Set-Serializability: A Formal Theory for Partitioned Data (Extended abstract)*

G. Alonso A. El Abbadi
Department of Computer Science
University of California
Santa Barbara, CA 93106

1 Introduction

High transaction throughput in a distributed database system can be achieved by increasing the autonomy of each site. This can be achieved by making more data available locally. One way to ensure this is to partition data items in several subsets and allocate one subset to each site. As long as a transaction can execute locally using the data items allocated to a single site, it will not need to access data at other sites. An example of partitioned data is a ticket reservation system in which instead of locating all the tickets for an event in one central site, they are divided into several lots and each lot is allocated to a site. In that way, sites can proceed locally without communication with the central site or with other sites, speeding up the processing of transactions [BGM92, GT92, SS90]. To guarantee correct executions, constraints must be imposed on the operations that can be executed in parallel over the partitioned data items. These constraints can be represented as *distributed assertions* and a mechanism must be provided to maintain them across the distributed system. The fact that the data has been partitioned may lead to situations in which an operation can not be performed over a partition of the data but it could be done if the data was not partitioned. As a consequence, Data Partitioning may reduce the overall availability of data, which is clearly undesirable. To solve this problem, a protocol has to be developed to dynamically redefine a partition of data in such a way that the data is available at the site where it is needed.

To our knowledge, this problem was first formalized by Carvalho and Roucairol [CR82]. They introduce the notion of a *distributed assertion*, key in partitioned data, and present an algorithm for maintaining these assertions on a distributed system. Barbará and García-Molina [BGM92] used these ideas and some previous work on Data Caching [ABGM90] to develop the Demarcation Protocol. To date, this protocol is the most complete attempt to solve Data Partitioning. However, it restricts the types of constraints allowed and issues like recovery and integration with non-partitioned data are not covered. Soparkar and Silberchatz [SS90] have proposed the Data Value Partitioning protocol (DvP) as a way to cope with network partition failures and avoid the blocking behavior commit protocols. The overall goal is to deal with network partition failures and not to improve transaction throughput. Golubchik and Thomasian [GT92] present a simplified analysis

*This research was partially supported by the NSF under grant number IRI-9117904.

of the DvP protocol in terms of the policies used for initial allocation and data redistribution. Hsu and Chan [HC86] have developed a Partitioned Two-Phase Locking mechanism that takes advantage of a hierarchically decomposed database. These same ideas have also been applied to enhance concurrency in replicated databases using the Escrow Transactional Method [KB92, KS88, Kum90, O'N86].

In this paper we introduce a general theory for distributed assertions and a protocol for maintaining the validity of these assertions over a distributed database. The operations of the protocol are executed as an atomic transaction, which allows an analysis of the problem using an extension of serializability theory. *Set-serializability* is defined as the correctness criteria for these type of databases. It has the advantage of allowing standard serializable histories and histories that, due to the execution of the protocol for partitioned data, contain cyclic dependencies among transactions. Such transactions must cooperate to maintain the correctness of the database and the validity f the distributed assertions. Thus, set-serializability guarantees the correctness of the database both from the serializability point of view and from the assertions point of view.

Our approach is novel and makes several significant contributions. Our theory models both partitioned and non-partitioned data objects and we propose solutions that integrate both concurrency control and recovery. In particular, we propose the notion of *transactions sets* and *set history*, that allow a set of independent transactions to cooperate to maintain the database consistency and to terminate conditionally to each other. Thus, the set of executions allowed by set-serializability is a superset of of that allowed by serializability. Previous approaches [BGM92, SS90] considered a restricted set of constraints, ignored the issue of recovery and did not address the problem of integrating partitioned data with the standard non-partitioned data. Our approach, therefore provides an uniform and comprehensive framework for partitioned data in distributed database systems

2 On Assertions

Consider a set of variables, x, y, z, which take values from the sets A_x, A_y and A_z respectively. Let $\mathcal{F}(x, y, z)$ be a predicate over those variables. For simplicity we consider only three variables, the notation can be easily generalized to predicates of any number of variables. Given the actual values of x, y and z, their *configuration sets*, $\mathcal{S}_x$, $\mathcal{S}_y$ and $\mathcal{S}_z$, are defined as subsets of A_x, A_y and A_z respectively, such that $x \in \mathcal{S}_x \wedge y \in \mathcal{S}_y \wedge z \in \mathcal{S}_z \Rightarrow \mathcal{F}(x, y, z)$. We define a *valid configuration*, $\mathcal{C}$, of x, y, z with respect to a predicate $\mathcal{F}$ as $\mathcal{C} = \{\mathcal{S}_x, \mathcal{S}_y, \mathcal{S}_z | x \in \mathcal{S}_x \wedge y \in \mathcal{S}_y \wedge z \in \mathcal{S}_z \Rightarrow \mathcal{F}(x, y, z)\}$

A valid configuration expresses the fact that given the actual values of x, y and z there exist a range of values for each of them such that if the variables are always within these ranges then the predicate is always true. There might be several valid configurations for the same value of x, y and z. The set of valid configurations given the values of x, y and z is called an *equivalent set*, $\mathcal{E}_{x,y,z} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3 ... | \mathcal{C}_i \Rightarrow \mathcal{F}(x, y, z)\}$ Two configurations are *equivalent* if they belong to the same equivalent set.

An operation, $g(x) = x'$, is *safe* if $g(x)$ when executed in a valid configuration $\mathcal{C} = \{\mathcal{S}_x, \mathcal{S}_y, \mathcal{S}_z\}$ produces a result x' such that $x' \in \mathcal{S}_x$. An operation $g(x)$ is *unsafe but solvable* if $g(x)$ when executed in a valid configuration $\mathcal{C} = \{\mathcal{S}_x, \mathcal{S}_y, \mathcal{S}_z\}$, produces a result x' such that $x' \notin \mathcal{S}_x$ but

$\exists \mathcal{C}' = \{\mathcal{S}'_x, \mathcal{S}'_y, \mathcal{S}'_z\}$ that is equivalent to $\mathcal{C}$ in which $g(x)$ is safe. An operation is *unsolvable* if it is not safe nor unsafe but solvable.

Safe operations can be performed immediately since they guarantee that the result will be within the current configuration. Unsafe but solvable operations, on the other hand, can not be directly executed without violating the current configuration. However, it may be possible to find an equivalent configuration in which the operation can be performed safely. Given a configuration $\mathcal{C}$ and an operation $g(x)$ that is unsafe but solvable on $\mathcal{C}$ there might be several configurations $\mathcal{C}'$, all equivalent to $\mathcal{C}$, where $g(x)$ is safe. In such cases, the current configuration $\mathcal{C}$ must be transformed to a new equivalent configuration $\mathcal{C}'$. We will denote such a transformation as a *reconfiguration*, $\mathcal{R}(\mathcal{C}) = \mathcal{C}'$. Any configuration in an equivalent set can be transformed to any other configuration in that set.

It is useful to consider a reconfiguration as a sequence of several smaller reconfigurations. We say that given a configuration $\mathcal{C}$ and an unsafe but solvable operation $g(x)$ in $\mathcal{C}$, a reconfiguration, $\mathcal{R}(\mathcal{C})$, is *divisible* if

1. $\mathcal{R}(\mathcal{C}) = \mathcal{C}'$ with $g(x)$ safe on $\mathcal{C}'$ and
2. $\exists \mathcal{R}_i, i = 1..n | \mathcal{R}_1(\mathcal{C}) = \mathcal{C}_1, \mathcal{R}_2(\mathcal{C}_1) = \mathcal{C}_2, \; ..\mathcal{R}_n(\mathcal{C}_{n-1}) = \mathcal{C}'$ and all $\mathcal{C}_i$ are equivalent valid configurations.

If, in addition, $g(x)$ is safe in all $\mathcal{C}_i$ then the reconfiguration is said to be *safely divisible*. Not all divisible reconfigurations are safely divisible. In a safely divisible reconfiguration one single step, one R_i, suffices to make $g(x)$ safe. In a divisible reconfiguration that is not safely divisible there is no guarantee that $g(x)$ is safe until the entire sequence of reconfigurations has been performed.

3 Reconfiguration Transactions

The database consists of a set of partitioned and non-partitioned data objects distributed among different sites. Sites communicate by message passing. The underlying communication system may lose messages. We assume that the database is initially in a valid configuration. To provide support for partitioned data, we need a protocol that guarantees that if the database is initially in a valid configuration, then only safe operations will be executed. Assuming that the operation triggering the reconfiguration is $g(x)$, the reconfiguration process first reads $\mathcal{S}_x$, discovers that $g(x)$ is not a safe operation, reads $\mathcal{S}_y$, y, $\mathcal{S}_z$ and z to find out if $g(x)$ is a solvable operation and if it is, a new configuration $\mathcal{C}'$ is created. This process is carried out in two parts that constitute the *Reconfiguration Protocol* (conceptually similar to the Demarcation Protocol [BGM92]). The protocol first checks whether an update operation is safe. If this is the case, the operation is performed and the protocol (the reconfiguration) terminates. If this is not the case the site sends reconfiguration requests to all other sites. Upon receiving such a request a site executes the second part of the protocol. A site may ignore the request or perform the changes and send an acknowledgement message back to the requesting site. When the requesting site receives this acknowledgement, the reconfiguration takes place, the update is performed and the protocol terminates. Thus a reconfiguration can be seen as a series of reconfiguration transactions: $t_R = t_{R0} t_{R1} t_{R2}$ where $t_{R0} = r[S_x] r[x] w[S_x] w[S_x]$,

$t_{R1} = r[S_y]r[y]w[S_y]$ and $t_{R2} = r[S_z]r[z]w[S_z]$. t_{R0} is executed at the site where the update is generated. Each write operation of t_{R0} corresponds to an acknowledgement message received at the local site. If no acknowledge arrives then t_{R0} only has the read operations and the the update will be aborted. Each of t_{R1} and t_{R2} are local transactions that are executed at the remote sites. The protocol uses only safely divisible reconfigurations, i.e. one response, either from t_{R1} or from t_{R2}, (and the corresponding update of the configuration set by t_{R0}) is enough to guarantee the safety of the update.

Assume the database is in configuration $\mathcal{C} = \{\mathcal{S}_{x1}, \mathcal{S}_{x2} ... \mathcal{S}_{xn} | x_1 \in \mathcal{S}_{x1} \wedge x_2 \in \mathcal{S}_{x2} \wedge ... \wedge x_n \in \mathcal{S}_{xn}\}$ and the predicate to be enforced is $\mathcal{F}(x_1, x_2, ..., x_n)$. When an operation, $g(x_i) = x_i'$, tries to update a partitioned data item, the local site, referred to as the *Coordinator*, sends reconfigurations requests to other sites, referred to as *Participants*. The execution of the following protocol corresponds to t_{R0}:

```
RECONFIGURATION (x'_i)
begin
    if (x'_i ∈ S_xi) then Perform update x'_i
                     else for all x_j and j ≠ i
                     ∀Participants, SEND(Participant, Request(x'))
    Receive(*, Ack(x'))
    If no Ack is received then Abort Update
                              else
                              Perform updates on S_x
                              Perform update x'_i
end RECONFIGURATION.
```

When the protocol returns, the operation has been performed or aborted. This piece of code is executed as a critical section, i.e. simultaneous reconfigurations over the same object are not allowed. If a response to a request arrives after the corresponding execution of the protocol has terminated, the changes on $\mathcal{S}_x$ are performed anyway. Each site has a request server, that handles these delayed responses as well as the code for the Participants, corresponding to the t_{Ri}'s, $i > 0$. This server is as follows:

```
RECONFIGURATION SERVER
begin
    Receive( Site, Message(x'))
    if Message = Request then (* Message is a request from the Coordinator *)
                              Compute requested change from S_xj to S'_xj
                              if x_j ∈ S'_xj then Perform updates on S_xj
                                              SEND(Coordinator, Ack(x'))
                                              else discard request
                         else (* Message is an Acknowledgement *)
                              Perform updates on S_x
end RECONFIGURATION SERVER.
```

Any particular implementation of the protocol will have several policies to decide how many sites to contact (acquisition policy), the amount of the request (amount-borrowed policy) and other details [BGM92, GT92, Kum90]. We will not consider these details here.

From the structure of a reconfiguration, we know that first the read operations of t_{R0} must be executed, then t_{R1} and t_{R2} are triggered, and after the commitment of each of them a write operation from t_{R0} may take place. Given a valid configuration $\mathcal{C}$ and a divisible reconfiguration $\mathcal{R}$, any partial execution of the corresponding reconfiguration results in a valid configuration. Furthermore, if $\mathcal{R}$ is safely divisible, any partial execution of t_{R0} that includes a write operation is enough to make $g(x)$ safe. Thus, it is enough that one of the two transactions, t_{R1} or t_{R2}, commits. It can be said that the changes introduced by t_{R0} can be committed only if t_{R1} or t_{R2} commit. More formally, a *conditional-commit* relation between two transactions t_i and t_j is defined as follows: t_i *conditionally-commits* t_j if the abortion of t_i implies the abortion of t_j. This definition includes the traditional recoverability relationship induced by *reads-from* relations [BHG87]. It also includes any other dependency among transactions that may lead to cascading aborts. In the previous example, t_{R1} or t_{R2} conditionally-commit t_{R0}

4 A motivating example of cyclic dependencies

The Reconfiguration Protocol allows transactions to communicate outside the realm of the database. Hence, this communication is not subject to the concurrency control rules imposed on the execution. As a consequence, the Reconfiguration Protocol may create cyclic dependencies among transactions that are not captured by concurrency control techniques such as serializability. To integrate partitioned data within a database that also contains non-partitioned data, we need to capture these relationships.

Consider, for example, a distributed system with two types of resources, X and Y and two sites, A and B. Each site has a certain amount of each resource. X_a will represent the amount of resource X residing at site A, and so forth. Assume that for both types of resources it is necessary to guarantee that at least 100 units are available, i.e. $X > 100$ and $Y > 100$. Since the resources are partitioned between sites A and B, these constraints become distributed assertions of the form $X_a + X_b > 100$ and $Y_a + Y_b > 100$. Assume that the initial values are $X_a = 50$, $\mathcal{S}_{Xa} = \{X_a | X_a \geq 50\}$, $X_b = 55$, $\mathcal{S}_{Xb} = \{X_b | X_b \geq 50\}$, $Y_a = 55$, $\mathcal{S}_{Ya} = \{Y_a | Y_a \geq 50\}$, $Y_b = 50$, $\mathcal{S}_{Yb} = \{Y_b | Y_b \geq 50\}$. There are two processes, P_a at site A and P_b at site B. P_a holds a number of units of resource of type X while P_b holds a number of units of resources of type Y. Furthermore, assume that both processes must maintain their total number of allocated units constant (regardless of the type). Hence, to acquire and release resources, each executes a transaction that must be executed in its entirety or not at all.

Now suppose that P_a wants to exchange its X resources for Y resources and P_b wants to exchange its Y resources for X resources. Given the initial configuration, if both processes need to exchange 5 units of resources allocated to them, the transactions can be executed locally. However, if they required to exchange 10 allocated units, the execution of transactions at each site is as follows:

$$Site_a = w_1[X_a + 10]w_1[Y_a - 10]$$

$$Site_b = w_2[Y_b + 10]w_2[X_b - 10]$$

In both transactions, the last operation (acquiring 10 units of resource) can not be performed without a reconfiguration. The execution including these reconfigurations may be as follows:

$$Site_a = w_1[X_a + 10] t^1_{R0} t^2_{R1} w_1[Y_a - 10]$$

$$Site_b = w_2[Y_b + 10] t^2_{R0} t^1_{R1} w_2[X_b - 10]$$

Serializability theory states that for this history to be correct, there must be an equivalent serial order of those transactions. However, there is no equivalent serial history for this example, e.g if t_1 is executed first $w_1[Y_a - 10]$ becomes an unsolvable operation, the same happens if t_2 is executed first. As a matter of fact, there is no way of performing this "trading" of resources with any correctness criteria based on standard serializability since it creates a cyclic dependency. Note that the operations issued by the transactions do not conflict at all, the non-serializability is due to the execution of the reconfigurations. Hence, according to standard serializability this execution is not correct. But if the two transactions are analyzed as a whole, the execution is perfectly valid. In a sense, the two transactions have cooperated and became a single entity that must be executed in its entirety or not at all. Before the execution of the two transactions the database is in a consistent state, after the execution of *both* transactions the database is left in a consistent state. Furthermore, the transactions work with valid data. Standard serializability can not capture the meaning of this execution because transactions are seen as completely isolated entities. With partitioned data this isolation is relaxed to the point that, as in the example, transactions may interleave in such a way that either both commit or both abort and in either case the final result is correct. It can be said that for t_1 to be able to commit, t_2 must commit and vice versa. Hence, if t_2 is guaranteed to commit, we could commit t_1, and vice versa. The solution is then to commit or abort both transactions. This concept can be formally captured by the same notion of *conditional-commit* defined for the reconfigurations. We have the following chain of relations: t^1_{R1} conditionally-commits t^1_{R0} and t^1_{R0} conditionally-commits t_1. The same can be said of t_2 and t^2_{R0} and t^2_{R1}. Furthermore, since t^1_{R1} reads uncommitted data from t_2 (the value of Y_b after t_2's increase by 10), t_2 conditionally-commits t^1_{R1}. By transitivity t_2 conditionally-commits t_1. Using the same type of reasoning it can be stated that t_1 conditionally commits t_2.

This sort of *cooperative* execution is not allowed in any of the existing protocols for partitioned data. The Demarcation Protocol [BGM92] will always abort both transactions since in this protocol a transaction that requires a reconfiguration is always aborted. Even with this strong restriction on the execution of the transactions and reconfigurations, the Demarcation Protocol may lead to executions that are not recoverable. The DvP protocol [SS90], solves this situations by forcing a transaction to predeclare its *access set* and lock all objects in it before any operation is done. Since Strict-2PL is used, the execution of the example can not occur. Both solutions reduce the overall throughput.

Standard serializability provides the isolation property by guaranteeing an execution is equivalent to a serial execution. Since each transaction preserves consistency, the serial sequence of transactions yields a consistent database. Any cyclic dependency among transactions is rejected by definition, since there is no equivalent serial order. As the example shows, with partitioned data there are cyclic dependencies that are correct as long as the transactions involved in the cycle are committed as a group. To allow this type of executions we need to modify serializability in such a way that instead of imposing an ordering on transactions, the order is imposed on sets of transactions. Within each set we can then have cyclic dependencies due to reconfigurations. Now the units of consistency are no longer the transactions but sets of cooperative transactions.

5 Set-Serializability

We will consider two type of transactions, normal transactions and reconfiguration transactions. A normal transaction, or transaction, t_i is a partial order, $<_i$, where

1. $t_i \subseteq \{r_i[x],\ w_i[x],\ x \textit{ is a data item}\} \bigcup \{a_i, c_i\}$.
2. $a_i \in t_i$ iff $c_i \notin t_i$;
3. If t is c_i or a_i, whichever in t_i, for any other operation $p \in t_i,\ p <_i t$.
4. if $r_i[x],\ w_i[x] \in t_i$, then either $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$.

A safely divisible reconfiguration, R_i, is a set of reconfiguration transactions with a conditional-commit relation $<<_i$ where

1. $R_i \subseteq \{t_i^{Rj},\ j \geq 0\}$.
2. Each t_i^{Rj} is a total order, $<_{Rj}$, where
   - $t_i^{Rj} \subseteq \{r[S_x], w[S_x]\} \bigcup \{r[x]\} \bigcup \{a_{Rj}, c_{Rj}\}$, where S_x is a configuration set and x is the corresponding partitioned data item.
   - $\forall t_i^{Rj},\ a_{Rj} \in t_i^{Rj}$ iff $c_{Rj} \notin t_i^{Rj}$.
   - if q is c_{Rj} or a_{Rj}, whichever in t_i^{Rj}, for any other operation $p \in t_i^{Rj},\ p <_{Rj} q$.
3. w_i^{R1} *or* w_i^{R2}*...or* $w_i^{Rk} <<_i w_i^{R0}$.

An extended history, EH, is a partial order, $<_{EH}$, and a conditional-commit relation, $<<_{EH}$, over a set of transactions and their reconfigurations, such that:

1. $EH = (\bigcup_{i=1}^{n} t_i) \cup (\bigcup_{j=1}^{m} R_j)$.
2. $<_{EH} \supseteq \bigcup_{i=1}^{n} <_i$, $<_{EH} \supseteq \bigcup_{j=1}^{m} <_{Rj}$ and $<<_{EH} \supseteq \bigcup_{j=1}^{m} <<_{Rj}$.
3. For any update, $w_i[x]$, that is unsafe in configuration $\mathcal{C}$ there must be an operation $w_{RO}^{i}[S_x]$, that changes to configuration $\mathcal{C}'$, such that $w_{RO}^{i}[S_x] <<_{EH} w_i[x]$ and $w_i[x]$ is safe on $\mathcal{C}'$.
4. If for a transaction t_i $w_i[x] \in t_i$, and there is a reconfiguration t_{Rk}^{j} such that $r_{Rk}^{j}[x] \in t_{Rk}^{j}$ and t_{Rk}^{j} reads-from t_i, then $w_i[x] <<_{EH} r_{Rk}^{j}[x]$.

Requirement 3 implies that any unsafe update requires a reconfiguration. Requirement 4 relates, through a conditional-commit, a transaction and a reconfiguration when the reconfiguration happens to read data written by the transaction.

The concept of serialization graph can be applied to extended histories. Given an extended history, EH, its corresponding serialization graph, $SG(EH)$ is defined as follows:

1. Nodes are transactions and reconfiguration transactions.
2. For any pair of conflicting operations $op_i \in t_i$ and $op_j \in t_j$, such that t_i and t_j are both transactions or both reconfiguration transactions and $op_i <_{EH} op_j$, then the serialization graph contains a *T-edge*: $t_i \rightarrow t_j$.
3. For any pair of operations $op_i \in t_i$, $op_j \in t_j$, such that $op_i <<_{EH} op_j$, the graph contains a *C-edge*: $t_i \Rightarrow t_j$.

T-edges capture conflicts between transactions, while C-edges capture conditional-commit dependencies. The correctness of an extended history will be determined by examining its serialization graph. As in standard serializability theory, the existence of cycles in the graph indicates a particular type of execution that requires special attention. We will consider *T-cycles* (containing at least one T-edge) and *C-cycles* (containing only C-edges).

5.1 Correctness

A history in which all transactions are committed is known as a *complete history*. To define correctness we will us the notion of *committed projection* of a history as the subset of committed transactions of a given history [Had88]. To ensure it contains all the transactions and reconfigurations (and rule out as a valid committed projection one in which a transaction is present but not its reconfigurations or vice versa), a valid committed projection must contain all transactions and reconfigurations that appear on the left hand side of a conditional-commit relation. From now on we will work only with committed projections. In databases, the most common correctness criteria for concurrency control is serializability: a history is correct when it is conflict equivalent to a serial history. We can not use this definition of correctness because conditional-commit relations may require that two transactions be executed concurrently as oppose to serially. To be able to relax the notion of serializability to cope with this new requirement, the notion of transaction set is introduced.

A *transaction set* is a set of transaction operations belonging to the same extended history. An extended history will be divided into several transaction sets. A *set history*, SH, is a set of transaction sets, TS, and an ordering relation $<_{SH}$ over the transaction sets. A set history, SH, can be derived from an extended history, EH, using the following rules:

1. They are defined over the same set of transactions and have the same operations.
2. $\forall op_i \in EH, \exists$ *a transaction set* $TS_j \in SH$ such that $op_i \in TS_j$.
3. if $op_i <_{EH} op_j$ and op_i and op_j conflict, and $op_i \in TS_i$ and $op_j \in TS_j$, then $TS_i <_{SH} TS_j$.

4. for all pairs $op_i <<_{EH} op_j$, such that $op_i \in TS_i$ and $op_j \in TS_j$, either $TS_i <_{SH} TS_j$ or $TS_i = TS_j$.

A set history, SH_s, is *serial* if each transaction belongs to only one set and for every two transactions sets, TS_i and TS_j, either all operations in TS_i appear before all operations in TS_j or vice versa. Using these ideas we can define *set-serializability* as our correctness criteria: an extended history, EH, is *set-serializable* if it can be converted into a serial set history. Note that if there are no reconfigurations, set-serializability is the same as serializability. The serialization graph of an extended history can be used to determine whether a given extended history is set-serializable or not. This is formalized in the following theorem (proofs for this and the following theorems and lemmas can be found in the appendix):

Theorem 1. *An extended history, EH, is set-serializable if and only if its serialization graph, $SG(EH)$, contains no T-cycles.*

Set-serializability does not contain any reference to the predicates that are being enforced by the reconfigurations and the notion of safe operation. Since each conflicting transaction is in a different set, consistency is preserved in the sense that conflicting transactions are executed in isolation. However, within a set transactions may be interleaved. Thus, to prove that a set-serializable history preserves the consistency of the database we need to show that a set preserves this consistency. If this is the case, then a serial execution of sets results in a consistent database if the original database was consistent.

Lemma 1: *Starting with a valid configuration and executed in its entirety, a reconfiguration always leaves the database in a valid configuration.*

Lemma 2: *Starting with a valid configuration, any possible execution of a safely divisible reconfiguration leaves the database in a valid configuration.*

Lemma 3: *In an extended history only safe updates are executed*

Theorem 2. *A transaction set always transforms the database from a consistent state to another consistent state*

From Theorem 1 and 2 we conclude that set-serializability guarantees correctness in a database with partitioned data items. Lemma 2 is particularly important when the communication media is not reliable and messages can be lost. Since reconfigurations are carried out through messages exchanges, a lost message will prevent a reconfiguration from being executed entirely. Lemma 2 states that even in this case, the resulting state of the database is consistent.

6 Outline of a protocol for databases with partitioned and non-partitioned data

In this section we provide a high level description of a protocol for both partitioned and non-partitioned data. This protocol ensures that the resulting execution is set-serializable, encompassing both concurrency control and recovery of such database. There are two separated schedulers, one for transactions (called Transaction Concurrency Control, or TCC) and one for reconfigurations (called Configuration Concurrency Control, or CCC). These two schedulers use Strict-2PL to guarantee the serializability of transactions and reconfigurations respectively. Transactions access configuration sets without locking them. Conversely, reconfigurations access data-items without locking them. This effectively divides the database in two, data items and configuration sets.

To be able to enforce conditional-commit dependencies, we first need to keep track of them. The conditional-commit relations are stored in a table. For simplicity of exposition we will assume that this table is accessible to all sites and the commitment of a transaction is known to all sites but it could be distributed among the different sites. This table contains entries of the form $t_i \Rightarrow t_j$, meaning t_i conditionally-commits t_j. In other words, t_j can not commit unless t_i commits, if t_i aborts, so does t_j. Hence, we can talk about the left hand side of a commit-agreement as the requirement to commit and the right hand side as the transaction waiting to commit.

When a transaction wants to commit, the TCC checks the table to see if it is waiting from some other transaction. If it is in the right hand side of a commit-agreement relation then the transaction is placed on a list of transactions waiting to commit. If it does not appear in the right hand side of any commit-agreement entry, the transaction can commit. Upon commitment, all entries in the table with the transaction in the left hand side are removed. For each entry removed, the list of waiting transactions is checked to see if there is some transaction that can now commit. When a transaction aborts, the same search through the table takes place, but now all transactions that depend on the aborted transaction will be aborted. This takes care of all non-cyclic conditional-commit dependencies. If there is a cycle in these dependencies, all transactions in the cycle will end up in the list of waiting transactions without any possibility of getting out of there. Once this cycle is detected, and provided that none of those transactions are waiting for transactions not in the cycle, all of them can be committed. If one of them aborts, all of them will have to be aborted. This concurrency control protocol guarantees set-serializable histories (see Theorem 3 in the Appendix).

7 Conclusions

Set-serializability has been proposed as a correctness criteria for databases with Partitioned Data. It has been developed from an analysis of the requirements and characteristics of the distributed assertions necessary to maintain the consistency of the partitioned data over the distributed database. The fact that a set-serializable extended history is serializable for conflicting transactions guarantees the traditional notion of correctness. The use of the Reconfiguration Protocol ensures the correctness of the database from the point of view of the assertions. Set-serializability uses the Reconfiguration Protocol to ensure the correctness of the database both in terms of traditional serializability and in terms of distributed assertions over partitioned data. Furthermore, most of the previous work on the subject has ignored the issue of recovery. Our concurrency control protocol solves this problem by committing or aborting sets of cooperative transactions. The structure of the safely divisible reconfigurations guarantees that at any point of the execution of a reconfiguration the validity of the predicates is preserved, making the overall criteria resilient to failures (both site and communication failures). Finally the notion of serializing sets of transactions instead of individual transactions allows a greater concurrency by grouping together transactions involved in a conditional-commit cycle. This enhances the transaction throughput of the overall distributed system. Our protocol is a generalization of the Demarcation Protocol [BGM92] and most of the previous work [BGM92, GT92, SS90] on partitioned data can be easily mapped to and analyzed with our model.

References

[ABGM90] R. Alonso, D. Barbara, and H. Garcia Molina. Data Caching isues in an information retrieval system. *ACM Transactions on Database Systems*, 15(3):359–384, September 1990.

[BGM92] D. Barbara and H. Garcia Molina. The Demarcation Protocol: A technique for maintaining linear arithmetic constraints in distributed database systems. *3rd International Conference on Extending Database Technology, Vienna, Austria*, pages 373–387, March 1992.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[CR82] O.S.F. Carvalho and G. Roucairol. On the distribution of an assertion. *Proceedings of the ACM-SIGOPS Symposium on Principles of Distributed Computing, Ottawa*, pages 121–131, 1982.

[GT92] L. Golubchik and A. Thomasian. Token allocation in distributed systems. *IEEE 12th International Conference on Distributed Computing Systems, Yokohama, Japan*, 12:64–71, June 1992.

[Had88] V. Hadzilacos. A Theory of Reliability in Database Systems. *Journal of the ACM*, 35(1):121–145, January 1988.

[HC86] M. Hsu and A. Chan. Partitioned Two-Phase Locking. *ACM Transactions on Database Systems*, 11(4):431–446, December 1986.

[KB92] N. Krishnakumar and A.J. Bernstein. High throuhput Escrow algorithms for replicated databases. *Proceedings of the 18th VLDB Conference, Vancouver, Canada*, 18:175–186, August 1992.

[KS88] A. Kumar and M. Stonebraker. Semantics based transaction management techniques for replicated data. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Chicago*, pages 117–125, 1988.

[Kum90] A. Kumar. An Analysis of Borrowing Policies for Escrow Transactions in a Replicated Data Environment. *Proceedings of the IEEE Conference on Data Engineering*, pages 446–454, 1990.

[O'N86] P. O'Neil. The Escrow Transactional Method. *ACM Transactions on Database Systems*, 11(4):405–430, December 1986.

[SS90] N. Soparkar and A. Silberschatz. Data-Value Partitioning and Virtual Messages. *Proceedings of the Conference on Principles of Database Systems*, pages 357–367, 1990.

8 Appendix

The following are outlines of the proofs of the different Theorems and Lemmas mentioned in the paper. These proofs will be detailed in the paper.

Theorem 1. *An extended history, EH, is set-serializable if and only if its serialization graph, $SG(EH)$, contains no T-cycles.*

Proof: (If): Assume $SG(H)$ contains no T-cycles. Without loss of generality, assume $t_1, t_2, ...t_n$ are the nodes of $SG(H)$. For any C-cycle in $SG(EH)$, merge together the nodes involved in the cycle. Now the nodes are $t_1, t_2, ...t_m$ with $m \leq n$. Remove the label (T-edge or C-edge) from all edges. The result is an directed graph. It can be easily show that this resulting graph has no cycles. It can not have a cycle that appeared from a C-cycle in $SG(EH)$ because they have been collapsed in one single node. It can not have a cycle that appeared because of a T-cycle in $SG(EH)$ because $SG(EH)$ does not contain such cycles. Thus, the result is a directed acyclic graph. This graph can be topologically sorted, assigning each node to a transaction set. This sorting is a serial set history. Now we have to prove that this topological sort is a set history constructed according to the rules and that it is serial. Since $SG(EH)$ contains the transactions and reconfiguration transactions of EH, and the final sets contain all these nodes, then the set history and EH are defined over the same set of transactions and have the same operations. For the same reason, each operation is assigned to a transaction set, that of the transaction it belongs to. To show that transaction sets are ordered in the same way that the conflicts between the operations they contain suppose op_i, op_j conflict and $op_i <_{EH} op_j$, then in $SG(EH)$ $t_i \rightarrow t_j$. These two transaction can not be involved in a C-cycle (if they are, there is also a T-cycle), therefore they are not merged together. Therefore, in the topological sort they will be assigned to transactions sets TS_i and TS_j such that $TS_i <_{SH} TS_j$. The last thing to prove that this is a correct transformation from EH to a set history, is to show that for all pairs $op_i <<_{EH} op_j$, such that $op_i \in TS_i$ and $op_j \in TS_j$, either $TS_i <_{SH} TS_j$ or $TS_i = TS_j$. In $SG(EH)$ this relation is represented as $t_i \Rightarrow t_j$. If this edge belongs to a C-cycle, t_i and t_j will be merged together and will end up in the same transaction set, which is correct. If they are not involved in a cycle there are two cases to consider. Either $t_i \rightarrow t_j$ or they are not related by T-edges (any other combination creates a T-cycle). In both cases after the removal of labels from the edges they will be assigned to transactions sets such that $TS_i <_{SH} TS_j$. This shows that the topological sort is a set history. Now we need to show that the resulting set history is serial. First we need to show that there is no transaction that belongs to two different transaction sets. The original $SG(EH)$ has individual transactions (or reconfigurations) as nodes. The first step merges together several transactions. But in no case a transaction is divided in two. Then, in the topological sort of the resulting graph, a transaction will be assigned to only one set, either by itself, or as part of a group that has been merged. Second, we need to show that there no cycles in the $<_{SH}$ ordering. By assumption, $SG(EH)$ has no T-cycles, therefore those cycles can not appear in the set history. And C-cycles have been collapsed into one node and assigned to a transaction set. Therefore the ordering $<_{SH}$ is acyclic.

(Only if): Assume the extended history is set serializable but its serialization graph contains T-cycles. There are two cases to consider. If the T-cycle contains only T-edges, it must be of the form $t_1 \rightarrow t_2 \rightarrow ... \rightarrow t_n \rightarrow t_1$, where all t_i are either all transactions or all reconfigurations, but not both. In any equivalent set history, all those transactions must be in different sets since $t_i \rightarrow t_j$ implies $TS_i <_{SH} TS_j$. But then $TS_1 <_{SH} TS_2 <_{SH} ... <_{SH} T_1$ is a cycle and the set history is not serial. Conversely, if the T-cycle contains both T and C-edges, let t_i and t_j be two transactions or two reconfigurations such that $t_i \rightarrow t_j$ (there is at least one T-edge, so they exist). Then in any equivalent serial set history $TS_i <_{SH} TS_j$. Following the cycle we have either $t_j \rightarrow t_k$ or $t_j \Rightarrow t_k$, in any case $TS_j \leq_{SH} TS_k$. And this for all edges on the path. Thus we conclude that $TS_j \leq_{SH} TS_i$, a contradiction with the fact $TS_i <_{SH} TS_j$. In both cases of T-cycles we arrive at a contradiction, therefore the extended history is set-serializable if the serialization graph contains no T-cycles. □

Lemma 1: *Starting with a valid configuration and executed in its entirety, a reconfiguration always leaves the database in a valid configuration.*

Proof: The Reconfiguration Protocol guarantees that a reconfiguration is executed only if it is safe. Set-serializability guarantees that the execution of reconfigurations is serializable (interleaved reconfigurations over the same data item produce T-cycles). Each reconfiguration transforms the database from a configuration to an equivalent configuration. Hence, if the initial configuration is valid, the final configuration is valid.

Lemma 2: *Starting with a valid configuration, any possible execution of a safely divisible reconfiguration leaves the database in a valid configuration.*

Proof: By definition of divisible reconfiguration, each configuration resulting from a partial execution of the reconfiguration is equivalent to the original configuration. Therefore, using divisible reconfigurations consistency is maintained. Since we only use safely divisible reconfigurations, a subset of divisible reconfigurations, the execution of reconfigurations leave the database in a valid configuration. □

Lemma 3: *In an extended history only safe updates are executed*

Proof : This follows from Lemma 2, the definition of the Reconfiguration Protocol and the fact that each update triggers a reconfiguration.

Theorem 2. *A transaction set always transforms the database from a consistent state to another consistent state*

Proof: In a database that contains both partitioned and non-partitioned data, correctness depends on two different aspects. One is the traditional notion of database consistency. The other is the validity of the distributed assertions. Set-serializability does not allow T-cycles, which amounts to say that conflicting transactions must be serializable. Thus, a transaction set of a set-serializable history always preserves this consistency in the traditional sense. Lemma 3 guarantees that all updates are safe, thus all updates in a transaction set are safe. Furthermore, since a set-serializable history is the committed projection of an extended history, all transactions in the transaction set are committed. Hence, a transaction set preserves the validity of the distributed assertions. Since transaction sets preserve correctness in this two aspects, they transform the database from a consistent state to another consistent state.

Theorem 3. *If the transactions follow Strict-2PL, the reconfiguration transactions also follow Strict-2PL and the commit-agreements are followed the resulting extended history has no T-cycles*

Proof: In the only-if part of the proof for Theorem 1 we saw that the T-cycles can be of two types. With only T-edges or with T and C-edges. T-cycles with only T-edges involved either only transactions or only reconfigurations. In any of the two cases, a cycle is not possible since they follow the rules of Strict-2PL, a concurrency control protocol that guarantees serializability. In the case the T-cycle contains both type of edges, take a T-edge $t_i \rightarrow t_j$. t_i and t_j are either both transactions or both reconfigurations. In both cases, t_i must commit before t_j acquires the conflicting lock. Following the cycle, either $t_j \rightarrow t_k$ or $t_j \Rightarrow t_k$, in the first case t_j commits before t_k acquires the conflicting lock. In the second case t_j must commit before t_k commits. In both cases t_j commits before t_k commits. This for every edge in the cycle. Thus, we conclude that t_j commits before t_i, a contradiction with the fact $t_i \rightarrow t_j$. Therefore, the concurrency control protocol ensures set-serializability. □