# A Classification of Update Methods for Replicated Databases

Stefano Ceri[*] Maurice A.W. Houtsma[†] Arthur M. Keller Pierangela Samarati[‡]

Computer Science Department
Stanford University

May 5, 1994

**Abstract**

In this paper we present a classification of the methods for updating replicated databases. The main contribution of this paper is to present the various methods in the context of a structured taxonomy, which accommodates very heterogeneous methods. Classes of update methods are presented through their general properties, such as the invariants that hold for them. Methods are reviewed both in their *normal* and *abnormal* behaviour (i.e., after a network partition).

We show that several methods presented in the literature, sometimes in independent papers with no cross-reference, are indeed very much related, for instance because they share the same basic technique. We also show in what sense they diverge from the basic technique. This classification can serve as a basis for choosing the method that is most suitable to a specific application. It can also be used as a guideline to researchers who aim at developing new mechanisms.

## 1 Introduction

One of the major obstacles to the development of distributed database applications consists in performing atomic updates. Though many commercial distributed database systems support atomic updates through the two-phase commit protocol, the use of this protocol has intrinsic limitations and disadvantages—including software complexity, cost and delay of execution, and reduced availability.

These limitations become even more severe in the presence of replicated databases. Several reasons make data replication very attractive, including the increased read availability (each application can select an arbitrary copy for access) and reliability (each copy may serve as a backup). However, all these features are compromised if all the copies are written atomically, because the write protocol requires all copies to be available at each write operation [23].

Requirements of applications accessing replicated data seldom insist that all copies be updated atomically. Copies may remain inconsistent for certain time intervals, or during specific operations, without compromising

[*] Stefano Ceri is partially supported by the LOGIDATA+ project of CNR Italy.

[†] The research of Maurice Houtsma has been made possible by a fellowship of the Royal Netherlands Academy of Arts and Sciences.

[‡] On leave from the University of Milan, supported by a scholarship from the Rotary Foundation

the applications' semantics [5]. Therefore, several protocols have been developed in distributed databases for updating replicated data without strictly requiring that all copies be atomically and synchronously updated (see, e.g., [3, 18, 24, 25, 29]). These protocols have very different origins; some of them were designed in academic or research environments, but many others were designed by engineers while developing advanced applications.

This paper presents a critical overview of protocols for updating replicated databases. We classify protocols, by understanding their common features and by stressing their relationships and dependencies. Protocols are analyzed in their *normal operation*, when the distributed database is functioning correctly, and in their *abnormal operation*, when failures occur. In particular, the most significant kind of failure that can affect a replicated database is a *network partitioning* [9]. It occurs when some nodes of the database are disconnected, though operational, and can perform conflicting updates. Strategies for dealing with network partitions are, therefore, analyzed. We concentrate here on update methods, and do not study the related problem of executing corrective actions when constraints do not hold anymore after temporary failures [15, 26].

This paper is organized as follows. In Section 2 we classify the update strategies, and describe the properties (also called *invariants*) that hold for each class of update strategy. We also discuss the requirements that may be achieved by reading operations (queries) in correspondence to each class of update strategy.

In Section 3, we informally describe the specific methods; we also explicitly indicate the dependencies between them. This comparison emphasizes the commonalities and differences between methods, which generally are not explicitly stated. As such, it constitutes a significant new contribution.

In Section 4, we deal with network partitioning. Again, we start by indicating the common features of the various recovery methods. After that, we describe each method.

# 2 Taxonomy of Update Methods

A taxonomy of update methods to replicated databases is built progressively.

## 2.1 Replicated data

Replicated data may be supported in three ways.

**Identical copies.** No copies have special rights, properties, or treatment.

**Primary/secondary.** One of the copies is selected as the primary copy, the other copies are secondary; the primary copy is either indefinitely decided (fixed primary), or it may transfer its role to another copy (non-fixed primary).

**Snapshot.** A snapshot is a view, defined on a database schema; view expressions allow to build either replicated or derived data. Snapshots are evaluated (or *materialized*) starting from a consistent database, and then installed at various sites, where they are not kept up-to-date. Periodically, they may be re-evaluated; snapshot re-evaluation is called *refresh*.

## 2.2 Update strategies

The update strategies described in the literature can broadly be classified as follows:

**Synchronous all.** All copies are updated synchronously; atomicity is guaranteed by the two-phase commit protocol.

**Synchronous available.** All available copies are updated synchronously; atomicity is guaranteed by the two-phase commit protocol. However, the write operation can take place even if some copies are not available; these copies will be updated later, through asynchronous mechanisms.

**Quorum-based.** Updates occur only on a subset of the copies, which form a so-called quorum. The copies in the selected quorum are updated synchronously. The copies which are not in the quorum are updated asynchronously.

**Primary/secondary.** Updates are performed on the primary copy. The secondary copies are updated asynchronously.

**Primary/backup.** Updates are performed on the primary copy. One of the non-primary copies is designated as a backup; this backup is responsible for recovery on failure of the primary copy. Other secondary copies are updated asynchronously.

**Independent.** Updates are performed on arbitrary copies; consequently, copies may become inconsistent. Inconsistencies can be automatically detected and sometimes corrected.

## 2.3 Invariants

Each of the above classes of update methods is characterized by an invariant condition:

**Synchronous all.** All replicas are up-to-date.

**Synchronous available.** All available replicas are up-to-date. By assuming no network partition, there is only one possible subset of available copies at a time.

**Quorum-based.** A write quorum of the replicas is up-to-date.

**Primary/secondary.** The primary copy is always up-to-date.

**Primary/backup.** The primary copy is always up-to-date (when accessible), the backup is the first copy to be subsequently updated, either synchronous or asynchronous.

**Independent.** No copy is guaranteed to be up-to-date.

## 2.4 Properties of update strategies

The following properties are relevant in the context of the all update strategies.

**Recoverable.** After a failure, a protocol is recoverable if it can produce a database state in which copies are consistent.

**One-copy serializability.** This property relates to the concurrency control; it holds when the interleaved execution of transactions on a replicated database is equivalent to a serial execution of those transactions on a one-copy database.

The following properties are only relevant in the context of primary/backup strategies.

**Order-preserving.** This property holds when transactions are executed at the backup site in the same logical order as at the primary site. Alternatively, transactions are *not order-preserving*; this may lead to inconsistencies.

**1-safeness.** Transactions are 1-safe if they first commit at the primary copy and are later propagated to the backup copy; committed transactions can be lost if the primary system fails.

**2-safeness.** Transactions are 2-safe if update transactions are either reflected at both the primary and the backup, or they are not reflected at all. Two-phase commit is needed to guarantee this property.

### 2.5 Query answers

In querying a replicated database, we may be interested into answers to queries with the following properties:

**Up-to-date.** The answer to a query is consistent with the current situation of the world.

**Consistent.** An application accesses data items from a consistent state (i.e., one produced by a serializable schedule). The database may have evolved, and therefore the answer may be not up-to-date.

**Partially Inconsistent.** The answer to a query is inconsistent, but the amount of inconsistency is bound, to:

- A number of versions (events) occurred since a consistent state.
- An absolute value, limiting the difference with the up-to-date value.
- The elapsed time since a consistent state was reached.

Table 1 indicates the type of read action that must be done in order to achieve the abovementioned properties, assuming that replicated data are managed by the strategies that were classified in Section 2.2.

## 3 Method Description

Table 2 summarizes the update methods for replicated copies; entries in the matrix refer to algorithms shortly described in the following.

### 3.1 Detailed Descriptions of the Methods

All methods described here are one-copy serializable, sometimes because of a special way of implementing them—in which case we explain so. Also, all primary/backup methods are order preserving and 1-safe. To some extend all methods are recoverable, we will go into this in Sec. 4.

**ROWA** *Read One Write All* [6]. A read operation may be executed on an arbitrary copy. A write operation has to be executed on every copy.

|  | Update Strategy |  |  |  |  |  |
|---|---|---|---|---|---|---|
| Type of answer | synchronous all | synchronous available | quorum-based | primary / secondary | primary / backup | independent |
| up-to-date | read any | read any available | read quorum | read primary | read primary | read any after recovering consistency |
| consistent | read any | read any in snapshot | read any in snapshot | read any in snapshot | read any in snapshot | read any after recovering consistency |
| partially inconsistent | N/A | read any | read any | read any | read any | read any |

Table 1: Read action in order to achieve answers' properties

- **ROWAA** *Read One Write All Available* [10, 17]. A read operation may be executed on an arbitrary copy. A write operation does not write all copies of the item: it ignores any copies that are down. Thereby, the problem of not-up-to-date copies is introduced. When a failed copy starts working again, it does not reflect the current database state. Transactions should be prevented from reading copies that have failed and recovered until these copies are brought up-to-date. All sites need to agree on the status table reflecting the sites that are up. Moreover, to ensure one-copy serializability, the implementation of ROWAA should be such that logical conflicts between transactions are reflected on physical copies as well. This has consequences, e.g., for the implementation of commit; each transaction has to check that its view of the system is still correct, to avoid one transaction updating an item inaccessible to another transaction and vice versa.

- **DOAC** *Directory-Oriented Available Copies* [7]. A directory of a data item consists of the references to the set of copies of that data item. Like a data item, a directory may be replicated to increase availability. A read operation is executed by finding a directory copy of the item, using that directory copy to locate an available copy of the data item, and finally reading such a copy. A write operation is executed by finding a directory copy of the item and issuing a write request to every copy of the data item listed in the directory. If some of the copies are unavailable the operation is aborted, the unavailable copies are deleted from all directories of the data item, and the operation is restarted. Note that this last step ensures one-copy serializability, as it guarantees that logical conflicts are reflected on physical copies. Also in this algorithm, all sites need to agree on the status table reflecting the sites that are up.

- **QC** *Quorum Consensus* [16, 20]. A non-negative weight is assigned to each copy of a data item. For each item, a read threshold $RT$ and a write threshold $WT$ are defined, such that both $2 \times RT$ and $(RT + WT)$ are

|  | Update Strategy |  |  |  |  |  |
|---|---|---|---|---|---|---|
| Replica | synchronous all | synchronous available | quorum-based | primary / backup | primary / secondary | independent |
| no primary | ROWA; MW | ROWAA; DOAC | QC; TQP; MW; VP; | N/A | N/A | MAR; IND |
| primary | N/A | N/A | N/A | RDF; RBP; DIP[1] | ASAP; QUC; DF; DR; CT | N/A |

Table 2: Classification of update methods

greater than the total weight of all copies of the item. A read (write) quorum of an item is any set of copies of the item with a weight of at least $RT(WT)$. Each read is translated into a set of reads on a read quorum and returns the most up-to-date copy—this one is guaranteed to contain the correct up-to-date value. Each write is translated into a set of writes on a write quorum. This algorithm assumes that it is possible to determine the 'recentness' of an item, for instance, by having a timestamp associated to each item. Note that the quorum is formed dynamically for each transaction.

**TQP** *Tree Quorum Protocol* [1, 2]. For each item, a logical tree is defined on its copies. For each data item a read/write quorum is defined in the following way. A read quorum is formed by selecting the root of the logical tree; if it is inaccessible due to failure, the majority of the children of the root forms a read quorum; if any of the selected nodes fail, the majority of its children is also required to form a read quorum (this happens recursively). A write quorum is formed by selecting the root and the majority of its children; for each selected node the majority of the children is also selected, until reaching the leaves. Execution of the actual read and write operations is as described for Quorum Consensus. Note that also here a quorum is formed dynamically.

**MW** *Missing Writes* [11, 12]. The assumption is made that a system knows reliable and failure periods of operation. During a reliable period, i.e., the system functions without any failure, Read One Write All is used. When a failure occurs, a switch is made to using Quorum Consensus. In this way, transactions execute either in normal mode or in failure mode. If they don't know that a failure occurred, they execute in normal mode. If they are aware of missing writes, they execute in failure mode. A transaction $T_1$ is aware of missing writes for a copy of an item if it attempts to write the copy without success, or if another transaction $T_2$ is aware of missing writes and there exists a path from $T_2$ to $T_1$ in the serialization graph, i.e., $T_2$ is executed before $T_1$.

**VP** *Virtual Partition* [13, 14]. Each site maintains a view consisting of the sites it believes it can communicate with. Within each view Read One Write All is used. A transaction is initiated at a given site. If the

[1] This is the only algorithm which explicitly supports the notion of a non-fixed primary. The other algorithms in this row can easily be adapted to accommodate for a non-fixed primary.

transaction contains read (write) operations on a given item, it can commit only if the view of the site at which the transaction started contains a read (write) quorum of the item. Contrary to QC and TQP, VP works with a fixed quorum assignment; the quorum can, therefore, not be adapted in case of failures.

**MAR** *Multi-Airline Reservation* Updates are executed independently on different copies and conflicts have to be resolved later [5]. In practice, several algorithms are used for this. One-copy serializability should be guaranteed by the reconciliation algorithm, which brings the database into a single consistent state.

A particular example of such an algorithm is the demarcation protocol [4]. It does not treat copies as true replicas anymore, but as independent copies. Local constraints are then formulated on each copy, which ensures that the independent copies can be merged again into a single logical copy later on. For instance, when there are 100 empty seats on an airplane and two independent copies, each copy is allowed to increase the number of seats sold by at most 50. Note that the demarcation protocol assumes operations that change the value of a data item to be commutative.

**IND** *Independent* Each copy operates independently, updates are executed only on one copy. Queries executed on a copy are therefore not guaranteed to give an up-to-date answer, but this is accepted by the applications. Copies can exchange information about their updates, thereby integrating update information from other sites. In this way the database may be brought to a fully consistent and up-to-date state. Hence, again it is the reconciliation algorithm that should guarantee one-copy serializability.

**RDF** *Remote Duplicate Database Facility.* [29] The database system is assumed to consist of a primary site and a backup site. At the primary site, undo/redo log entries are written in a master log for every transaction. As this log is written, a copy is sent to a control process at the backup site. When a transaction commits it is assigned a ticket, which determines the order in which transactions must install their updates (i.e., the changes made) at the backup. To commit a transaction at the backup site, all the transactions with an earlier ticket must already have been committed.

This algorithm is order-preserving, as transactions commit in exactly the same order at the backup site as at the primary site. It is not 2-safe but only 1-safe. If the primary site fails and some of its messages regarding committed transactions are lost, the backup site is not aware of these committed transactions and they are not executed.

**RBP** *Remote Backup Procedure* [19] The database system is assumed to consist of a primary site and a backup site. Each site is composed of many stores, among which the data are partitioned. At the primary site two-phase commit is used, and each transaction is assigned a ticket upon its completion. The log is propagated to the backup, either on a per transaction basis or on a per store basis. The log is kept at the action level, storing redo information, but no undo information; read-only transactions do not have to be propagated to the backup site.

At the backup site no real processing of transaction takes place. The backup site installs the changes that happened on the primary site, but does not perform any computation by itself. Transactions do not necessarily commit in proper ticket-order, as long as all transactions that a transaction that is about to commit depends on have already been committed, the result is correct. To detect such dependencies between transactions, the read-set of each transaction as well as the write-set is necessary. To ensure that the dependencies are satisfied, two-phase locking is used.

This algorithm is order-preserving, as transactions commit in the same *logical* order at the backup site as at the primary site; this is due to the two-phase locking that preserves read/write dependency among transactions. It is not 2-safe but only 1-safe. If the primary site fails and some of its messages regarding committed transactions are lost, the backup site is not aware of these committed transactions and they are not executed. The main difference between RBP and RDF is the parallelism that may be introduced by not enforcing commits at the backup site to be in proper ticket order.

**DIP** *Disaster protection* [18] All data items are fully replicated over all sites. Duplexed disks are used on each site to protect against disk crashes. Local updates are executed after checking the timestamp of the data item with time stamps of the copies of the data item on other sites. Thereby, updating an 'old' value is avoided. After updating the data item at the site of the transaction, the update is executed as a separate transaction at the other sites.

**ASAP** *As Soon As Possible* [7]. Write operations are executed on the primary copy. Committed writes are collected and sent to all other copies as independent transactions.

**QUC** *Quasi Copies* [3]. Information is controlled at a single central site, but the methods can also be applied in case of multiple central sites. Coherency conditions associated with a copy define the allowable deviations between an object and the copy. Coherency conditions can be related to time, version, or value. The central site, when operational, has to make sure that each remote site receives a message at least every $s$ seconds. If a site does not receive any messages for $s$ seconds it assumes the central site to have failed.

Propagation of updates on the primary copy can be done in four different ways:

- *Last minute*: updates on the replicas are delayed up to the point where a coherency condition can be violated;
- *Immediately*: updates on the primary are propagated as soon as they occur;
- *Early*: updates on the primary can be propagated at any time before violation of the conditions;
- *Delayed update*: the installation of updates on the primary site is delayed so that no condition is violated. The values are installed when convenient.

**DF** *Differential File* [28] A differential file is used to record the changes made on the primary copy, this differential file is then used to update the copies. The time of update depends on the algorithm, for instance, it can be on time of access of replica, on user demand, or periodically.

**DR** *Differential refresh* [21] A timestamp is associated with tuples at the base table. To each copy, a snaptime is associated, which reflects the time when it was last refreshed. To each tuple in the copy an address is associated stating where the corresponding base tuple is stored. The algorithm is started by sending the snaptime to the base relation and checking it against the time associated with the base tuples. Hence, tuples with a timestamp greater than the snaptime are refreshed (updated, inserted, or deleted) and afterwards the snaptime time is updated. This technique requires maintaining the status of every possible address.

This algorithm can be optimized in various ways. For instance, if each tuple in the base table stores the address of the previous tuple, it is not necessary to maintain the status of every possible address. The address space between the current address and the previous tuple is guaranteed to be empty. This information can be used when updating the copy.

**CT** *Copy token* [22] Each item is associated a logical copy and a set of physical copies. Writes are enforced on the logical copy and buffered until commit time, only then are they actually executed on the physical copies. Each logical item has a token associated with it, which is exclusively handed out to one of its physical copies. The copy that holds the token is regarded as the primary copy, i.e., all updates are performed on it and later propagated to the other copies. If the copy holding the token becomes unavailable, a new token may be handed out.

## 3.2 Dependencies between algorithms

Some of the algorithms that implement an update strategy depend on other algorithms, i.e., they use the other algorithms in their own implementation. Some of the algorithms may thus be classified as basic, while others are derived from these basic ones. Let us study this in some more detail.

In the following, we denote the dependencies between algorithms in the following way: $Y \leftarrow X$ denotes that algorithm $X$ is used in algorithm $Y$; $X = Y|Z$ denotes that algorithm $X$ is equal to $Y$ xor $Z$. We can therefore express dependencies between algorithms as expressions in this language.

$$MW = ROWA \mid QC + \text{system status management}$$

The missing write algorithm (MW) uses either ROWA or QC, depending on the state of the system. The additional component of this algorithm is keeping track of the system status. This means that information about missing writes has to be detected and propagated amongst the transactions.

$$DOAC \leftarrow ROWAA + \text{directory management}$$

The directory oriented available copy (DOAC) is based on ROWAA. The additional component of this algorithm is supporting a replicated directory structure, keeping track of all the copies of an item that are available in these directories, and updating this information in case of system changes.

$$TQP \leftarrow QC + \text{logical quorum management}$$

The tree quorum protocol (TQP) is based on QC. The additional component of this algorithm is supporting a logical tree defined on the copies, and a special strategy for defining a quorum.

$$VP \leftarrow QC, ROWAA + \text{view management}$$

The virtual partition algorithm (VP) is based on both QC and ROWAA. QC is used to define a static quorum for each view, within each view ROWAA is used. The additional component of this algorithm is defining and maintaining the views, keeping track of available copies, and updating the view information if necessary (e.g. including new sites in views).

$$RDF \leftarrow ASAP + \text{ticket handling}$$

The remote duplicate database facility (RDF) is based on ASAP. The additional component of this algorithm is handing out tickets to transactions that commit at the primary site, and using this information when installing updates at the backup site.

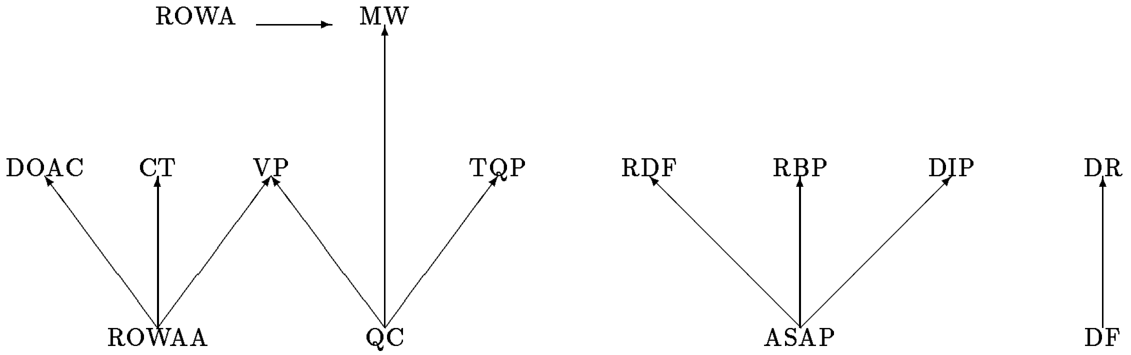

Figure 1: Dependencies between update methods

$$RBP \leftarrow ASAP + \text{ticket handling}$$

The remote backup protocol (RBP) is based on ASAP. The additional component of this algorithm is handing out tickets to transactions that commit at the primary site, and using this information when installing updates at the backup site. RBP is basically the same as RDF, but less strict in the sense that transactions do not have to commit in strict ticket order.

$$DIP \leftarrow ASAP + \text{safeness protection}$$

The disaster protection algorithm (DIP) is based on ASAP. The additional component of this algorithm is guaranteeing safeness by duplicating writes and queueing committed transactions to the other site.

$$DR \leftarrow DF + \text{address management}$$

The differential refresh algorithm (DR) is based on DF. The additional component of this algorithm is keeping track of the status of each address in the base table to optimize the amount of data transmitted from the primary to the copies.

$$CT \leftarrow ROWAA + \text{token info}$$

The copy token algorithm (CT) is based on ROWAA. The additional component of this algorithm is keeping track of the copy that holds the token.

The dependencies that we have just described can be represented as a graph, with nodes representing algorithms and arcs representing that one algorithm is used in the other. This is shown in Fig. 1, for the above-mentioned algorithms.

# 4 Recovery from site failures and network partitions

A site failure occurs when one node of the network goes down; when it comes up, it performs a *recovery*; in particular, copies are brought up-to-date by being over-written. The most critical failure that can affect replicated databases is a network partition; characterized by a failure of links connecting the nodes of the network so that two or more sets of nodes remain operational, but disconnected from each other. In this context, a violation to one-copy serializability may occur, because two or more copies representing the same value can be written by different applications, resulting in an inconsistent database. All the reviewed methods make the following assumptions on the detection of network failures:

- On recovery a site knows that it has gone down.
- Recovery is always started by the recovering site.
- If a method allows continuous execution in case of network partitions, then it is assumed that partitions can be detected. This happens either by:
    - Having a primary site periodically sending a message to each site;
    - Having each site monitoring the state of the network, discovering changes in the state (operational/failed) of a site or a communication link as soon as possible, and propagating new state information consistently to all sites in its partition.

In the table 3 we represent for each class of algorithms: the conditions against which it is resilient to site failure, what happens in case of network partition, and the actions to be executed upon recovering from a failure.

## 4.1 Description of methods

We now analyse the specific types of failures that the algorithms previously introduced in Sec. 3 are resilient to, and what these algorithms do for implementing recovery. Note that recovery is an aspect that is often not explicitly described in papers describing the algorithms, but more or less taken for granted.

**ROWA** It is not resilient to failures. When a failure occurs the system stops working, thereby not introducing inconsistencies.

**ROWAA** It only deals with site failures, in case of communication failures the database may get into an inconsistent state. If a site fails, the system continues to operate: updates are enforced at the operational sites. When a failed site comes up, data items are initialized again using up-to-date copies or by executing a transaction that writes into it. This brings the site to an up-to-date state.

**DOAC** It only deals with site failures, in case of communication failures the database may get into an inconsistent state. When a site goes down it is deleted from the directory of all the items of which it contains copies. Then, the system continues to operate. When the site comes up, for each data item, the system finds a directory and hence an available copy; this copy is read and its value is copied into the copy at the site that comes up.

|  | Update Strategy |  |  |  |  |  |
|---|---|---|---|---|---|---|
| Dealing with Failure | synchronous all | synchronous available | quorum-based | primary / backup | primary / secondary | independent |
| resilient to site failure | all sites up | at least one site up | read/write quorum up | primary site or backup site up | primary site up | at least one site up |
| in case of network partition | stops working | continues working but database may become inconsistent | the partition containing a quorum (if existing) continues to work | continues to work | the partition containing the primary continues to work | continues to work |
| Action at recovery | none | copy | none | repeat write actions | copy/repeat write actions | none |

Table 3: Failure behaviour of the various classes of algorithms

**QC and TQP** They deal both with site and communication failures. If a site goes down but the currently operational sites form a quorum, the system continues to operate. Moreover, no actions are necessary when the failed sites come up. If many sites fail and the operational sites do not constitute a quorum for the operation to be executed (read or write) then no transaction can be processed. Transactions can be queued and executed when the total number of operational sites constitute a quorum, or simply ignored.

**MW** It deals with both site and communication failures. When the system enters a reliable period of operation, all sites have to be brought up-to-date. This can be done by copying, for each data item, the value from an up-to-date copy.

**VP** It deals both with site and communication failures. When a failed site comes up the views have to be updated. Upon inclusion in a view, the site is brought up-to-date using the copies present in the view. Note that a view may lose its read (write) quorum when a site fails, in which case read (write) actions are no longer allowed on that particular view.

**MAR** It deals with both site and communication failures. As updates are executed independently, no special algorithm is necessary for recovery from failures. The same algorithm that is used to resolve conflicts in absence of failures can be used.

**IND** It deals with both site and communication failures. If a site fails, the system continues to operate at the other sites. If the system has to be brought fully up-to-date and a site is down, the operational copies may be brought to a consistent state. When a failed site comes up, nothing special has to be done; its updates will be incorporated the next time the system has to be brought up-to-date. Recovery from a partition is done in the same way as recovery from site failure.

**RDF & RBP** It is assumed that both primary and backup site are full-fledged distributed database systems. How primary and backup sites deal with failures, therefore, completely depends on their local strategy. The specific type of failure RDF and RBP were developed for are so-called disasters, i.e., a complete failure and stop of the primary site. When a disaster happens, the backup becomes primary. Before assuming the primary role, the backup is brought up-to-date, i.e., all committed transactions received at the backup site have to be executed before it takes over. When the primary site comes back, the copies in it are similarly brought up-to-date and the site regains its primary role.

**DIP** It deals with both site and communication failures. If a site fails, its workload is temporarily taken over by the other site. In case of disasters, i.e., a combination of site and communication failure, some transactions may be lost.

**ASAP** It deals with most site and all communication failures. If the primary copy goes down, no transaction can be processed. It is assumed that messages that are sent are eventually delivered; this means the network should, e.g., support stable message queues.

**QUC** It deals with most site and all communication failures. If the primary copy goes down, no transaction can be processed. If a non-primary copy fails, then the system continues to operate (on the primary copy and asynchronously on the other available copies). When the failed copy comes back the coherency conditions are checked. If they are satisfied nothing needs to be done. If the coherency conditions for the copy of an item are not satisfied, then the copy is brought up-to-date by copying in it the value contained in the primary copy. If the network goes down, the system continues to operate. Non-primary copies are informed of the failure and hence that some of the coherency conditions (which cannot be checked) may not be satisfied. When the network comes up, the coherency conditions are checked and the copies which do not satisfy the constraints are brought up-to-date.

**DF & DR** They deal with most site and all communication failures. If the primary copy goes down, no transaction can be processed. If a non-primary copy goes down, updates on it are postponed until it is available again.

**CT** In case of a partition, the group containing the token can continue working. Accessibility is lost if the site with the token or the communication medium fails. Upon recovery, sites are brought up-to-date using the information of the site holding the token.

## 4.2 Dependencies between algorithms

As was described in Sec. 3.2, algorithms sometimes depend on each other, or use the same basic strategy. This also holds for the recovery part of the algorithms. We describe it here in the same way as we did in Sec. 3.2

$$ROWAA = \text{initialize recovering copy}$$

The read one write all available algorithm (ROWAA) is a basic algorithm; upon recovery it initializes a copy again, either by copying the value from up-to-date copies, or by allowing a write operation on the copy that is recovering.

$$DOAC \leftarrow ROWAA + \text{update directories}$$

The directory oriented available copies algorithm (DOAC) is based on ROWAA. The additional component of this algorithm is updating the directories to include recovered copies.

$$MW \leftarrow ROWAA + \text{update system status}$$

The missing writes algorithm (MW) is based on ROWAA. The additional component of this algorithm is updating the system status when copies have been recovered.

$$VP \leftarrow ROWAA + \text{update views}$$

The virtual partition algorithm (VP) is based on ROWAA. The additional component of this algorithm is updating the views at each site, to include copies that have been recovered.

$$CT \leftarrow ROWAA + \text{update token information}$$

The copy token algorithm (CP) is based on ROWAA. The additional component of this algorithm is updating the token information when copies have been recovered.

$$RDF, RBP, DIP = \text{propagate transactions to former primary, reverse roles}$$

The remote duplicate database facility (RDF), remote backup procedure (RBP) and disaster protection (DIP) algorithm all follow a similar strategy when recovering from failure. Transactions that were executed while the primary site was down are propagated from the backup, installed at the former primary, and the former primary takes over the primary role again from the backup.

$$QUC, DR, DF = \text{if necessary, copy from available and restore conditions}$$

The actions at recovery time of the quasi-copy (QUC), differential file (DF) and differential refresh (DR) algorithms depend on the database state. If some of the conditions do not hold anymore—e.g., the difference in value between a copy and the primary is greater than allowed (QUC) or the copy is too old (DF, DR)—values have to be updated, e.g., by following the same strategy as ROWAA.

$$QC, TQP, ASAP, MAR, IND = \text{nothing special}$$

For the abovementioned algorithms no special action is required upon recovery. The algorithm keeps on functioning in the normal way; e.g., recovered sites can participate in a quorum, but their timestamp is used to note that they do not contain an up-to-date value.

## 5 Conclusions

This paper has presented a survey of the methods for updating replicated databases. The main contribution of this paper is to present the various methods in the context of structured taxonomy, which has accommodated very heterogeneous methods. Classes are presented through their general properties, such as their invariants; most detailed comparisons refer to methods belonging to the same class. Methods are reviewed both in their *normal* and *abnormal* behaviour, after a network partition.

We have shown that several methods presented in the literature, sometimes in independent papers with no cross-reference, are indeed very much related, for instance because they share the same basic technique. This survey should serve as a basis for choosing the method that is most suitable to a specific application, and as a guideline to researchers aiming at the development of new mechanisms.

## References

[1] D. Agrawal and A. El Abbadi "The tree quorum protocol: an efficient approach for managing replicated data," in *Proc. 16th Int. Conf. on VLDB*, Brisbane, Aug. 1990, pp. 243–254.

[2] D. Agrawal and A. El Abbadi, "Efficient techniques for replicated data management," *Proc. of the Workshop on Management of Replicated Data,* Houston, TX, Nov. 1990, pp. 48–52.

[3] R. Alonso, D. Barbara. H. Garcia Molina, S. Abad, "Quasi-copies: efficient data sharing for information retrieval systems," *Proc. of the Int. Conf. on Extending Data Base Technology, EDBT'88.*

[4] D. Barbara, H. Garcia-Molina, *The demarcation protocol: a technique for maintaining arithmetic constraints in distributed database systems,* CS-TR-320-91, Princeton University, April 1991.

[5] D. Barbara, H. Garcia-Molina, "The case for controlled inconsistency in replicated data," *Proc. of the Workshop on Management of Replicated Data*, Houston, TX, Nov. 1990.

[6] P.A. Bernstein, N. Goodman, "An algorithm for concurrency control and recovery in replicated distributed databases," *ACM TODS,* 9(4), Dec. 1984, pp. 596–615.

[7] P.A. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[8] S. Ceri, M.A.W. Houtsma, A.M. Keller, and P. Samarati, "A theory of independent updates and recovery," in preparation.

[9] S. Ceri and G. Pelegatti, *Distributed database systems*, McGraw-Hill.

[10] A. Chan, D. Skeen, *The reliability subsystem of a distributed database manager*, Tech. Rep. CCA-85-02, Computer Corporation of America, 1986.

[11] D.L. Eager, *Robust concurrency control in distributed databases*, Tech. Rep. CSRG #135, Computer System Research Group, University of Toronto, Oct. 1981.

[12] D.L. Eager, K.C. Sevcik, "Achieving robustness in distributed database systems," *ACM-TODS*, 8(3), Sept. 1983, pp. 354–381.

[13] A. El Abbadi, D. Skeen, F. Christian, "An efficient fault-tolerant protocol for replicated data management," *Proc. 4th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, Portland, OR, March 1985, pp. 215–228.

[14] A. El Abbadi, S. Toueg, "Availability in partitioned replicated databases," *Proc 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, Cambridge, MA, March 1986, pp. 240–251.

[15] H. Garcia-Molian and K. Salem "Sagas" *Proc. ACM Sigmod*, 1987, pp. 249–259

[16] D.K. Gifford, "Weighted voting for replicated data," *Proc. 7th ACM-SIGOPS Symp. on Operating Systems Principles*, Pacific Grove, CA, Dec. 1979, pp. 150–159.

[17] N. Goodman, D. Skeen, A. Chan, U. Dayal, S. Fox, D. Ries, "A recovery algorithm for a distributed database system," *Proc. 2nd ACM SIGACT-SIGMOD, Symp. Database System* Atlanta, GA, March 1983, pp. 8–15.

[18] J.N. Gray, M. Anderton, "Distributed computer systems: four case studies", *Proc. of the IEEE*, Vol. 75, No. 5, May 1987.

[19] R.P. King, N. Halim, H. Garcia-Molina, C.A. Polyzois, "Management of a remote backup copy for disaster recovery", *ACM-TODS*, 16(2), June 1991, pp. 338–368.

[20] A. Kumar and A. Segev "Optimizing voting-type algorithms for replicated data," in *Advances in Database Technology–EDBT'88*, J.W. Schmidt, S. Ceri, and M. Missikoff (Eds.), LNCS **303**, 1988, pp. 428–442.

[21] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms, "A snapshot differential refresh algorithm," *Proc. ACM Sigmod*, 1986, pp. 53–60.

[22] T. Minoura and G. Wiederhold. "Resilient extended true-copy token scheme for a distributed database," *IEEE TSE*, Vol. 8, No. 3, pp. 173–189.

[23] C. Pu, A. Leff, "Replica control in distributed systems: an asynchronous approach,", *Proc. ACM-SIGMOD'91*, Denver, CO, May 1991.

[24] C. Pu and A. Leff, "Replica control in distributed systems: an asynchronous approach," Technical report No. CUCS-053-90, Columbia University, Jan. 1990.

[25] C. Pu and A. Leff, "Epsilon-Serializability," Technical report No. CUCS-054-90, Columbia University, Jan. 1990.

[26] A. Reuter and H. Wächter, "The contract model," *IEEE Database Engineering bulletin* Vol. 14, No. 1, March 1991.

[27] S.K. Sarin, C.W. Kaufman, and J.E. Somers, "Using history information to process delayed database updates," *Proc. 12th Int. Conf. on Very Large Data Bases*, Kyoto, Japan, 1986.

[28] D.G. Severance, G. Lohman, "Differential files: their application to the maintenance of large databases," *ACM-TODS,* 1(3), Sept. 1976.

[29] Tandem Computers. *Remote Duplicate Database Facility (RDF) System Management Manual,* March 1987.