merge or delete of a transaction ***T.*** Then it checks whether the decisions in ***DL*** are consistent with the ones taken by predecessor merge-nodes. If so, they are either committed or undone, and the corresponding decision is taken from the decision list. Note, the decision list *DL* consists of both the decisions taken by the current partition *P* and the delete decisions taken by the direct predecessor merge-nodes of ***P.***

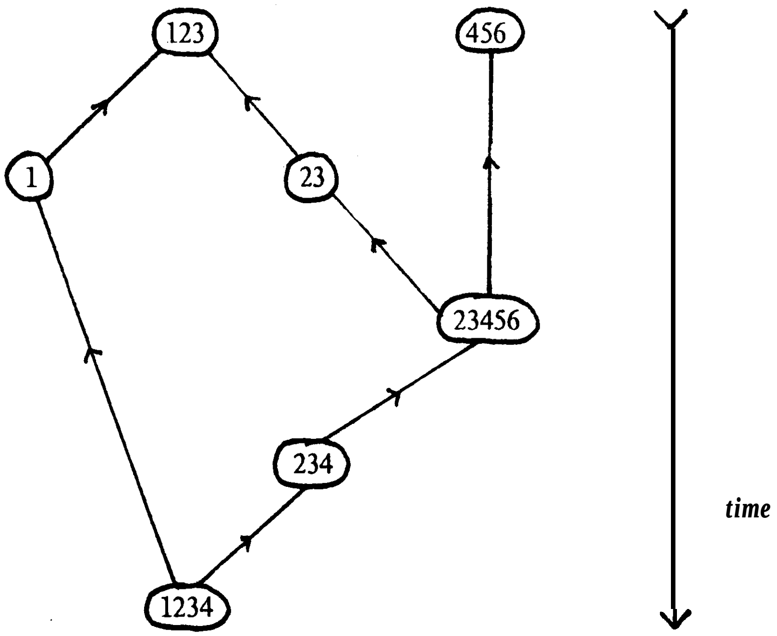

Fig. 12. An example partition-scenario graph of the node labeled 1234

```
procedure COMMIT_UNDO=(history H1,H2;partition P;ref history FH)void:
begin
    procedure all_sites_know=(decision D)boolean:
    begin
        set of site S;
        S . = sites in the partition P;
        visit recursively the predecessors of P in the partition-scenario graph and add the set of sites
        corresponding to the predecessors if they are merge-nodes and they have also taken decision
        D;
        all-sites-know := (SS = whole network)
    end;

    MERGE(H1,H2,FH,DL);
    DL := DL + delete decisions in DLs of predecessors of P without doubles;
    foreach decision D= (type,T) in DL do
        case type in
        merge:     if all_sites_know(D) then commit(T) fi;
        delete:    if all.. sires _know(D) then undo(T) fi
        esac;
    od;
end
```

Fig. 13. Algorithm ***COMMIT-UNDO***

Algorithm *COMMIT_UNDO* will bc executed in different partitions. It may occur that executions. of algorithm *MERGE* take different decisions about a transactions. But as wc will prove below the condition that all sites know and agree about a particular decision is sufficient to commit or undo a transaction inspite of different decisions taken by algorithm ***MERGE.***

Theorem 3 If T_y is committed or undonc in partition P all transactions T_x $(x<y)$ arc already committed or undone.

Proof The thcorcm holds of course for transactions known in partition P at thc timc 7” is committed or undonc, because the decision list is ordcred on the time stamps of thc transactions. So, the only thing we havc to show is that P knows all T_x with $x<y$. P knows that all sites know and agree about the final decision about T_y through thc application of algorithm ***MERGE*** in the different partitions (otherwise P could not dccidc to commit or undo T_y). Through these samc applications of algorithm ***MERGE*** P also knows about thc transactions executed at other sites before T_y.

□

Theorem 4 If partition P decides to commit or undo transaction T no other partition will take a decision to the contrary, and all unconditionally committable transactions will be committed.

Proof Partition P can only commit or undo transaction T if it can see in the partition-scenario graph that all sites agree. So, it is impossible that another partition will take another decision.

The precondition of a unconditionally commitable transaction U executed in partition P can only be violated by a conditionally committable transaction C with a smaller time stamp executed in a partition that is not a direct or indirect predecessor of P. Transaction C can never commit because it would require all sites to agree about merging C into the final history and all sites that know about U, the sites in all successors of P, will disagree. Thcrefore, all unconditionally committable transactions will commit.

cl

Example

Figure 14 shows the partition-scenario graph know by sites in the partition 45. In partition 1 transaction X_1 was executed, in 23 Y_2, and in 45 Z_3 (the subscripts are their local time stamps). All three transactions belong to the unconditionally committable class, and the post-condition of X violates the pre-condition of Y, and the post-condition of Y violates the pre-condition of Z.

In the merge-node 12 the following two decisions were taken: (*merge*,X_1) and (*delete*,Y_2). No transactions werc committcd or undone. In merge-node 35 the following two decisions were taken: (*merge*, Y_2) and (*delete*,Z_3). Again no transactions were committed or undone. In mergenode 134 three decisions arc taken: (*merge*,X_1), (***delete,*** Y_2), and (*merge*,Z_3). Transaction X_1 can still not be committed because site 5 does not know about the merge decision. In merge-node 25 two decisions ara taken: (***merge*** ,X_1) and (***delete,*** Y_2). The same is true here because the sites in partition 25 do not know yet that the sites in partition 134 have taken a merge dccisions about X_1. In merge-node 45 two decisions arc taken: (*merge*,X_1) and (***merge*** ,Z_3). The dccison list DL ***now*** consists of

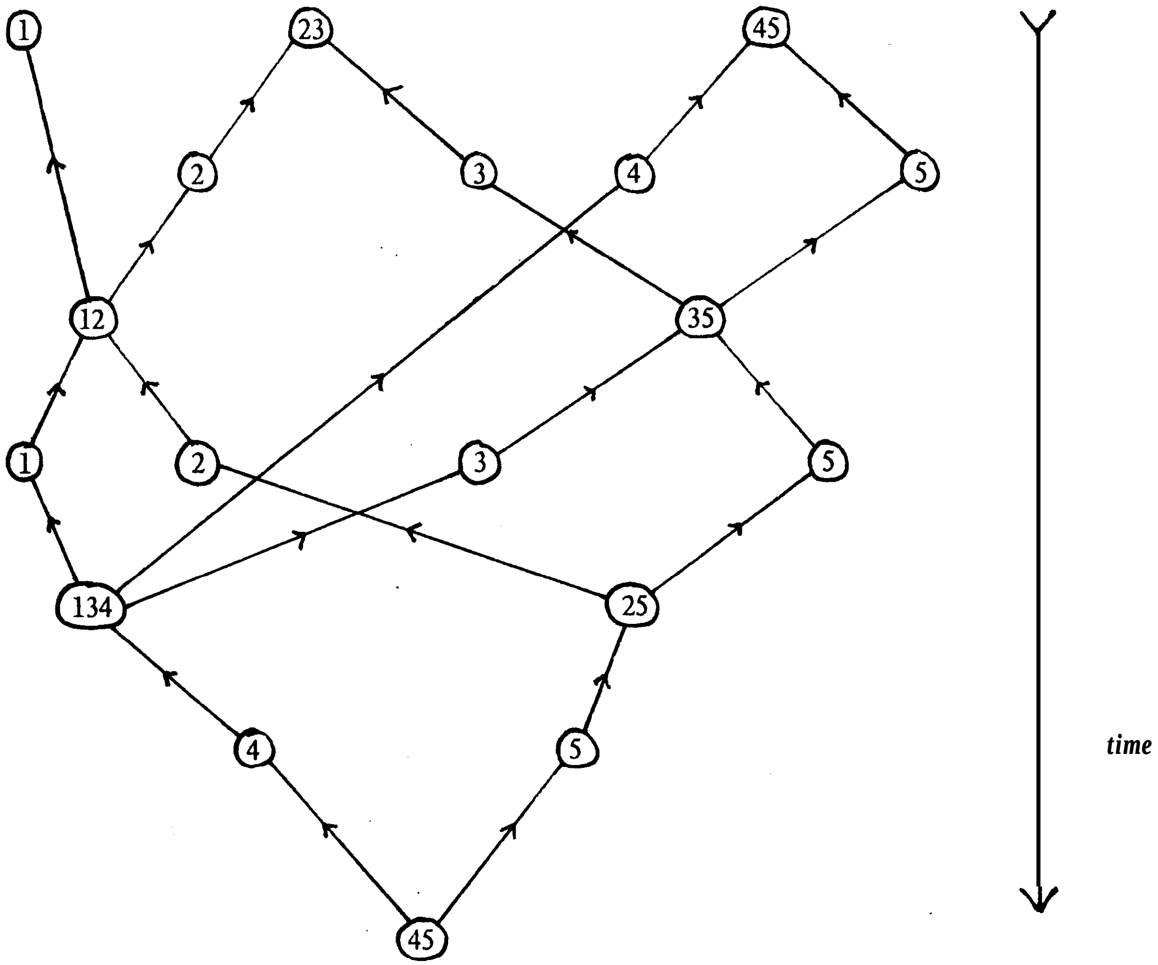

Fig. 14. Partition-scenario graph of partition 45

$(merge, X_1)$

(delete, Y_2)

$(merge, Z_3)$

From the partition-scenario graph we can see that all sites in the network know about ***(merge,&)*** and $(delete, Y_2)$. Thcreforc, transaction X_1 is committcd and Y_2 is undone. A final decisions about transaction Z_3 cannot be taken at this point.

□

7. Implcmcntation Issucs

Until wc now have carefully avoided to talk about implementation aspccts, bccausc we wantcd to explain the notions and algorithms without going into the details of the implementation. Furthcrmorc, thcrc is of coursc no unique implementation. In this section wc will merely show how everything can be implcmcntcd in a straightforward but incfficient way and indicatc ways of assuring the cfficiency of the algorithms.

Whcn a partition occurs the values of thc data itcms produccd by the last committed transactions arc saved. Transactions that are not able to commit bccausc they arc not able to access all their rcquired logical and/or physical data items are aborted and have to be executed again under the network partition mode.

During network partition transactions are executed in the local partition as if the available data comprise thc whole database. If, undcr normal circumstances, a transaction would commit it is put in thc history of the partition data itcms it accessed. If a transaction cannot access all its required data items it is aborted.

After communication has been re-established the histories of the partition data items of the different partitions are merged into a valid history and the transactions placed in this history are redone based on the values of thc physical data items savcd before the partition. This may be improved by starting from the values of the physical data items obtained during network partition and try to incorporate transactions executed in the other partition by undoing and redoing transactions [3]. This approach requires that the invcrscs of the transactions are known. By redoing wc mean that only thc database operations of the transaction are executcd again.

Conclusion

A schema has been proposed to allow for near normal fimctioning of a database during a network partition. A knowledge base is used to store knowledge about transactions, such as pre- and post-conditions and thcir classification. The classification indicates whether a transaction is guaranteed to commit or not after communication has been re-established. An algorithm is given to compute the classification of thc updates. This algorithm requires intcraction with the database administrator to rclax constraints and/or adjust transactions. Based on this classification and rules in the knowlcdgc basc the mcrge of histories can be computed after communication has been re-established. If system-wide communication is restored, the transactions in the computed merge of histories can be committed. Otherwise, thc history has to bc kept for future merges. A proof is given that thc rulcs in thc knowledge base guarantee that unconditionally committable transactions arc committed on thc logical database. Finally, thc problem of committing transactions was invcstigatcd in an environment where sy-stem-wide communication hardly ever occurs. An algorithm is given that decidcs to commit or undo transactions.

Acknowledgement

WC would like to thank Wolfgang Effelsberg, Goran Fagerstrom, and Arthur Keller for the fruitful discussions we had, and Hector Garcia-Molina for his comments on an earlier version of this paper.

Rcfcrcnccs

1. ADIBA, M., CHUPIN, J.C., DEMOLOMBE, R., G.GARDARIN,, AND BIHAN, J. LE, "Issues in Distributed Data Base Management Systems: A Technical Overview," *Proc. 4th Int. Conference **Very** Large Data **Bases,*** pp. 89-110 (September 1978).

2. ALSBERG, P.A. AND DAY, J.D., " A Principle for Resilient Sharing of Distributed Resources," ***2nd Int. Conference on Software Engineering,*** pp. 562-570 (1976).

3. BLAUSTEIN, B.T., **GARCIA-M• LINA, I-I.,** RIES, D.R., CHILENSKAS, R.M., **AND KAUFMAN,** CH.W., " Maintaining Replicated Databases Even in the Presence of Network Partitions," , CCA, Boston ().

4. **CHILENSKAS,** R.M., BLAUSTEIN, B., **AND** RIES, D.R., " Concurrency After the Fact," pp. 63 in ***Symposium on Reliability in Distributed Software and Database Systems,*** ed. Wiederhold,IEEE, Pittsburgh (July 1982).

5. **DAVIDSON,** S.B. **AND** GARCIA-M OLINA, H., "Protocols for Partitioned Distributed Database Systems," ***Proc. Symp. on Reliabilty in Distributed Software and Database Systems,*** pp. 145-149 IEEE, (198 1).

6. **EFFELSBERG,** W., FINKELSTEIN, ***S.,*** **AND** SCHKOLNICK, M., "Single Database Image in a Cluster of Processors," Report RJ 4175 (46103), IBM San Jose (Jan. 1984).

7. **GARCIA-M• LINA,** H., ALLEN, T., BLAUSTEIN, B., CHILENSKAS, R.M., **AND** RIES, D.R., " Data-Patch: Integrating Inconsistent Copies of a Database After a Partition," ***Proc. Third Symposium on Reliability in Distributed Software and Database Systems,*** IEEE, (1983).

8. MINOURA, T. **AND** WIEDERHOLD, ***G.,*** "Resilient Extended True-Copy Token Scheme for a Distributed Database," *IEEE TSE* SE-8(3) pp. 173-189 (May 1982).

9. **PARKER,** D.S. AND ET, AL., " Detection of Mutual Inconsistency in Distributed Systems," ***Proc. 5th Berkeley Workshop on Distributed Data Management and Computer Networks,*** pp. 172-183 (February 19800).

10. **ROTHNIE,** J.B. **AND GOODMAN, *N.,*** " A survey of research and development in distributed database management," ***Proc. 3rd Int. Conference Very Large Data Bases,*** pp. 48-62 (October 1977).

11. STONEBRAKER, M.R. AND NEUHOLD, E., "A Distributed Version of INGRES," *Proc. 2nd Berkeley Workshop Distributed Data Management and Computer Networks*, pp. 19-36 (May 1977).

12. WILLIAMS, R. AND ET, AL., "R*: An Overview of the Architecture," RJ 3325, IBM Research Laboratory, San Jose, Calif. (December 1981).