Enhancing Concurrency and Availability for Database Systems

by

Wai Chee Ada Fu

B.Sc., Chinese University of Hong Kong, 1983

M.Sc., Simon Fraser University, 1985

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
April 1990

Approval

Name:	Wai Chee Ada Fu
Degree:	Doctor of Philosophy
Title of Thesis:	Enhancing Concurrency and Availability for Database Systems

Examining Committee:

Dr. Joseph G. Peters, Chairman

Dr. Tiko Kameda
Senior Supervisor

Dr. Stella Atkins
Supervisor

Dr. WoShun Luk
Supervisor

Dr. Slawomir Pilarski
Examiner

Dr. Vassos Hadzilacos
Department of Computer Science
University of Toronto
External Examiner

March 28, 1990
Date Approved

Abstract

This thesis consists of two parts. In Part I, we study concurrency control of distributed database systems with emphasis on availability, and in Part II, we study semantically-based concurrency control for a centralized database system.

In Part I, we concentrate on concurrency control schemes for replicated database systems. We start with a slight generalization of the **virtual partition protocol** by El Abbadi and Toueg, which we call **Generalized Partition Protocol** (GVP). We then show some existing protocols and some new ones that are members of the GVP family. We introduce a mathematical structure, called **bi-coterie**, to investigate read quorum and write quorum sets. Next, we introduce the **Transaction Replication Scheme**. Transaction replication eliminates the need for locking remote data as required in some conventional concurrency control schemes for distributed database systems.

In Part II of this thesis, as an example of a semantically-based concurrency control and a proof technique for such schemes, we study nested transaction accessing B-trees. We employ the I/O automaton model introduced by an MIT research group in the specification and correctness proof of this fairly complicated concurrency control scheme.

Acknowledgements

I'd like to thank my supervisor, Prof. Tiko Kameda for keeping my spirits up during some of my weary moments. In addition to being a superb scholar in aiding me with the technical contents of the thesis, Prof. Kameda has helped me all the way from the search for a thesis topic to the details of correcting scripts and text processing of the thesis. It has been most enjoyable to work with Tiko because he encourages search for truth and beauty in life; besides, he likes jokes and puzzles.

I'd like to thank Prof. Woshun Luk for his encouragement on the writing of the thesis and the publication of a paper from it. Woshun is also one of the most friendly professors I know who cares and acts for the right cause. My heart-felt thanks to the external examiner, Prof. Vasos Hadzilacos, for his patience in diciphering confusing statements in the original version of this thesis, and for providing me with valuable and thoughtful comments. Thanks are also due to the other committee members, Prof. Stella Akins, Prof. Jai-wai Han, and Prof. Slawomir Pilarski for their support in the thesis research.

I am most grateful to my good friends Frank Tong and Mimi Kao for helping me to finish the final draft. My kind friend Sanjeev Mahajan has been one of my constant sources of intellectual and ethical stimulations. Thanks to all my friends for filling my life with happy memories, so that I see meaningful things other than the thesis. Thanks to Kersti and the office of the School of Computing Science for helping me to meet the official requirements.

My husband Steven has been almost the sole collector of my complaints and his concern and care were always available when I needed them. All of my worthwhile accomplishments, if any, have been motivated by the philosophy of my parents and my family, who, I am lucky to be able to say, are the greatest persons I know.

Finally, thanks to Simon Fraser University, Advanced Systems Foundation of British Columbia, and the Natural Sciences and Engineering Research Council of Canada for providing me with the environment and means to pursue my research.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Designing a highly available and reliable distributed database system has been an elusive goal for the past several years. Similarly, potential concurrency beyond that achieved by the classical locking scheme hasn't been fully exploited. In our view, the latter situation is partly due to lack of widely-accepted and clear-cut correctness criterion such as serializability as well as lack of a suitable modeling technique. As a result, both correct implementation of a new scheme and its correctness proof become a rather difficult task. This thesis consists of two parts. In part I, we study concurrency control of distributed database systems with emphasis on availability, and in part II, we study semantically-based concurrency control for a centralized database system.

In part I, we concentrate on concurrency control schemes for replicated database systems. There are two major advantages in having replicated data in a wide-area distributed system:

(1) We can improve the reliability of a system since the replicated data act as backup for each other.

(2) We may improve the response time and reduce the communications cost for read-only transactions when we can access global data from the local copy instead of requesting the data from another site.

We start with a slight generalization of the **virtual partition protocol** by [ElT86], which we call the **Generalized Virtual Partition Protocol** (GVP, for short). This is not so much a concurrency control scheme as a paradigm for designing quorum-based schemes with different properties. We then show some existing protocols and some new protocols that are members of the GVP family. In GVP, as well as in the **Dynamic Quorum Adjustment (DQA)** scheme due to [Her87], one needs to have a set P of read quorums and a set Q of write quorums such that each element of P intersects each element of Q.[1] This condition is satisfied if $|H| + |G| > n(X)$ for each $H \in P$ and $G \in Q$, where $n(X)$ is the

[1] Some protocols require that each logical write operation first access a read quorum in P before writing a write quorum in Q.

number of copies of data object X. There are other (perhaps more desirable in some cases) choices for P and Q satisfying the above condition. We introduce a mathematical structure, called **bicoterie** to investigate such pairs (P, Q).

An old idea that had been totally ignored in database applications until recently is the replicated executions of transactions. Transaction replication eliminates the need for locking remote data as required in the conventional schemes. While this **Transaction Replication Scheme (TRS)** eliminates the need for remote locking and saves on communications cost, it pays in terms of replicated, redundant computations. Therefore, TRS is more suitable for "hot-spot" data that require simple computation, for example, the seating plans of an airline reservation system.

In combining GVP and DQA mentioned above (Section 4.8.4), we find that things get quite complicated. We feel that on the basis of informal arguments, we may not be able to convince ourselves that the scheme is correct. Hence we looked into more formal proof techniques. First of all, we made use of formalisms based on logic but found that not only are the resulting proofs difficult to read, but also that they are very long even for simple protocols. We then studied the **I/O automaton model** introduced by an MIT research group for the specification and correctness proof of concurrency control protocols. It is more comprehensible and yields proofs less lengthy than the logic-based approach.

Although we leave the correctness proof for the combination of GVP and DQA based on the I/O automaton model for future research, we do make use of the I/O automaton model in part II. As an example of a semantically-based concurrency control and a proof technique for such schemes, we study **nested transactions** accessing B-trees. This scheme is a generalization of the resilient 2-phase locking [Mos81] applied to the $\mathbf{B}^{\mathbf{link}}$ **tree** [Sag86]. We have been able to use the I/O automaton model in the correctness proof of this moderately complicated concurrency control scheme.

Next we shall give more detailed summaries of the contributions of this thesis.

1.1. Quorum-Based Protocols for Partitioned Replicated Database

We first study the conventional quorum based protocols for partitioned replicated database systems [ElT86, ElT89].

An important result is the derivation of a set of basic criteria necessary for the correctness of any protocol that deals with partition failures by using the following techniques:

(1) quorum consensus,

(2) virtual partition (each transaction executes with a certain "sequence vector" and there is a different quorum assignment for each sequence vector).

(3) any output schedule has an equivalent 1-copy serial schedule that orders the transactions according to the "sequence vectors" with which they are executed.

The sequence vector of a transaction identifies the "view"[2] of the transaction. For a given replicated data object X, let $rq(v)$ and $wq(v)$ denote the **read quorum set** (i.e., the set of read quorums) and the **write quorum set** (i.e., the set of write quorums) for X in "view" v, respectively. Our main criteria are:

Criterion 1: When X is read in view v, at least one copy in each read quorum in v has the most up-to-date value in the previous view v' if no writing on X has committed in view v.

Criterion 2: A view v' is *write-disabled* for data object X (i.e., no writing on X is allowed in view v') if X was read in some later view v (i.e., a view v with a "greater" sequence vector than v').

We show that the existing protocols such as the virtual partition protocol and dynamic quorum assignment [Her87] satisfy these criteria.

In the original virtual partition scheme [ElT86, ElT89], quorums are restricted to have some minimum sizes so that the sum of the sizes of two quorums is greater than the number of copies of the

[2] The use of the term "view" here is different from that in Chapter 4.

data (globally or within a virtual partition). We restate the requirements in terms of intersection of quorums and derive new classes of quorums, which give rise to new members of the GVP family. One such new class is derived from the theory of finite projective planes [AlS68] (a quorum corresponds to a line if the set of copies are taken as points in a finite projective plane). It has the advantage of requiring only $\lceil \sqrt{n} \rceil$ physical read/write accesses for each logical read/write access, where n is the number of copies of a data object.

In the quest to find useful members of GVP, we study read quorums and write quorums as abstract mathematical objects. The most basic requirement is that each read quorum intersect each write quorum. We define an abstract structure called a "bicoterie" as follows: Let U be a set of elements and let $A = (P, Q)$ be a pair of sets of groups of elements from U. A forms a **bicoterie** if

(1) for each group G in P or Q, $G \neq \phi$,

(2) for each group G in P and each group H in Q, $G \cap H \neq \phi$.

(3) for any two groups G and $H \in P$, $G \not\subset H$, and for any two groups G and $H \in Q$, $G \not\subset H$.

Our intention is to use U, P, and Q to model the copies of X, the read quorum set and the write quorum set, respectively. One desirable property for a bicoterie is to be **non-dominated** (cf. [GaB85]). We show that the classes of quorum set pairs derived from the theory of finite projective plane are not dominated by any group of size $\sqrt{n}$ or less, where n is the number of elements.

1.2. TRS: Transaction Replication Scheme

The basic assumptions of the system are the following:

(1) There is an upper bound of *M-delay* (sec) on message transmission delay between any two sites.

(2) Each site has a clock and the clocks at any two sites are synchronized to within *C-diff* (sec).

(3) Each site has a unique ID.

There are two types of data objects: a data object may be **private** to one site, that is, only that site can update the data object but other sites may read it; or a data object may be **public** and every site can read and/or write it. Transactions are then divided according to the type(s) of data objects they access and the way they access the data. In conventional replicated database models, data are all public and every transaction can read and write them. Therefore, we first concentrate on systems with only such data.

At regular time intervals, each site broadcasts all transactions submitted at the site during the last interval to every other site. If no transactions are submitted, then the null message is broadcast. When a site has received all transactions or null messages from all other sites that are sent in the same interval, it can start executing the transactions by scheduling them according to a commonly agreed-on total order: for example, the submission time concatenated with the submission site ID. The execution can be concurrent within a site according to a special timestamping scheme which will not abort scheduled transactions.

Some related work can be found in [GPD86], [PiG87] and [PiG89], where a Triple Modular Redundant Database System is designed to achieve high reliability.

1.3. Concurrency Control of Nested Transactions accessing B-trees

Work has been done to enhance concurrency by using the semantics of particular structures of data such as B-tree, hashed file, etc. ([LeY81], [Sag86], [El187], [VLS87], [ShG88]). However, most such **semantically-based** concurrency control schemes make the simplifying assumption that a transaction consists of a single **decisive operation** [ShG88], such as *read, insert* or *delete.* We deal with a more realistic and more complicated model, where a database is a collection of search structures (B-trees), and each transaction may be nested and may perform more than one decisive operation.

Among the known concurrency control schemes for B-trees, the one due to Sagiv [Sag86] can probably achieve the highest degree of concurrency. He makes use of the B^{link} tree proposed in [LeY81].

A B^{link} tree is obtained from a B-tree by adding to each vertex a pair (k, p), where k is the highest key stored in the subtree rooted at the vertex, and p is a pointer to the next vertex at the same level, called a *link.* In [Sag86], there is no need for **2-phase locking** to ensure serializability, since his transactions have at most one decisive operation. *Search, split* and *merge* operations are non-decisive.

We consider **nested transactions** [LyM86] accessing B-trees, combining the B^{link} tree algorithm with 2-phase locking. As in [Mos81], a transaction can acquire a lock only if all the holders of conflicting locks are its ancestors. (Initially, the root transaction holds all locks.) In applying 2-phase locking, in order to take advantage of the B^{link} tree algorithm, we want to lock the individual vertices and keys of a B-tree.

Serializability is a widely accepted criterion for correctness for conventional schedules. But when the semantics of transactions and data are considered, some non-serializable schedules can be considered "correct". We propose a correctness criterion, called "strongly-serially correct", which basically says that a schedule α is correct if there is a serial schedule β such that no committed "user-visible" transaction can tell the difference between α and β.

We use the **I/O automaton model** [LyM86] to formally describe our system and in proving its correctness. Both transactions and data objects are represented by I/O automata. Some of our results were presented at the 1989 PODS [Fu89].

1.4. Organization of the thesis

As we mentioned above, the thesis contains two parts. Part I consists of Chapters 2 to 6: Chapter 2 is a review of the theory of serializability in concurrency control of database systems. Chapter 3 is a review of related work, including 2-phase locking, timestamping, and replicated database protocols. Readers who are familiar with these areas may skip these chapters. Chapter 4 describes the generalized virtual partition protocol (GVP) and related results. Chapter 5 contains a study of bicoteries. Chapter 6 describes the transaction replication scheme (TRS). Part II of this thesis consists of Chapter 7 and addresses concurrency control of nested transactions accessing B-trees, with specification and proofs by means of the I/O automaton model.

CHAPTER 2

THEORY OF SERIALIZABILITY

2.1. CONCURRENCY CONTROL

Concurrency control is concerned with the synchronization of concurrent transaction operations on a database. The objective is to preserve "consistency" of the database. Let us illustrate with an example.

Example: We consider an on-line airline seat reservation system. Suppose that two customers A and B who want to book a seat are simultaneously accessing the seat-plan of flight 100 by execution of the following two transactions.

Customer A: read the seat-plan of flight 100, if there is some vacant seat, then mark the seat as reserved and assign it to A.

Customer B: read the seat-plan of flight 100, if there is some vacant seat, then mark the seat as reserved and assign it to B.

In the absence of concurrency control, these two transactions could interfere with each other as follows: Suppose there is a vacant seat X on flight 100. A's transaction reads the seat-plan, and finds that X is vacant. Next B's transaction reads the seat-plan, and also finds that X is vacant. Then A's transaction reserves the seat X and assigns it to A. Finally B's transaction also reserves the seat X and assigns it to B. Both customers A and B now think that they have seat X which should not happen. □

In the above example, consistency of the database requires that a seat be reserved by at most one customer at any time. This example shows why concurrency control is necessary in database management. We also see that if A's transaction and B's transaction are executed serially one after the other, then consistency can be preserved. In fact, one widely-accepted criterion for correctness of a concurrency control scheme is that a concurrent execution should be equivalent to some serial execution. This is known as "serializability" and will be formally defined next.

2.2. SERIALIZABILITY IN A SINGLE-SITE DATABASE SYSTEM

A database system consists of a set D of data objects and a set of transactions T = $\{T_0, T_1, T_2, \ldots, T_f\}$. T_0 and T_f are two fictitious transactions called the **initial transaction** and the **final transaction**, respectively. Transaction T_0 is a write-only transaction that "writes" all the data objects in D before any other transaction starts, and T_f is a read-only transaction which "reads" all data objects after all other transactions have completed. A **read operation** $R_i[X]$ of transaction T_i returns a value of data object X, and a **write operation** $W_i[X]$ of transaction T_i updates the value of X.

The execution of a transaction $T_i \in$ T is modeled by a totally ordered set $T_i = (\Sigma_i, <_i)$, where Σ_i is the set of read and write operations issued by transaction T_i, and $<_i$ is a total order on Σ_i, representing the order in which these operations are executed.

A **log** (or **history** or **schedule**) over a set of transactions T is a totally ordered set L = (Σ(T), <), where

(1) $\Sigma(\mathrm{T}) = \bigcup_{i=0}^{f} \Sigma_i$;

(2) $\bigcup_{i=0}^{f} <_i \subseteq <$;

(3) for every $A[X] \in \Sigma_0$ and every $B[Y] \in \Sigma(\mathrm{T}) - \Sigma_0$, $A[X] < B[Y]$ holds; and for every $A[X] \in \Sigma_f$ and every $B[Y] \in \Sigma(\mathrm{T}) - \Sigma_f$, $B[Y] < A[X]$ holds.

Two operations on the same data object are said to **conflict**, if one of them is a write operation. For two operations A and B in Σ(T), we say that A **precedes** B in L if $A < B$.

Two logs L and L' are said to be **equivalent**, written $L \equiv L'$, if for each data object X and indices i and j, transaction T_j reads X from transaction T_i in L if and only if T_j reads X from T_i in L'. A **serial log** is a log such that for every pair of transactions T_i and T_j, either all of T_i's operations precede all of

T_j's, or vice versa. A log L is **serializable** if there exists a serial log L' such that $L \equiv L'$.

2.2.1. Read-From Graphs and DITS

The **transaction read-from graph** (TRF graph, for short) [IKM87] for log L over a set T of transactions, denoted by TRF(L), has a node set T and an arc set A. If a transaction T_j reads X from T_i in L, an arc $(T_i, T_j) \in$ A labeled by X is introduced. This arc is denoted by $(T_i,T_j){:}X$. There are no other nodes or arcs in TRF(L). An **interval** of a TRF graph is a set of all arcs that have the same label and originate from the same node.

The **transaction IO graph** (TIO graph, for short) for log L over a set T of transactions, denoted by TIO(L), is an arc-labeled directed graph with the node set T $\cup$ T', where T' consists of dummy nodes as defined below, and the arc set A. If T_j reads X from T_i, there is an arc $(T_i, T_j) \in$ A labeled by X. If $W_i[Y]$ is a **useless write** (i.e., it creates a value that is overwritten without being read by another transaction), then we introduce a dummy node $T_i' \in$ T' together with a dummy arc from T_i to T_i' labeled by Y. There is no other node or arc in TIO(L).

Definition 2.1: ([IKM87]) Let L be a log over a set T of transactions. A total order << on the node set of TIO(L) is a **disjoint-interval topological sort** (DITS, for short), if it satisfies the following two conditions:

(1) if $T_i << T_j$, then there is no path from T_j to T_i in TIO(L), and

(2) if $T_h << T_k$ and there are two arcs labeled by X from T_h to T_i and T_j to T_k in TIO(L) ($j \neq h$), then $T_i << T_j$. □

Intuitively, if TIO(L) has a DITS, then it can be ordered linearly by the order <<, so that all the arcs are directed from left to right (condition (1)), and no two intervals labeled with the same data object overlap (condition (2)).

Theorem 2.1: [IKM87] A log L is serializable if and only if TIO(L) has a DITS which orders T_0 first and T_f last. □

2.2.2. Serialization under Conflict-Preserving Constraints

It is shown in [Pap79] that testing whether a given log is serializable is, in general, NP-complete. In [IKM87], some conflict-preserving constraints on serialization order are imposed so that testing if a given log is serializable under some such constraints can be performed in polynomial time. The following is the definition of the constraints.

Definition 2.2. Let $h = (\Sigma(T), <)$ be a log, X be some data object,

(a) [ww-constraint] If $W_i[X] < W_j[X]$, then T_i must be serialized before T_j.

(b) [wr-constraint] If $W_i[X] < R_j[X]$, then T_i must be serialized before T_j.

(c) [rw-constraint] If $R_i[X] < W_j[X]$, then T_i must be serialized before T_j.

(d) [rr-constraint] If $R_i[X] < R_j[X]$, then T_i must be serialized before T_j.

A log L is said to belong to classes WW, WR, RW, and RR, respectively, if L is serializable under constraints (a), (b), (c), and (d). The set of serializable logs satisfying both the wr- and rw-constraints is called WRW (also called DSR in [Pap79] and CSR (conflict serializable) in [BHG87]) and it properly includes WW. WRW can be recognized in polynomial time. Recognizing WR, RW, RR is NP-complete.

2.3. SERIALIZABILITY IN A REPLICATED DATABASE SYSTEM

In a distributed database system, data objects may be replicated at different sites. The copy of a data object X at site i is denoted by X_i. A data object and its copies are called **logical data object** and **physical data objects**, respectively. The user, when writing transactions, specifies accesses to logical objects. When a transaction T_i executes, the system uses a **translation function** τ_i to translate **logical operations** into a set of one or more **physical operations**, i.e., $W_i[X]$ is translated to $W_i[X_a]$, $W_i[X_b]$, $\cdots$, $W_i[X_l]$, where $X_a, \cdots, X_l$ are *some* copies of X and $R_i[X]$ is translated into $R_i[X_e]$ for some copy X_e of X.

The execution of a set of transactions in a distributed database system with replicated data objects can be modeled by a **replicated log** (or rd log) [BeG81]. A replicated log over a set T of transactions $\{T_i = (\Sigma_i, <_i)\}$ is a partially ordered set $L = (\Sigma(T), <)$ such that

(1) $\Sigma(T) = \bigcup_{i=0}^{f} \tau_i(\Sigma_i)$, where τ_i is the translation function for T_i;

(2) for each i and any two operations p_i and q_i in Σ_i, if $a \in \tau_i(p_i)$, $b \in \tau_i(q_i)$ and $p_i <_i q_i$, then $a < b$;

(3) all pairs of conflicting physical operations are < related (two physical operations **conflict** if they operate on the same physical copy of a data object and at least one of them is a write operation); and

(4) T contains two fictitious transactions T_0 and T_f. T_0 is translated into a set of physical write operations, one for each copy of each data object, and these precedes all other physical operations. T_f is translated into a set of physical read operations, one for each logical data object, and these are preceded by all other physical operations.

A transaction T_j **reads X from** another transaction T_i in a rd log $L = (\Sigma(T), <)$ if there exists a copy X_a such that (1) $W_i[X_a]$ and $R_j[X_a]$ are operations in $\Sigma(T)$; (2) $W_i[X_a] < R_j[X_a]$; and (3) there is no $W_k[X_a]$ such that $W_i[X_a] < W_k[X_a] < R_j[X_a]$.

A rd log L_1 is **equivalent** to another log L_2, if both L_1 and L_2 have the same read-from relation. A rd log is **one-copy serializable** (1C serializable) if it is equivalent to a 1 copy serial (1C serial) log over the same set of transactions.

If in a rd log $L = (\Sigma(T), <)$ transaction T_j reads X from T_i, then the arc from T_i to T_j in TIO(L) should be labeled by X, i.e. , the arc is labeled by a logical data object. The TIO graph of a rd log is defined in the same way as the TIO graph of a log in a single-site database.

Theorem 2.2: [Che88] A rd log L is serializable if and only if TIO(L) has a DITS which orders T_0 first and T_f last. □

CHAPTER 3

RELATED WORK

3.1. CONCURRENCY CONTROL PROTOCOLS

3.1.1. 2-PHASE LOCKING (2PL)

Two phase locking (2PL) was introduced in [EGL76]. The basic 2PL follows the *two phase rule* which states that once a transaction releases any lock, it may not subsequently obtain any more locks. To avoid **deadlocks**, transactions may obtain all the locks before execution; this is **conservative 2PL**. In this variation, if a transaction T_i is waiting for a lock held by T_j, then T_i holds no lock, and deadlock is not possible. **Cascading aborts** occur if a transaction T_j has read from a transaction T_i which aborts, so that T_j must also abort. In order to avoid cascading aborts, we may use **Strict 2PL**, where a transaction T_i keeps all its locks until it aborts or commits so that no transaction T_j can read from T_i until T_i commits.

3.1.2. TIMESTAMP ORDERING (TO)

The **basic TO** dictates that if an operation of T_i arrives after some conflicting operation of another transaction T_j has been scheduled and T_j has a greater timestamp than T_i, then T_i is aborted and resubmitted with a greater timestamp. To avoid cascading aborts, we may use **Strict TO**, where in writing a data object X, a transaction would mark the object inaccessible to subsequent reads until it aborts or commits. This is similar to write-locking the data object X.

In a distributed system, to avoid aborting scheduled transactions, we can use **Conservative TO** [BeG81]. As with Conservative 2PL, one assumes that transactions predeclare their readsets and writesets. An operation of a transaction is always scheduled after all conflicting operations of transactions with smaller timestamps. TO is simpler than distributing 2PL, where coordination among all sites is needed to handle distributed deadlocks.

3.2. COPING WITH FAILURES: A REVIEW

In a distributed database system, different kinds of failures can occur. One of them is a site failure, which can be either **fail-stop** or a **Byzantine failure**.

In fail-stop failures, a site just crashes, losing all the information it had in volatile memory before the crash; thereafter no transaction operation takes place at the site until it is repaired. In general, fail-stop can be handled by using checkpoints and a transaction log to recover a consistent state of the database [BeG83, Gra79]. A database is in a **consistent** state, if the values of all its data objects are the same as the results of serially executing each transaction completely before starting a new transaction.

In [NeT88], a hierarchy of failure severity is described as follows:

(1) *Crash failure*: (or *fail-stop failure*) A faulty processor fails by halting prematurely. It may lose the data contained in main memory, but the data contained in *stable storage* [LaS76] is unaffected by the failure. There are a number of techniques for increasing the probability that a site behaves as if it failed only in the fail-stop mode [ScS83].

(2) *Send-omission failures*: A faulty site may fail not only by halting, but also by occasionally omitting to send some of the messages that it should send.

(3) *General-omission failures*: A faulty site may fail by halting, or by omitting to send and/or receive messages.

(4) *Arbitrary (or Byzantine) failures with message authentication*: A faulty site may deviate arbitrarily from its prescribed behavior. However, sites use a message authentication mechanism so that faulty sites or links can neither alter messages nor spontaneously generate spurious messages that claim to be from other sites.

(5) *Arbitrary (Byzantine) failures*: A faulty site can have arbitrarily behavior and sites do not have access to built-in message authentication.

With Byzantine failures, no assumption is made about the behavior of a faulty site. Many existing replicated database protocols (e.g., protocols to be discussed later in this section) cannot cope with Byzantine failures. Instead, fail-stop failures are usually assumed. With Byzantine failures, a set of faulty sites might cooperate to disrupt correct communication among other sites. There has been a great deal of research on the theoretical aspect of the problem. In this thesis, however, we shall not consider Byzantine failures, since the overhead of coping with such general failures are normally too high to be practical. In practice, therefore, some provisions are often made to convert those failures which are difficult to cope with into fail-stop failures. In the rest of this chapter, we shall discuss partition failure handling with replicated data.

Network partitioning may be caused by link failures or site failures. A link failure occurs when the direct (i.e., not going through other sites) physical connection between from one site a to another b has broken down. A link failure is **clean** if it disrupts the direct physical connection from a to b and that from b to a at the same time.

To make these definitions more formal, we define relation R between two sites a and b as follows:

$a\ R\ b$ iff messages from a can reach b within a time-out period.

A link failure has occurred if for some a and b, $a\ R\ b$ has been true and is now false. Thus, link failures are **clean** iff relation R is reflexive (i.e., $a\ R\ b$ implies $b\ R\ a$). We say that partition failures are **clean** iff relation R is reflexive and transitive (i.e., $a\ R\ b$ and $b\ R\ c$ implies $a\ R\ c$).

When a partition failure occurs, we say that each site a is in a "partition seen by a", which is the set of sites x such that $a\ R\ x$ and $x\ R\ a$. Therefore, in a clean partition failure, the partition seen by two sites will be either disjoint or identical, and any two sites in the same partition can communicate and any two sites in different partitions cannot communicate. In an unclean partition failure, it may happen that a site a can communicate with b and with sites in a set P, while b cannot communicate with P. In this case, the partition seen by a contains b and P. while the partition seen by b contains a but not P.

It is not possible for a site to instantaneously track the partition failure accurately. Since partition failures occur dynamically, while a site a thinks that it can communicate with a group of sites S_1, a link might have failed or have been repaired so that the group of sites it can communicate with is actually $S_2 \neq S_1$.

A partitioned database protocol must solve two problems:

(1) Correctness must be maintained within the part of the database operated on by the sites comprising the partition, and

(2) Correctness must be maintained *across all partitions*, despite dynamic changes of the topology of the network.

Many solutions are based on the simple observation that a sufficient (but not necessary) condition for correctness is that no two partitions execute conflicting data operations (in addition to the use of a correct algorithm *within* each partition).

A partition-handling strategy must also solve the following two problems.

(1) **Atomic Commit:** The database is faced with the problem of *atomically committing ongoing transactions* in spite of the partitioning. This problem arises in any distributed database system whether it is partitioned or not.

(2) **Recovery:** When partitions are reconnected, mutual consistency between copies in different partitions must be reestablished. That is, the updates made to a logical data object in one partition must

be propagated to its copies in the other partitions.

The second problem may be solved by extra bookkeeping whenever the system partitions. For example, each update applied in a partition can be logged, and this log can be sent to other partitions upon reconnection. Such a log may be integrated with the *recovery log* that is already kept by many systems. Hence an efficient solution is likely to depend on the normal recovery mechanisms.

In [Che88], there is a discussion on **prevention protocols**, which are protocols that make sure that the global execution consisting of all the operations granted in individual partitions is serializable. These are **pessimistic strategies** as classified in [DGS85].

In this section, we review the handling of partition failures by prevention protocols.

PREVIOUS WORK

The general strategy used for prevention protocols is to define a mutually exclusive condition for read and write operations on the copies of the same logical data object. If a write operation on a data object is allowed in one partition, then usually any read or write operation on copies of the same logical data object is not permitted in any other partition.

(1) Primary Site/Copy [Alsberg and Day 1976; Stonebraker 1979]

Alsberg and Day [AlD76] used the notion of "primary site" to implement read-write exclusion. In their **primary site** model, a single site is designated as the primary site and every read/write access to *any* data object must first be granted by the scheduler at that site. In the original proposal, locking was used by the scheduler. However, this scheme is too centralized, causing a bottleneck at the primary site. Also, a failure of the primary site will jeopardize the whole system. In the case of a partition failure, only the transactions submitted in the partition which contains the primary site can be executed.

Stonebraker modified the idea of primary site by "distributing" the primary site. Instead of one primary site, one copy of each data object is designated as the **primary copy** [Sto79] of that data object and these primary copies are distributed at different sites. Any access to a data object must be preceded by the locking of its primary copy. In this scheme, there are no longer severe bottlenecks. Moreover, in the case of a partition failure, more than one partition might be able to execute transactions. However, this scheme also has some shortcomings.

(1) There might be distributed deadlocks.

(2) If the access demand on a primary copy within the partition in which it resides is relatively low in comparison with that from other partitions, then availability degrades.

(3) This approach works well only if site failures are distinguishable from network failures. If this is the case and the primary copy site for a data object fails, a new primary copy can be elected (for a discussion of election protocols, see [Gar82]). However, if it is uncertain whether the primary copy site failed or the network failed, the assumption must be that the network failed and no new primary copy can be elected.

(4) This approach cannot take advantage of local copies for logical read operations.

(2) Voting [Thomas 1979; Gifford 1979]

The first voting approach was the **majority consensus** algorithm [Tho79]. Gifford [Gif79] presents a generalization of that algorithm which uses a simple and elegant "voting scheme" to enforce read-write exclusion. In this approach, every copy of a replicated data object is assigned some number of votes. Every transaction must collect a **read quorum** of $q_r(X)$ votes to read a logical data object X and a **write quorum** of $q_w(X)$ votes to write the logical data object X. In other words, a transaction at a site s_i can execute a read (write) operation on logical data object X only if there are at least q_r (q_w) votes of X located within the partition to which s_i belongs. In order to achieve mutual exclusion,

(1) $q_r + q_w$ must exceed the total number of votes v assigned to the object.

(2) $q_w > v/2$

The first condition ensures that read and write operations on the same data object are not performed in two different partitions. The second condition guarantees that a write operation on a data object done in one partition will exclude any write operation on the same logical data object in the other partitions. If read operations are much more frequent than write operations, then a "read-one write-all" approach can be applied, where each read quorum consists of only a single element, and hence the local copy of a data object can always be chosen as the read quorum for a logical read operation. The voting approach has the following weaknesses:

(1) There might be distributed deadlocks.

(2) Availability degrades if the access demand on a data object within the majority partition is relatively low in comparison with that from other partitions.

(3) Suppose that data objects are fully replicated and each copy has a vote of one. In order to accommodate partition failures where the size of the majority partition may be close to $n/2$ (n is the number of sites), the write quorums must have close to $n/2$ votes, so that the read quorums also have close to $n/2$ votes. This means that read-intensive databases cannot really take advantage of the local copies.

(3) Missing Writes [Eager and Sevcik 1983]

To fix the last disadvantage above, Eager and Sevcik [EaS83] have proposed the **missing writes** algorithm which is a variant of Gifford's voting scheme. In this scheme, a transaction normally considers a read operation as the reading of any copy and a write operation as the writing to all the copies. However, this is only possible when there is no partition failure. Once a partition failure is detected, the system goes into the "partition mode", in which Gifford's scheme of mutually exclusive quorums is used. If some copy cannot be updated, a transaction T becomes "aware" of a missing update and must run in the "partition mode". Quorums must now be obtained for each data object. This *missing update information* is then passed along to all subsequent transactions that need the information. These transactions also become aware of missing updates and must run in the "partition mode". Missing update information is posted at sites, along with a description of what transactions need the information. The major advantage of this scheme is a reduction in the overhead of reading when there is no failure. One disadvantage is that during a partition failure, each logical read has to access a read quorum of data copies, thus reducing availability.

(4) Virtual Partition [El Abbadi et al. 1986]

Abbadi, Skeen and Christine [ESC85], and later El Abbadi and Toueg [ElT86], modified Gifford's scheme to the "virtual partition scheme". They attempt to track changes in the network topology as closely as possible without being constrained by the need to cope with the changes instantaneously. A **virtual partition** as seen by a site is a set of sites the site assumes that it can communicate with. A data object is accessible in a virtual partition, if the partition has a majority of its copies. This scheme permits cheaper read operations. (We can choose to use the read-one write-all approach in a virtual partition.) In return, it must be made sure that all the copies have the most up-to-date value when a virtual partition is formed. This is done by a **view-update transaction.** Such bookkeeping incurs a lot of overhead whenever there is any change in the network's topology. (See Section 4.2 for more details.)

(5) Dynamic Quorum Adjustment [Herlihy87]

In this protocol, each operation is associated with a set of quorums (a quorum being a set of data copies that the operation must access). Each object has a *quorum assignment table* which binds quorum assignment to each level (a row of the table). The most up-to-date version of the data object at each level is maintained. An object's quorum assignments must satisfy the following "quorum intersection invariant": *If writes to that object are enabled at level l, then each write quorum at level l must intersect each read quorum at levels greater than or equal to l.* A quorum assignment at a level can be used if a user transaction can execute logical read (write) operations by selecting read (write) quorums from the quorum assignment at that level. If a partition failure occurs in such a way that a higher level assignment can be used in place of the current level, then no bookkeeping is necessary for the adjustment. The quorum assignment at a level may change via a "deflation" process which is similar in nature to a view-update transaction of the virtual partition protocol, but since the process is done on a per object basis instead of a per site basis, the overhead will be smaller. However, the overhead in storing the assignment tables for each object can be excessive. More details of this protocol are described in Section 4.8.

CHAPTER 4

A FAMILY OF VIRTUAL PARTITION PROTOCOLS

4.1. INTRODUCTION

Replicated database systems hold a great potential for achieving higher database availability and faster accesses. The subject of replica control for replicated distributed database systems has recently been of intensive research interest (for surveys, see [DGS85], [BHG87] Chap. 8). In most known concurrency control schemes for replicated distributed database systems which are resilient to partition failures, transactions in only one partition with the majority of "votes" for a logical data object X are allowed to read and write X. In particular, if each data object is replicated at all sites and each site has one vote for each data object, then only the partition with the majority of sites can write any data object. This is the case for such schemes as **weighted voting** [Gif79], **missing writes** [EaS83] and **virtual partition** [ESC85]. This restriction is undesirable in many applications since, if the network is partitioned into three or more groups, each having only a minority of sites, then no partition will be able to write any data object. Even if the network is always partitioned in such a way that the majority partition exists, minority partitions suffer. We are interested, therefore, in protocols that do not give the sole right to the majority partition.

The major motivation behind the work reported in [ESC85] and [ElT86] was to make read operations (which are more numerous than write operations in most applications) less expensive in the face of partition failures. The **virtual partition protocol** (VP) of [ElT89] presents the most recent results by El-Abbadi and Toueg in this direction. Here we carry these efforts a step further. We propose the **generalized virtual partition protocol** (GVP), abstracting and generalizing the idea behind VP. GVP is presented in terms of general conditions to be satisfied. The main concept is that of a **read (write) quorum** for a data object X, which is a set of copies of X that need to be accessed in order to perform a logical read (write) operation on X. (See Sec. 4.2.) Therefore, GVP can be considered both as a paradigm for, and as a family of, replica control protocols, and different protocols can be derived from it by choosing appropriate quorums, depending on the needs of particular applications. We also present a simple correctness proof for GVP.

As an example of a protocol belonging to GVP, we then design a new protocol, called the **small partition protocol** (SP), combining the ideas of *primary copy* [Sto79] and quorum consensus. As in the primary copy scheme, each data object X is "owned" by one site $S(X)$. While in [Sto79] $S(X)$ is selected dynamically to be the lowest-ordered copy of X (the copies of each data object are totally ordered) in the majority partition, we assign $S(X)$ statically. During a partition failure, only the partition containing the site $S(X)$ is allowed to access X, and reading and writing within the partition is quorum-based. (Unlike the primary copy scheme, reading need not access the copy at $S(X)$). If the ownership of data object is evenly distributed, every partition will be able to access some data objects, and the amount of accessible data will be roughly proportional to the size of the partition. SP will work well if most transactions access only one data object. If a transaction accesses many data objects, however, the probability that all of them are accessible within a partition will be small.

We also design another protocol belonging to GVP, based on the theory of *finite projective planes* [AlS68]. A similar idea was used in [Mae85] for achieving mutual exclusion in decentralized systems. This results in the **finite projective plane protocol** (FP), which requires accesses to only $\left\lceil \sqrt{n[X]} \right\rceil$ copies for each logical read or write access to data object X, where $n[X]$ is the number of copies of X.

VP and SP are also members of a family of **Vote Assignment Protocols**, VAP, which is a subfamily of GVP. Thus, the correctness of GVP implies the correctness of all its members, in particular, all protocols in VAP, and the three protocols, VP, SP and FP.

Virtual partitions, also called **views** in [ElT86], are defined as follows. Intuitively, the view of a site s contains all sites with which s assumes it can communicate, i.e., the partition that s thinks it belongs to, although it need not always reflect reality. We find this concept of virtual partition or view very helpful in partition handling. This is because the real partitioning is too elusive to keep track of; it is not possible for a site to always know what partition it is in. One approach is to make use of views and make the execution equivalent to a 1C serial schedule (recall the definition in Chapter 2) that orders transactions according to a total ordering of the views. We derive some criteria which such protocols should satisfy and show that any member of GVP and dynamic quorum adjustment (with a proper definition of views in terms of the "levels" in the protocol) in [Her87] satisfy these criteria.

4.2. REVIEW OF VIRTUAL PARTITION PROTOCOL (VP)

In this section, we briefly review the VP protocol as presented in [ElT89]. Note that the terminology used here is somewhat different from the original and we present VP in a slightly generalized form (as pointed out later in this section). For each data object X, there are two positive integers, $A_r[X]$ and $A_w[X]$, called **read** and **write accessibility thresholds**, respectively, satisfying

$$A_r[X] + A_w[X] > n[X], \tag{1}$$

where $n[X]$ denotes the total number of copies of X. Thus, a set of copies of X of size $A_w[X]$ has at least one copy in common with any set of copies of X of size $A_r[X]$. Each site maintains a set of sites called its **view**. Views are totally ordered according to their unique **view-id's**, which are non-negative integers.

Each copy of a data object has a **version number** = $<V\_id, k>$, indicating that it was last written in view V with view-id V_id and that its value is the result of the kth update in that view, where $k = 0$ indicates the initial value written by the "view-update transaction" (see below). A "*less than*" (or "*larger* *than*") relation is defined among version numbers by their lexicographical ordering. (I.e., a version number $<V_1\_id, k_1>$ is less than $<V_2\_id, k_2>$ if $V_1\_id < V_2\_id$, or $V_1\_id = V_2\_id$ and $k_1 < k_2$.)

In view V, each logical data object X is assigned, if possible, read and write **quorum sizes**,[3] $q_r[X,V]$ and $q_w[X,V]$, which specify, respectively, how many copies of X must be accessed to, respectively, read and write X in view V. (An access operation on a copy may return only its version number, not its value. In this thesis, reading/writing a value is also called an access.) In our terminology, a **view read (write) quorum** for data object X in view V, is a set of copies of X that can be accessed to perform logical read (write) on X in view V. $rq(X,V)$ ($wq(X,V)$) denotes the set of all view read (write) quorums for X in V. Let $n[X,V]$ be the number of copies of X that reside at sites in view V. The quorum sizes must satisfy the following conditions. For all X and V,

$$q_r[X,V] + q_w[X,V] > n[X,V] \tag{2}$$

$$1 \le q_r[X,V] \le n[X,V] \tag{3}$$

$$A_w[X] \le q_w[X,V] \le n[X,V] \tag{4}$$

$$2q_w[X,V] > n[X,V] \tag{5}$$

These ensure that each view write quorum for X in V, if any, has at least one copy in common with each view read quorum for X in V (by Eq. (2)) and with any view which has at least $A_r[X]$ copies of X (by Eqs. (1) and (4)). If there are at least $A_r[X]$ copies of X in view V, then we say that X is **inheritable**[4] in V. If $n[X,V] < A_w[X]$, then there is no choice for $q_w[X,V]$ which satisfies Eq.(4); in this case both $rq(X,V)$ and $wq(X,V)$ will be $\varnothing$.

[3] In [ElT89] $q_r[X,V]$ and $q_w[X,V]$ are called *quorums*.

[4] This is our own terminology, not used in [ElT89].

Consider a transaction T executing at a site s having view V with view-id V_id. (In this case we say that T **executes** in V.) It can read or write copies at another site s' only if s' also has view V with the same view-id. (Ways to relax this restriction are discussed in [ElT89].) If $rq(X,V) \neq \emptyset$ then the logical read operation $R_T[X]$ by transaction T executing in V with view-id V_id is implemented as follows (the steps R3 and W3 are justified in the paragraphs that follow):

R1: Access all copies in a view read quorum in $rq(X,V)$ at sites having view V with view-id V_id,

R2: Determine $vnmax = <Vidmax, k>$, the maximum version number among the accessed copies, and

R3: If $V\_id \neq Vidmax$, then abort T, else read a copy in $rq(X,V)$ with version number $vnmax$.

Note that in [ElT89], X cannot be read in V unless X is also inheritable in V. We relax this requirement by allowing a read operation on X in V once X has been "initialized" in V. (See condition (3) in Section 4.3.) This will make it possible for two concurrent partitions (under different views) to perform both read and write operations on the same logical data, provided that some transaction has performed a write without read on the data.

If $wq(X,V) \neq \emptyset$ for view V with view-id V_id, then the logical write operation $W_T[X]$ by transaction T executing in V is implemented as follows:

W1: Access all copies in a view write quorum in $wq(X,V)$ at sites having view V with view-id V_id,

W2: Determine $vnmax = <Vidmax, k>$, the maximum version number among the accessed copies[5], and

W3 Update the copies in a view write quorum in $wq(X,V)$ and change their version numbers to $<V\_id, k+1>$, if $V\_id = Vidmax$ and to $<V\_id, 1>$ if $V\_id > Vidmax$.

A site may change its view from time to time. For example, a site may want to change its view when it notices a difference between its current view and the sites it can actually communicate with.

[5] $Vidmax \leq V\_id$ holds since each accessed site has view-id $= V\_id$, and from step W3 and the "view-update transaction" to be discussed below, the view-id part of a version number of a copy is never increased beyond the view-id of the site that contains the copy.

Whenever a site s changes its view to a new view, s must execute a **view-update** transaction[6] that updates data object copies stored at site s. Site s may decide on the members of a new view V based on its own information, in which case s is called the **initiator** of V. It may also decide to use a view V initiated by another site, in which case, s **adopts**[7] view V.

Sites change their views atomically as follows. (For details, see [ElT89].) An initiator s of a new view V first assigns to V a unique view-id, *new_view_id*, that is larger than any other view-id that s has encountered. (Uniqueness of the view-id can be achieved by using the initiator's unique site ID (identification number) to be the least significant digits of the view-id.) Site s then executes a view-update transaction to update the local copy of each data object inheritable in V. For each such data object X, the view-update transaction reads the copy of X at $S'(X)$, where $S'(X)$ denotes a site in V that has a copy of X with the largest version number among a set of $A_r[X]$ copies. The version number of X is set to be <*new_view_id*, 0>. If the view-id of $S'(X)$ is not larger than *new_view_id*, then the value of X is copied from $S'(X)$; otherwise, the view-update transaction is aborted. When this is repeated for all inheritable data objects X, the new view is **installed** at s. If a site s' accessed by the view-update transaction has a view-id less than *new_view_id*, then s' immediately adopts *new_view_id*. If a site accessed by the view-update transaction has a view-id greater than *new_view_id*, then the view-update transaction is aborted, in which case site s immediately adopts the greater view-id and initiates a view-update transaction with that view-id.

Some comments are in order for the case where X is not inheritable in V, since unlike [ElT89], we may still allow reading of X in V. After V has been installed at some sites, but before any user transaction is executed in V, the copies of X at these sites, if any, have version numbers $<V'\_id, k>$ such that $V'\_id < V\_id$. At this time, no user transaction should be allowed to read X (R3). However, X can be

[6] As against a *user* transaction. In [ElT89] it is called an *update* transaction.

[7] This is our own terminology, not used in [ElT89].

written if $wq(X, V) \neq \emptyset$, and the first write on X in V will **initialize**[8] X in V, by changing the version number of the updated copies to $<V\_id, 1>$ (W3). Thereafter, X can be read in V (R3).

Definition 4.1: View V' **precedes** V if the view-id of view V' is smaller than that of V. View V' is the **immediately precedes** view of V if the view-id of view V' is the greatest among those of all views that precede V.

4.3. THE GENERALIZED VIRTUAL PARTITION PROTOCOL (GVP)

Let P and Q be two sets of groups (quorums) of elements (data object copies). If each quorum in P intersects each quorum in Q, then we say that P **intersects** Q **group-wise.** We call a pair (P, Q) of quorums for a data object **read quorum set** and **write quorum set**, respectively, if P intersects Q group-wise.

We now define **Generalized Partition Protocol, GVP.** The major difference from VP is the definition of a quorum as a set of copies and not by its size. This gives us much greater flexibility in selecting a (read quorum set, write quorum set) pair. In each view, a concurrency control protocol schedules the transactions executed in that view. Such a protocol is **correct** if any schedule it generates in the view is 1C-serializable. The following conditions for GVP are implementation-independent (except for condition (5)). Each condition is followed by comments which give intuitive meaning of the condition and also suggest some possible implementations.

(1) [View-id and concurrency control] Each user transaction **executes in one view** at a site. Each view has a unique view-id, V_id, and a new view to be initiated or adopted by a site has a view-id not smaller than any view that the site has come across. The transactions executing in a view are controlled by a correct concurrency control protocol within the view and are committed atomically.

[8] This is our own terminology, not used in [ElT89].

The concurrency control protocol is based on quorum consensus. Moreover, each logical write of transaction T must write a write quorum with a version number greater than that written by any logical write by another transaction T' which commits before T commits.

Comments: Let *current_Vid* and *high_Vid* be two mappings from the set of sites to the set of view-id's. *current_Vid(s)* changes whenever s installs a new view and takes as its value the view-id of the new view. If a site s receives at time t a message sent at time t' from another site s', then *high_Vid*(s) changes to $Max\{high\_Vid(s)$ at time t, $high\_Vid(s')$ at time $t'\}$. Intuitively, *high_Vid*(s) is a "high water mark" of view-id's known to s. By definition, we have *current_Vid(s)* $\leq$ *high_Vid(s)*. One way to satisfy condition (1) is for a view initiator s to choose a new V_id greater than the current value of *high_Vid*(s) (*high_Vid*(s) then becomes V_id). A site s may also change its view by adopting *high_Vid*(s) when *high_Vid*(s) becomes greater than *current_Vid*(s). To achieve uniqueness of view-id, the site ID can be used as suggested in the previous section. □

(2) [Global read quorums] For each data object X, a **global read quorum** set $RQ(X)$ is defined. X is **inheritable** in view V if V contains a global read quorum belonging to $RQ(X)$.

Comments: Let $A_r[X]$ satisfy the inequality (1) of Section 4.2. Then for VP, we have

$RQ(X) = \{ S \mid S$ contains at least $A_r[X]$ copies of $X\}$. □

(3) [View quorums] For each data object X in view V, a **view read quorum** set $rq(X,V)$ and a **view write quorum** set $wq(X,V)$ are defined. Each view write quorum in $wq(X,V)$, if any, intersects each quorum in $RQ(X)$ and each view read quorum in $rq(X,V)$, if any. If $wq(X,V) \neq \varnothing$ ($rq(X,V) \neq \varnothing$), X is said to be **writable (readable)** in V.

Comments: Let $q_w[X,V]$ and $q_r[X,V]$ satisfy the inequalities (2)-(4) of Section 4.2. Then for VP, we have

$wq(X,V) = \{ S \subseteq CP(X,V) \mid S$ contains at least $q_w[X,V]$ copies of $X\}$,

$rq(X,V) = \{ S \subseteq CP(X,V) \mid S$ contains at least $q_r[X,V]$ copies of $X\}$, where $CP(X,V)$ denotes the set of all copies of X stored at sites in V.

Note that X may be readable in V even if X is not inheritable in V. In this case, a transaction executing in V can read X only if some other user transaction executing in V has "initialized" X in V. (See condition (5) below.) □

A transaction that writes the value of a data object copy for the first time in a new view is said to **initialize** the copy in the view. □

(4) [View updating] When changing its view to a new view V', a site s, if possible, executes a **view-update transaction** in view V', which atomically initializes its copy of each data object X inheritable in V' by reading the most up-to-date copy of X in a global read quorum in $RQ(X)$ contained in V', where each accessed copy must have a version number $<V\_id, k>$ with $V\_id \leq V'\_id$.

Comments: Site s may not be able to initialize *every* inheritable data object. Skeen shows that there exists no commit protocol that is non-blocking to partition failures [Ske82b]. Therefore, after a partition failure, some transactions may be in the "blocked" state, and cannot be committed or aborted. If such a transaction holds a write-lock on a data object copy, for example, then its value cannot be read to initialize other copies. □

Let *update_Vid* be a partial mapping from the set of all copies at all sites to view-id's defined as follows: $update\_Vid(X,s)$ is the largest view-id of any view such that its view-update transaction accessed a read quorum in $RQ(X)$ which contained a copy of X at site s. (Initially $update\_Vid(X,s) = -\infty$.)

(5) [Reading & writing] • A logical write $W_T[X]$ by user transaction T executing in view V with view-id V_id is required to update/initialize all copies in a view write quorum in $wq(X,V)$ (where each copy has a version number that contains a view-id $\leq V\_id$), giving them the same new version number $<V\_id, k'>$ larger than their previous version numbers. T is committed only if $update\_Vid(X,s) \leq V\_id$ at each accessed site s when T commits. The examination and writing

(if applicable) of the view write quorum must be executed as an atomic action.

- A logical read $R_T[X]$ by user transaction T is allowed in view V only if at least one copy of X accessed by $R_T[X]$ has been initialized in V. $R_T[X]$ proceeds according to steps R1 to R3 of Section 4.2.

Comments: Note that we are in fact using the *relaxed write rule* in [ElT89]: we do not require that the view-id of the site storing a copy of X equal that of the view of a transaction T writing X, but once the copy is written by T, the new version number will prevent read operations on X by transactions executing in views with older view-id's.

An example of a correct concurrency control protocol is VP in [ElT86], which requires that each view write quorum in $wq(X,V)$ intersect every other view write quorum in $wq(X,V)$. (This is stated as an inequality: $2q_w[X,V] > n[X,V]$.) Another example is a quorum-based protocol using $wq(X,V)$ and $rq(X,V)$ with the following modification. Since it is not required that each member of $wq(X,V)$ intersect every other member, a logical write $W_T[X]$ should be implemented by steps R1 and R2 followed by W3 of Sec. 4.2. In this case, locking in R1 and R2 refers to write locks. This makes sure that two write operations on X in the same view discover conflict between them. Since transactions are executed atomically, the value written by a write operation is reflected in the database only if the writing transaction has committed. $\square$

Having extracted the essential features of VP, we now have a great deal of flexibility in designing replica control protocols satisfying the conditions of GVP. VP is based on the vote assignment approach, where the vote given to each copy is one. GVP is based on the quorum assignment approach, which has been shown in [GaB85] to be more general than vote assignment. We shall discuss some members of GVP in Section 4.6.

4.4. CORRECTNESS PROOF FOR GVP

Since view-update transactions read/write copies of data objects, we need to define "reads-from" relation that includes such operations.

We say that user transaction T **reads** X **from** user transaction T' if T reads X from T' in the usual sense [BHG87] or if there exists a set UT of view-update transactions $\{T_1, T_2, ..., T_n\}$, such that T_1 reads X from T, T_2 reads X from T_1, ..., and T' reads X from T_n.

In a serial schedule α, transactions are serially executed, i.e., each transaction reads a data object X, from the transaction preceding and nearest to T in α.

Lemma 4.1: If T reads X from T', and T' and T are not executed in the same view, then T executed in view V_j is a view-update transaction writing the version $<V_j\_id,0>$ of X.

Proof: Suppose T is a user transaction and the operation that reads X from T' is logical operation $R_T[X]$. From step R2 and R3, whenever $R_T[X]$ is successful, it will read from a version with version number containing V_j_id. We see that in all conditions, a version number can become V_j_id only by initiation, either by view-update transaction T_1 as in condition (4) or by a write operation of a user transaction T_2 as in condition (5), and T_1 or T_2 must be executed in view V_j. Therefore a user transaction executed in view V_j always reads from another transaction in V_j. Hence T' cannot be from another view.

The only remaining possibility is that T is an update transaction and from condition (4), T reads from the most up-to-date copy of X in a global read quorum, which will be written by some transaction in another view. □

Lemma 4.2: The commit of a transaction T in view V that writes X with a version number n must precede the commit of another transaction T' in view V that writes X with a version number $n' > n$.

Proof: Assume not, that is, T commits after T' has committed. From the requirement on the correct concurrency control protocol within view V in condition (1) of GVP, T must find out the maximum version number *vnmax* written from among all transactions that have committed before T commits. Since T' commits before T, $vnmax \geq n'$, T must write with a version number greater than *vnmax*. Hence $n > n'$, a contradiction.

Lemma 4.3: Any schedule α, consisting of the operations of the view-update transactions as well as those of user transactions, generated by any member of GVP is equivalent to (i.e., has the same reads-from relation as) a 1C serial schedule σ.

Proof: Let $\{ T_i^V \mid 1 \leq i \leq m_V \}$ be the set of all transactions that are executed and committed in view V. By condition (1) for GVP (Section 4.3), the schedule α_V generated in view V is *1C serializable* [BHG87]. Without loss of generality, let α_V be equivalent to the serial schedule $\sigma_V = T_1^V T_2^V \cdots T_{m_V}^V$, a prefix of which contains all the is a view-update transactions executed in V by different sites. We claim that α is equivalent to $\sigma = \sigma_{V_1} \cdots \sigma_{V_m}$, where m is the number of views and $V_1\_id < V_2\_id < \cdots V_m\_id$. To prove this claim, we show that for each transaction T and data object X, T reads X from T' in α iff T reads X from T' in σ. If T' is executed in the same view as T, then condition (1) for GVP guarantees this.

From Lemma 4.1, if T' and T are not executed in the same view, then in α, T executed in view V_j is a view-update transaction writing the version $<V_j\_id, 0>$ of X. This implies that X is inheritable in V_j, and T copies the value of X written by T' which executed in a view V_i such that among all copies of X in a global read quorum contained in V, the value had the largest version number $<V_i\_id, k>$ less than $<V_j\_id, 0>$ at the time of copying.

What remains to be shown is that, in σ, T' is the last transaction preceding T that writes X. Suppose not. Then either a version of X with version number larger than $<V_i\_id, k>$ is written in V_i, or there is a view V_l such that $V_i\_id < V_l\_id < V_j\_id$ and X is written by a transaction executing in V_l. In the first case, from Lemma 4.2, the transaction T'' writing a new version of X with a version number > $<V\_id,k>$ must commit after transaction T' commits and also commit after transaction T has accessed a global read quorum contained in V (otherwise, according to R2 and R3, T would have read X from T'' instead of from T'). To write X, T'' accesses a view write quorum in $wq(X, V_i)$. Since T has successfully read X, X is inheritable in view V_j_id. Therefore, from condition (2), V_j contains a global read quorum Q_R in $RQ(X)$, which has been accessed by T according to condition (4). By condition (3), this write quorum has at least one copy (at site s) in common with the global read quorum Q_R accessed by T to read X. At the time T'' accesses X's copy at s, $update\_Vid(X,s) \geq V_j\_id$ > since it has been read by T with view-id V_j_id. Since $V_j\_id = V_i\_id$, we have $update\_Vid(X,s) \geq V_i\_id$, therefore T'' could not have committed (according to condition (5) for GVP), a contradiction. In the second case, the transaction T'' writing a new version of X in view V_l must also be committed after transaction T has accessed a global read quorum contained in V (otherwise, according to R2 and R3, T would have read from T'' instead of from T'). With a similar argument, T'' would not be able to commit, again a contradiction. □

Theorem 4.1 Any schedule α, consisting of operations of user transactions, generated by any member of GVP is 1C-serializable.

Proof: It suffices to show that in the above scenario, view-update transactions do not "interfere" with the correct execution of user transactions. Note that a view-update transaction simply functions as a "relay", passing data object values written in a previous view to transactions in the current view. Suppose that in α a user transaction T reads the value of X written by another user transaction T' via a view-update transaction. To show that T reads X from T' also in the 1C serial schedule σ', where σ' is obtained from σ by deleting the operations of all view-update transactions, we point out the following characteristics.

In the schedule σ, a view-update transaction T'' in view V always appears after all transactions in views preceding V and before all user transactions in V. Whenever T'' executed at site reads X from T, it also writes the value read into the local copy of X. If T'' reads X from T, then transaction T' reads X from T'' implies that T' reads X from T.

In α, T'' is committed after the commit of all transactions T_W that write X in views V_W preceding V (if T'' commits before T_W, T_W will come across a copy with *update_Vid* $> V_W$*_id* and has to abort). T'' locks and writes atomically a view write quorum before any transactions that reads X from itself in views $\geq$ are committed. Hence if T'' reads X from T, then transaction T' reads X from T'' implies that T' reads X from T.

It follows that the deletion of view-update transactions from both σ and α preserves the equivalence in the read-from relations. Therefore, the schedule over the user transactions corresponding to α is equivalent to serial schedule σ'. □

4.5. OPTIONS UNDER GVP

This section closely follows the discussions in [ElT89] as most of the options open to VP are also open to GVP. Some of them are intended to increase data availability, while others reduce the cost of view-update transactions.

(1) *Options for Changing Views.* One strategy is called the **aggressive tracking** in [COK86]. It tracks changes in the network topology as closely as possible, and reduces the chance of aborting transactions that read or write data objects that are not physically accessible in the current view. If partition failures are not frequent and a failure lasts for some time, then aggressive tracking is suitable. Another strategy is called the **demand tracking** in [ElT89], with which a view is changed only if some "high priority transactions" cannot execute in it, but would be able to execute in the new view. These strategies can be dynamically chosen to meet varying demands of the database system. If partition failures are frequent so that the network topology changes quickly, demand tracking would probably be better than aggressive tracking.

When an object initiates a view-update, other objects in the partition would execute view-update to adopt the new view. Actually since every site in the partition may detect the partition failure and probably initiate a new view (unique at each site or object), and since then only the initiator x with the highest ID will dominate, it may be wise for a site or object y with smaller ID to wait a while for x to do the initiation, and then adopt the new view. However, y may not hear from x because during view-update, x would only try to access a global read quorum, and may miss y, so that y has to initiate a view-update itself. Hence it would be helpful if x would also try to inform every site in the new view. This is applicable to VP also.

(2) *Associating views with data copies.* So far, we have considered a view as an attribute of a site. The view-update transaction initiated at a site s must update all the local copies of inheritable data objects in the new view. [Her87] performs updates on a per object basis, rather than per site. With GVP, we can similarly associate a view with each copy of an object instead of with each site. (*This is equivalent to considering each copy as residing on a "virtual site" of its own.*) Then the view-update transaction for a data object X initiated at site s only updates the value of X_s (the copy of X at s). Together with demand tracking, this policy will greatly reduce the cost of updates. However, there will be overhead in storing views with each data copy.

When views are no longer associated with sites, a site may still record all the views it has come across on behalf of the data object copies. A transaction submitted at a site may choose to be executed in the view with the highest view-id that the site has come cross. The read rules and write rules have not changed if we pretend that each copy resides on a virtual site of its own, so that the view of the data copy is the view of the virtual site.

(3) *Options for Logical Read.* In condition (5) of Section 4.3, we require that a logical read by a transaction T (executed in view V with view-id V_id) read copies in a view read quorum rq in $rq(V,X)$ which reside at sites with view-id *equal* to V_id. This will make sure that at least one of the copies in rq is written by the most recent logical write on X (either in V or in a preceding

view).

An alternative is to require the above logical read to read the copies in a global read quorum RQ in $RQ(X)$, residing on sites with views whose view-id's are $\leq V\_id$. This also ensures reading from the most recent write on X because each quorum in $RQ(X)$ intersects each view write quorum, if any, of every view. Once a copy X_s is accessed by T, it rejects all write operations issued by user transactions executing in views with view-ids less than $V\_id[T]$.

There is a tradeoff between availability and cost in the above alternative. To increase data availability, we would allow a user transaction executing in view V to read a logical data object X by accessing copies that reside on sites with views other than V. However, each read must now access a global read quorum in RQ(X), whose size is always greater than or equal to that of a view read quorum.

(4) *Reducing the Cost of View-Update Transactions.* For each inheritable data object X, we previously required that all sites installing a new view V to independently access a global read quorum of X to initialize their local copies of X. To reduce the overhead, a single site could execute the view-update transaction, accessing a global read quorum to update its local copy of X, and then propagate the updated value to all other sites that want to install the new view. We can choose the site with the greatest site ID within V to perform the view-update.

(5) *Using Multiple Versions.* So far, we only kept the latest version of a data object. Sometimes we may find that a transaction T should be serialized into an equivalent 1C serial schedule before some other committed transactions, so that T may have to read from logical writes before the most recent logical writes (i.e., read from older versions). In such cases, since we do not store older versions, we would have to abort T. With multiple versions, T can read older versions. To modify GVP to a protocol with multiple versions, we associate versions to views (or to view-id's of views). Now, each quorum is a set of data object copies, and each copy has multiple versions for

different views.

The view-update transaction at site s for a new view V with view-id V_id accesses, for each data object X, a global read quorum in $RQ(X)$ and updates the version of X associated with V using the copy in the read quorum that has the maximum version number with view-id $\leq V\_id$. After the view-update, any write with view-id $< V\_id$ is rejected at the site.

The scheduler assigns a view V to each user transaction T (T and its operations executes in V), which need not be the most recent view. However, if the data X written by a transaction T is inheritable in the most recent view, then T will be aborted if it is assigned a preceding view, hence in such cases, T should be assigned the most recent view.

A logical write operation on X in a view V with view-id V_id (writing the version that is associated with the view) is executed by accessing a view write quorum in $wq(X,V)$, examining the versions associated with the highest view-id $\leq V\_id$ in the write quorum and writing with a version number greater than the maximum version number among them.

A logical read $R_T[X]$ in a view with view-id V_id is executed by accessing a view read quorum in which at least one copy of X must have been initiated, and examining the version that is associated with V_id at each copy of the read quorum. The version with the maximum version number is chosen.

4.6. MEMBERS OF THE GVP FAMILY

4.6.1. The Vote Assignment Protocol (VAP)

This is a sub-family within the family of GVP. Let U be the set of copies of a logical data object X. A **vote assignment** is a function $v: U \rightarrow N$, where N is the set of non-negative integers. $v(x)$ denotes the number of **votes** assigned to copy x. Let $TOT(X)$ be the sum of $v(x)$ for all x. For each view V, let $tot(X,V) = \sum_{a \in V} v(a)$ and let $rt(X,V)$ and $wt(X,V)$ be the **read threshold** and the **write**

threshold of the view V, respectively, such that $rt(X,V) + wt(X,V) = tot(X,V) + 1$. Let $RT(X)$ be the **global read threshold** which satisfies $RT(X) + wt(X,V) \geq TOT(X) + 1$ for all V.

We define $RQ(X)$, $wq(X,V)$ and $rq(X,V)$ as follows.

$$RQ(X) = \{ S \mid \sum_{a \in S} v(a) \geq RT(X) \}$$

$$wq(X,V) = \{ S \subseteq CP(X,V) \mid \sum_{a \in S} v(a) \geq wt(X,V) \}$$

$$rq(X,V) = \{ S \subseteq CP(X,V) \mid \sum_{a \in S} v(a) \geq rt(X,V) \}$$

where $CP(X,V)$ denotes the set of all copies of X stored at sites in V

4.6.2. The Virtual Partition Protocol (VP)

We show that VP of Section 4.2 is a member of GVP. Let $A_w[X]$, $A_r[X]$, $q_w[X,V]$ and $q_r[X,V]$ satisfy the inequalities (1)-(5) of Section 4.2. Define

$RQ(X) = \{ S \mid S$ contains at least $A_r[X]$ copies of $X \}$,

$wq(X,V) = \{ S \subseteq CP(X,V) \mid S$ contains at least $q_w[X,V]$ copies of $X \}$,

$rq(X,V) = \{ S \subseteq CP(X,V) \mid S$ contains at least $q_r[X,V]$ copies of $X \}$.

Then we have VP, if logical read and write are implemented as in Section 4.2. The cost of write operations can be reduced if steps W1 and W2 are replaced by R1 and R2, and if for any view write quorum q there is a view read quorum not larger than q.

We can easily see that VP is a special case of VAP, with a vote of 1 assigned to each copy of a data object X, $RT(X) = A_r[X]$, $wt(X,V) = q_w[X,V]$ and $rt(X,V) = n[X,V] + 1 - q_w[X,V]$, which, it is reasonable to assume, equals $q_r[X,V]$.

4.6.3. The Small Partition Protocol (SP)

For each data object X, designate one site $s(X)$ as the **owner** of X (cf. the *primary copy* [Sto79]). We define $RQ(X)$, $wq(X,V)$ and $rq(X,V)$ as follows.

$RQ(X)$: consists of all groups of copies of X, containing the copy of X at $s(X)$.

Let $q_w[X,V]$ and $q_r[X,V]$ satisfy Eq. (2) of Section 4.2. For each view V,

$wq(X,V)$: consists of all subsets of $CP(X,V)$ containing at least $q_w[X,V]$ copies of X, one of which must be the copy at $s(X)$.

$rq(X,V)$: consists of all subsets of $CP(X,V)$ which contain either $q_r[X,V]$ copies of X, or the copy of X at $s(X)$.

It is easy to see that each view write quorum in $wq(X,V)$ intersects each view read quorum in $rq(X,V)$ and each global read quorum in $RQ(X)$.

We claim that SP has a number of advantages over VP. We argue that for VP to be practical in a fully replicated database system, $A_r[X]$ would have to be close to half the total number of sites. If $A_r[X]$ is too small, $A_w[X]$ will be large, meaning that data objects can be updated only when a partition failure results in a large partition with at least $A_w[X]$ sites. (See inequality (4) of Section 4.2.) If $A_r[X]$ is almost half the total number of sites, however, then only the majority partition will be able to access X. SP, on the other hand, needs to access only the copy at $s(X)$, which forms a read quorum by itself, in order to perform a logical read on X. Similarly, view-update transactions are less costly in SP than in VP. The main disadvantage of SP is that X is inaccessible in a large partition unless it contains $s(X)$.

We now show that SP is also a special case of VAP: Let X be a logical data object. For each copy of X at a non-owner site, assign a vote of one. For the owner site copy, assign a vote of $n[X]$, where $n[X]$ is the total number of copies of X. Then $TOT(X) = 2n[X] - 1$. Let $RT(X) = n[X]$. For each view V, set $wt(X,V) = n[X] + w$, for some parameter $0 \le w \le tot(X,V) - n[X]$, and $rt(X,V) = tot(X,V) - wt(X,V) + 1 = tot(X,V) - n[X] - w + 1$.

We see that $RT(X) + wt(X,V) = 2n[X] + w \geq TOT(X) + 1$ for all values of w.

4.6.4. The Finite Projective Plane Based Protocol (FP)

The quorum sets in GVP can be defined in terms of a *finite projective plane* [AlS68]. This results in the **finite projective plane based protocol (FP)** which requires accesses to only $\lceil \sqrt{n[X]} \rceil$ copies for each logical read or write operation on data object X, where $n[X]$ is the total number of copies of X.

Definition 4.1. A mathematical system (plane) consisting of a set Π (of points) and distinguished subsets of Π (lines), $G_1, G_2, \ldots$, is a **projective plane** if the following hold:

(a) If a and b are any two distinct elements of Π, there is a unique distinguished subset G_i such that $a \in G_i$ and $b \in G_i$. *(Any two points are incident with exactly one line.)*

(b) If G_1 and G_2 are any two distinct distinguished subsets of Π, there is a unique element a of Π such that $a \in G_1$ and $a \in G_2$. *(Any two lines intersect at exactly one point.)*

(c) There are four distinct elements a, b, c, d of Π such that no three of them are elements of a common distinguished subset G_i of Π. *(There exist four points, no three of which are colinear, that is, incident with a single line.)* $\square$

A **finite projective plane** is projective plane with a finite number of elements. As an example, suppose a finite projective plane has 7 points $\{1, 2, \ldots, 7\}$. There are 7 lines on the plane as follows.

{1,	2,	4}
{2,	3,	5}
{3,	4,	6}
{4,	5,	7}
{5,	6,	1}
{6,	7,	2}
{7,	1,	3}

For a finite projective plane with 13 points $\{1, 2, \ldots, 13\}$, there are 13 lines as follows:

{1,	2,	3,	4}
{1,	5,	6,	7}

{1,	8,	9,	10}
{1,	11,	12,	13}
{2,	5,	9,	13}
{2,	8,	12,	7}
{2,	11,	6,	10}
{3,	5,	12,	10}
{3,	8,	6,	13}
{3,	11,	9,	7}
{4,	5,	8,	11}
{4,	6,	9,	12}
{4,	7,	10,	13}

[AlS68] (Section 2.3) proves the following theorem.

Theorem 4.1. ([AlS68]) Let Π be a finite projective plane. Then there is an integer m such that

(a) Every point (line) of Π is incident with exactly $m+1$ lines (points) of Π;

(b) Π contains exactly $m^2 + m + 1$ points (lines). □

We say that the above finite projective plane is of **order** m. We can see that if N is the number of points on a finite projective plane, then a line on the plane has $\lceil \sqrt{N} \rceil$ points. Thus, there could be projective planes with N points, where $N = 7, 13, 21, 31, 43, 56, \ldots$ (corresponding to $m = 2, 3, 4, \ldots$). There exist examples of projective planes with 21, 31, and 56 points (order 4,5,7), but it is known that there is no projective plane with 43 points (order 6).

At the present, no efficient algorithm known for generating the lines on a finite projective plane of a given size (if one exists). If the number of data object copies, n, does not exactly match the number of points of any known finite projective plane, then there are two straightforward methods to construct a set of lines over $\leq n$ points such that the they intersect each other. The first method is simply to take the lines in a finite projective plane of the highest order that has a number of points $< n$.

The second method is to take a finite projective plane of the lowest order which has a number of points $> n$, and delete all superfluous points from each line. For example, for 8 points, we can take the projective plane of order 4 with points 1 to 13, and delete from each line any occurrence of points 9 to

13. Then, from the resulting set of lines, we delete any line if it is a superset of another line. The end result, which we call P_8, is as follows:

```
{1}
{2,   5}
{2,   7,   8}
{2,   6}
{3,   5}
{3,   6,   8}
{3,   7}
{4,   5,   8}
{4,   6}
{4,   7}
```

Note that, for each line L in P_7 (finite projective plane of order 3), there is a line in P_8 which is a subset (some of these are proper subsets) of L, and there are lines in P_8 which are not in P_7. Therefore, if each line is taken to be a read or write quorum in a quorum consensus protocols, then there are more choices of quorums in P_8 than in P_7. In this sense, P_8 is more attractive than P_7. We conjecture that the second method always yields a more attractive quorum set than the first method in the above sense. (This idea of "more attractive" will be formalized in Chapter 5.)

In our application of finite projective planes, the set Π of size $n[X]$ is considered as the set of copies for a certain data object X. The set $RQ(X)$ of global read quorums is modeled by the set of all lines of Π. $wq(X,V)$ consists of all global read quorums in V, i.e., a subset of $RQ(X)$, since any global read quorum in V intersects each element in $RQ(X)$. $rq(X,V)$ consists of all subsets of V that intersect each write quorum in $wq(X,V)$. As an example, suppose that the copies of data object X are stored at 7 sites, $S = \{1, 2, ..., 7\}$. We define $RQ(X)$, $wq(X,V)$ and $rq(X,V)$ as follows.

$RQ(X)$: consists of $\{1,2,4\}$, $\{2, 3, 5,\}$, $\{3, 4, 6\}$, $\{4, 5, 7\}$, $\{5, 6, 1\}$, $\{6, 7, 2\}$, and $\{7, 1, 3\}$.

For any view V,

$wq(X,V)$: consists of some or all quorums in $RQ(X)$ that are contained in V.

$rq(X,V)$: consists of all subsets of V each of which intersects every quorum in $wq(X,V)$.

In the next chapter, we shall see that no vote assignment can realize a pair of read/write quorums derived from finite projective planes. This implies that FP is not a member of VAP.

4.7. IMPLEMENTATION DETAILS

We mentioned in condition (1) of GVP that the transactions executed in a view V are controlled by a correct concurrency control protocol and are committed atomically. Here we give some suggestions on the details of a possible implementation.

As stated in Section 4.2, read operations use the version number associated with each copy to identify (and read) the most "up-to-date" copy in a read quorum. For each data object X, let $RQ(X)$ be a global read quorum set, and for each view V, let $rq(X,V)$ and $wq(X,V)$ be the view read quorum set and view write quorum set, respectively. Each site s keeps track of $high\_Vid(s)$, $update\_Vid(X,s)$ for object X, and $current\_Vid(s)$ as defined in the comments on condition (1) of GVP. Each message sent by s contains $high\_Vid(s)$. If $high\_Vid(s)$ deviates from $current\_Vid(s)$ during the execution of a transaction at s, then the transaction is aborted. Initially, all sites have a common view V_0 with view-id V_0_id which consists of all sites, and all copies have version number $<V_0\_id, 0>$. We now describe how user transactions execute read and write operations under GVP.

We assume that each transaction executes in two phases: the **read phase** followed by the **write phase**. All the logical writes of a transaction are reflected in the database atomically by means of a *commit* operation. *Strict 2-phase locking* (i.e., locks held by a transaction are released only after the transaction commits) is used to ensure serializability of the resulting schedules and to avoid *cascading aborts*[9] [BHG87]. Note that any given transaction can easily be transformed into this form by deferring its writes to the end.

[9] The data objects written by a transaction T are called *dirty* until T commits. If another transaction T' reads dirty data and eventually T aborts, then T' must also abort. This is an example of cascading aborts. With strict 2-phase locking, we can avoid this problem.

4.7.1. Logical Read

Due to view-update transactions to be described in Section 4.7.3, a copy of a data object may be marked "*unreadable*", indicating that it is not accessible for reading. Suppose a transaction T is submitted at a site s_o and s_o has view V with view-id V_id when the execution of T is initiated. Then T executes a logical read operation $R_T[X]$ on a data object X as follows:

(1) If $rq(X,V) = \varnothing$, then T is aborted. Otherwise, (the transaction manager at) s_o accesses a read quorum in $rq(X,V)$ as follows: s_o sends a **readlock request message** to each site s storing a copy in the read quorum. When site s receives such a message, if $current\_Vid(s)$ is equal to V_id, then it tries to obtain a readlock[10] on X. (If $current\_Vid(s) \neq V\_id$ or if s is not successful in getting a readlock, s may send a negative reply or do nothing.) After securing a readlock on X, site s returns a **reply message** containing the current value of X, the current version number of X, and $current\_Vid(s)$.

(2) If no reply comes from a site within a "timeout" period, then s_o assumes that the site has failed or is disconnected. In this case and in case a negative reply returns, s_o will access another copy in V until it accesses all copies in a read quorum. If not enough copies are accessible, then T is aborted. (When a transaction T is aborted, s_o sends out messages to release any lock that T may be holding.)

(3) When s_o has received a reply message from every site in a read quorum with the same view-id as s_o, provided that the maximum version number of X has a view-id = V_id, then T is given the copy of X with the maximum version number; otherwise T is aborted because X is not initialized.

(4) If T is not going to write any data object, it enters the *lock-releasing phase* of 2-phase locking. s_o sends a **commit message** to each site in the read quorum to release the readlocks. Otherwise, the

[10]As usual, a readlock conflicts with a writelock on the same data object but not with other readlocks.

steps of logical write in the next section are carried out.

4.7.2. Logical Write

A transaction T submitted at site s_o with view V executes the logical operation $W_T[X]$ on a data object X as follows:

(1) If $wq(X,V) = \varnothing$, then T is aborted. Otherwise, s_o reads either a read quorum in $rq(X,V)$ or a write quorum in $wq(X,V)$ by readlocking the copies in it as in steps (1) to (3) of the logical read on X. If T is not aborted, let the returned maximum version number = $<Vidmax,k>$. The version number to be used for the subsequent physical writes is $<V\_id,k+1>$ if $V\_id = Vidmax$; otherwise (i.e., $V\_id > Vidmax$), $<V\_id,1>$ should be used. The readlocks on X are held until the lock-releasing phase.

(2) s_o selects a write quorum in $wq(X,V)$ and sends a **writelock request message** to each site in the write quorum. The value to be written is piggybacked on the message. When site s receives such a message, if $current\_Vid(s) \le V\_id$ and $update\_Vid(X,s) \le V\_id$ (see GVP condition (5)), then it tries to get a writelock on its copy of X.[11] After securing the writelock on X, a **tentative write** on its copy of X is performed with the piggybacked value. (A tentative write of X stores a tentative value of X in temporary memory, which will be used to update the permanent copy of X only if the transaction writing X commits.) Site s then returns a **reply message** to inform s_o of a successful locking of X and of $current\_Vid(s)$. Negative reply may also be used to speed up timeout.

(3) If s_o does not receive a reply from a site s within a timeout period, or if it gets a negative reply, then it tries to access some other site to make up a write quorum. If this fails, then the transaction

[11] A writelock conflicts with a readlock and another writelock. It also conflicts with view-update transactions, as we shall see later.

is aborted.

(4) When T has locked all the data object copies it is going to write, it can enter the *lock-releasing phase* of 2-phase locking. Then T commits, sending a **commit message** to each site holding copies in the read-quorums and write-quorums of the data objects it accessed. Upon arrival of a commit message, site s copies the tentative write values into stable storage at s, and releases the readlocks and writelocks held by T.

In the above protocol, we have combined the quorum consensus scheme, 2-phase locking, and 2-phase commit all into one package. For 2-phase commit, the sending of writelock requests and the replies constitutes the first phase, and the sending of commit messages is the second phase. The result is a concurrency control scheme that guarantees one-copy serializability in the face of partition failures provided that no view changes take place. The next section describes the steps needed to cope with view changes, completing the description of an implementation of GVP that provides one-copy serializability in spite of partition failures and view changes.

4.7.3. View-Update Transactions

Views may change during transaction execution [ElT86]. As in VP described in Section 4.2, whenever a site s changes its view to a new view, s must execute a view-update transaction that updates data object copies residing at site s. Site s may change its view in two different ways. It may be the initiator and decide on the membership of a new view. It may also decide to adopt a view initiated by another site. A new view may be adopted in three different ways: first, when a site receives a readlock request, writelock request or reply message with a greater view-id; second, when a site receives a read request for a data object from a view-update transaction that is trying to introduce a greater view-id; and third, when an initiator comes across a greater view-id in executing its view-update transaction.

Informally, sites change their views as follows.

(1) When a site s decides to initiate a new view, it first assigns to the new view V a unique view-id *new_view_id* that is larger than any other view-id that it has encountered. We ensure uniqueness by using the site ID of the initiator as the least significant bits of *new_view_id*.

(2) Site s then executes a view-update transaction to update all the local copies of data objects that are inheritable in V. For each such data object X, the view-update transaction accesses the copies in a global read quorum Q_R in $RQ(X)$. Since every successful logical write on X has to write a view write quorum and every view write quorum for X intersects with every quorum in $RQ(X)$, we can be sure to get the most up-to-date value of X. If $high\_Vid(s) \leq new\_view\_id$ for each site s in Q_R, then the value of X is updated using the returned copy with the maximum version number. The $update\_Vid(X, s)$ of each site s in Q_R is now updated to *new_view_id*. If this is the case for all X, the new view is installed at s. The view-update transaction then terminates successfully. (Any site s' with $high\_Vid(s') < new\_view\_id$ adopts the new view by invoking a view-update transaction of its own using *new_view_id*.) However, if a site s' is accessed with $high\_Vid(s') >$ *new_view_id*, then site s will adopt the view corresponding to $high\_Vid(s')$.

Next we consider the case where a view-update transaction at site s with *new_view_id* comes across a writelock L at a copy of data object X at site s' (s' may equal s) holding a copy in a global read quorum in $RQ(X)$ and the view-id of s' is not greater than *new_view_id*. The writelock indicates that a write is in progress or was interrupted by a partition failure. The view-update transaction then marks the copy of X at s **unreadable** so that no transaction in the new view will be able to read it. by a logical write operation in the new view. However, an unreadable copy can be initialized by write operation $W_T[X]$; thereafter, the copy becomes readable. In this case, upon recovery, even if the previous write operation that holds the write lock L succeeds, update will not take place at s since it has been overwritten by $W_T[X]$.

If there was a partition failure, then, on recovery, the sites in some partitions will undergo view changes. However, before the view change, any message that was lost (or when sender gets no acknowledgement) due to the partition failure should be retransmitted. In any event, all the sites with blocked transactions will be able to receive the reply messages and commit or abort, and the locked data objects will be able to receive the release-lock messages and hence release the locks. Such measures reduce the chance that the view change will come across locked data copy.

4.7.4. Correctness

Here we show that the above implementation satisfies the conditions of GVP. First part of condition (1) is satisfied by the 2-phase locking and 2-phase commit protocol. Second part of condition (1) of GVP is satisfied because of the way that the view-id's are chosen by the view-update transaction. The initiator of a new view always chooses a view-id larger than any view-id it has come across, and with the site ID as the least significant bits, the chosen view-id will be unique. Conditions (2) and (3) are satisfied by assumption. Condition (4) is satisfied by the view-update transaction. To satisfy condition (5), step (1) of a logical write assigns a proper version number; for a logical read, step (1) accesses copies at sites with the same view-id as the site initiating the logical read, and step (3) chooses a copy with the maximum version number.

4.8. REVIEW OF DYNAMIC QUORUM ADJUSTMENT (DQA)

In [Her87] Herlihy proposed a protocol called dynamic quorum adjustment (**DQA**, for short). In his terminology a **quorum assignment** associates each operation (read or write) on a data object with a set of quorums. The **quorum assignment table** for an object **binds** a quorum assignment to each **level**, where levels are numbered 1, 2, 3, ... The quorum assignment table for object X is stored with each copy of X and possibly at other sites where no copy of X is stored. Each transaction is assigned to a level, and for all data objects accessed by the transaction the quorum assignments bound to that level are used to execute its operations.

When a transaction begins execution, it chooses its level, which is the level of the quorum assignment tables of the data objects it accesses. Transactions at different levels are serialized in the order of their levels, and the transactions at the same level are serialized by *timestamping*.

A mechanism similar to Reed's multiple version scheme [Ree79, Ree83] is used to ensure that lower level transactions are serialized before higher level transactions. Each copy of a data object maintains several versions, at most one for each level, ordered by levels. If no write operation is performed on a data object at a level, then we say that the version is not **installed** at that level. Each copy also has a **ratchet lock,**[12] which is a counter that records the highest level of a transaction that has read the copy. A logical write on object X is **enabled** at level l if all the ratchet locks in a write quorum for X at level l have values $\leq l$. A write can be carried out only if it is enabled.

Logical read:

A level l transaction reads X by accessing a level l read quorum. At each copy in the read quorum, the version associated with the highest level $\leq l$ is chosen, among which the version with the highest level and latest timestamp[13] is returned to the transaction. When the transaction commits, each ratchet lock in the read quorum is set to the maximum of its current value and the transaction's level.

Logical write:

A level l transaction writes X by accessing a level l write quorum. If none of the ratchet locks in the write quorum $> l$, then the version of X at level l at each copy of the write quorum is updated with its timestamp also updated to that of T.

[12] A ratchet lock here is similar to $update\_Vid(X,s)$ in GVP, which is the highest view-id of a view-update transaction that has accessed the copy of X at s.

[13] Note that the role of the timestamp is similar to the role of version number in GVP.

Due to the "quorum intersection invariant" stated below, a read (write) quorum can grow (shrink) but cannot shrink (grow) as the level increases. Therefore, if a transaction T reads X, a lower level is preferable for T, whereas if it writes X, a higher level is preferable. Since reads are usually more frequent than writes, it is desirable to stay at a low level. However, as stated above, there is a lower limit on the level given by the ratchet locks of the copies of X to be written by T.

An object's quorum assignment table must satisfy the following **quorum intersection invariant**: *If writes to that object are enabled at level l, then each write quorum at level l must intersect each read quorum at levels greater than or equal to l.*

Quorum Inflation: A transaction that cannot access a write quorum because of failures (of sites or links) or ratchet locks may abort and restart at a higher level.

Since a ratchet lock is non-decreasing, the lowest level available to a transaction in general increases with time, which means read quorums grow larger and larger. Therefore, we need a complementary mechanism to inflation for "deflating" read quorums to make reads less expensive. Deflation for an object binds a new quorum assignment to a level of the object's quorum assignment table. Clearly, if the sites are partitioned, only sites in the same partition can be informed of the deflation. In the following definition, an old (new) quorum means a quorum for level l before (after) the quorum deflation at level l. A **coquorum** for an operation is any set of copies which intersects every quorum for that operation. The quorum deflation protocol at level l consists of the following steps, which are executed as a transaction.

Quorum Deflation by object X at level l:

(1) For each copy of X in an old read quorum, read the closest preceding version for level l, its timestamp, and the copy's ratchet lock. If there exists an l', $l > l'$, such that writes are enabled at level l', but some new read quorum does not intersect some level l' write quorum, disable writes at level l' by advancing the ratchet lock beyond l'.

(2) Copy the latest version of X among those read in step (1) and its timestamp to all copies of X in a new write quorum, and the highest ratchet lock value to all copies of X in a new read quorum.

(3) Update the quorum assignment tables at all the copies of X in an old coquorum for read and write, by replacing the level l entries by the new sets of read and write quorums for X.

Step (3) ensures that any new transaction T that is assigned to level l after the "committing" a deflation transaction will know about the deflation. To prove this claim, assume that transaction T successfully executes accessing the old read quorum rq and/or write quorum wq for level l. rq must intersect the old coquorum for read operation that now posses new quorum assignment for level l. Similarly, wq must intersect the old coquorum for write operation that now posses new quorum assignment for level l. Therefore, T will discover that level l has new quorum assignment, a contradiction.

Other transactions can run concurrently with deflation transactions as long as the resulting schedule consisting of all the transactions is 1C-serializable. (Note that a quorum assignment table constitutes a part of a data object, which is read/written by transactions.) For example, a transaction cannot use an old quorum for some operations and a new quorum (installed by deflation) for some other operations.

If deflation was necessary, then it would mean that the read quorums at the old level l were too big. By the quorum intersection invariant, the read quorums at level $l+1$ would be even bigger. To avoid overhead of this "cascading" deflation (deflation at levels above l because of deflation at level l), we might execute a deflation that deflates multiple levels on the tables at the same time.

4.8.1. Relation between Levels and Views

We notice that levels at which transactions execute are similar to views in GVP. However, there is a difference in that, over time, more than one quorum assignment may be bound to the same level by deflation, while a view has unique sets of read and write quorums. Therefore, we find it more appropriate to relate each quorum assignment, instead of a level, to a view of the object. Then the views of a data object (="virtual site", see Sec. 4.8.3) are ordered as follows: a view corresponding to a quorum assignment at level i precedes any view corresponding to a quorum assignment at level $i+1$; and two views corresponding to quorum assignments at the same level are ordered by their binding times to the level. We denote this precedence relation by $<$.

Note that, in GVP, if an object X is inheritable, an old coquorum of write for X automatically gets the new view (hence the new quorum assignment), since a global read quorum of X is accessed by the view-update transaction. Unlike deflation in DQA, however, a view-update in GVP may not access any old coquorum of read for X.

4.8.2. Improvement on DQA

Borrowing an idea from GVP, we can improve on DQA and allow two concurrent levels in different partitions to read and write a logical data object X provided that both partitions contain read quorums of X. The modification is as follows.

(1) In performing deflation at level l, if no old read quorum is accessible, then skip the deflation step (1) and change the deflation step (2) as follows: A new write quorum is accessed and marked "non-inheritable",[14] so that no transaction can read X at level l before a write takes place at level l. Deflation step (3) is carried out as usual.

(2) Read on X by a transaction at level l succeeds if it does not come across any "non-inheritable" copy or if there is a read quorum containing some copy written at level l. If it is successful, it updates the ratchet locks of all copies of X in the read quorum only if the quorum contains no copy that was written at level l.

[14] This marker gets erased when the copy is written at level l. The "non-inheritable" marker carries unique identity of the deflation transaction. The quorum assignment table also records the identity of the last deflation transaction, if any, at each level. If these identities do not match, the "non-inheritable" marker is obsolete and should be ignored or erased.

The reason for (2) is that if the read quorum contains some copy that was written at level l, then either X was initialized by deflation, in which case enough ratchet locks have been advanced already, or deflation did not initialize X but a write at level l did, in which case again there is no need to advance the ratchet locks.

Note that this modification cannot improve the protocol if a transaction always reads a data object before it writes it. Note also that, if the set of all old coquorums of write equals the set of all old read quorums, then DQA does not improve with this modification. This is because, when no old read quorum is accessible, none of the old coquorums of write is accessible; therefore, the modification of (1) in the above has no effect.

4.8.3. Comparison of GVP and DQA

In light of our observation made in Subsection 4.8.1, we relate quorum assignments in DQA to views in GVP. Then we see that the two protocols are quite similar: they both use quorum consensus and generate 1C-serializable schedules that order transactions in the view order. The following are some differences between the two.

One difference (as pointed out in [ElT89]) is that DQA utilizes multiple versions, although DQA need not be restricted to multiversion. If we simply keep only the most up-to-date version (at the highest level where transactions have been executed) and disallow any transactions at lower levels, we have a single-version protocol. In this case, the previous levels of the table need not be stored, since they will not be utilized any more. On the other hand, GVP can be modified to be multiversion (Section 4.5).

Another difference is that DQA associates "views" with data object copies, whereas GVP associates views with sites. Associating views with sites has the disadvantage (as discussed in [Her87]) of costly view-update process, since at view-update, all copies of all data objects stored at a site are subject to updating (this corresponds to *eager* quorum adjustment [Her87]). In Section 4.5, we pointed out that GVP can also associate views with copies. The protocol and correctness proof remain much the same.

View-update is then on per data object basis and is much less costly, but to store a view with each data object copy would incur more overhead than storing a view with each site. We may associate a "virtual site" with each data object copy, however, even if we use "virtual sites", views must be defined in terms of real sites.

From the above discussions, we conclude that neither single- versus multiversion nor copies versus sites to be associated with a view is a major difference between GVP and DQA. The important differences between the two are the use of quorum assignment tables and deflation in DQA versus view-update using a global read quorum set in GVP.

The use of quorum assignment tables in DQA provides some static global information about the whole network: if we do not consider deflation, then each site storing a copy of a data object X can know what quorums of X transactions executed anywhere in the network may access. Since GVP doesn't make use of any quorum assignment table, it cannot ensure that each view read quorum set in a view intersects group-wise each view write quorum set in all previous views; e.g., the view of the sites in another concurrent partition may be a previous view. Therefore, in GVP, a new view always has to secure the most up-to-date value through the view-update transaction, reading from a global read quorum, which is a coquorum of all write quorums. Hence, in changing views, GVP has the overhead of view-update transactions, as compared to no overhead for inflation for DQA.

[Her87] pointed out that quorum inflation and deflation can be combined to realize every quorum assignment permitted by VP as follows. Let V_0 be the first view of VP. Then each data object X binds read and write quorums of sizes $q_r[X,V_0]$ and $q_w[X,V_0]$, respectively, to level 1, and read and write quorums of sizes $A_r[X]$ and $A_w[X]$ to each higher level. A transaction can execute in a new view V by restarting at a level $l > 1$. A deflation transaction is executed to rebind read and write quorums of sizes $q_r[X,V]$ and $q_w[X,V]$, respectively, to this level. We can do something similar to simulate GVP by DQA as follows. Each data object X binds $rq(X,V_0)$ and $wq(X,V_0)$ to level 1, and $RQ(X)$ and $WQ(X)$

to each higher level, where $WQ(X)$ intersects $RQ(X)$ group-wise, i.e., $WQ(X)$ is the set of coquorums of global read (which uses quorums in $RQ(X)$). However, we show below that deflation is not the same as view-update.

One significant difference between DQA and GVP is due to the notion of global read quorum set $RQ(X)$ in GVP. With DQA, different "views" (quorum assignments) may be bound to the same level at different times through deflation. Hence we want to make sure that each transaction (including deflation transaction) executing at level l follows the most up-to-date binding of quorum assignment to level l. This is because, if a transaction reads a read quorum of an old assignment, it is only guaranteed the most up-to-date value from a previous level, i.e., not necessarily the most up-to-date value at the current level (with new assignment). If a transaction T uses an old write quorum at level l, a new read quorum may not intersect it and the write by T may not be read by subsequent transactions executing at level l. This stems from the fact that a new read (write) quorum after deflation at a level may not intersect an old write (read) quorum at the same level. In order to make transactions using an old quorum become aware of the deflation, DQA requires that, at deflation (step 3), the quorum assignment tables at an old coquorum for read and write be updated. Thus, deflation is more restrictive than view-update in GVP. As pointed out earlier, at a view-update in GVP, an old coquorum of write for an inheritable object automatically gets the new view, but there is no need to access any old coquorum of read. In other words, if a global read quorum is accessible, then X is inheritable and the most up-to-date value is read and copied to a view write quorum. If not, transactions executing in the new view cannot read the un-initialized data object copies, but they can still write those data objects whose view write quorum sets are non-empty. Once a data object is initialized by a write in the new view, it is also readable. The reason behind this is that, with GVP, the global read quorum intersects all view write quorums.

With the improvement in Section 4.8.2, DQA may allow transactions in two concurrent partitions to read and write the same data object as in GVP. However, the necessity to access the old coquorums in deflation makes this improvement difficult to realize.

4.8.4. Combining GVP and DQA

In view of the the advantages of having quorum assignment tables in DQA and the global read quorum set $RQ(X)$ in GVP, we propose here to combine the two ideas into one protocol. In this protocol, we shall call a quorum assignment table a DQA table. There will be a DQA table for each object **associated with** each GVP view V at each site in V. However, since at any time a site has only one GVP view, at any time only one table is being used at the site. Informally, each user transaction accesses each logical data object using the quorum assignment of a level in a DQA table associated with a GVP view for that object. We denote the table of logical object X associated with view V at site s by $TB(V, X, s)$. The quorums that appear in $TB(V, X, s)$ contains only copies of X that reside on sites in V.

There is a static global read quorum set $RQ(X)$ as in GVP. All write quorums that might exist in the quorum assignment table $TB(V, X, s)$ for any view V must intersect groupwise $RQ(X)$.

Initially, the system starts with an initial GVP view V_0 = all sites in the network, and $TB(V_0, X, s)$ is the initial DQA table associated with V_0 for each object X at site s. Before any failures, transactions submitted at s accesses X under the quorum assignment at level 1 of $TB(V_0, X, s)$. If a partition failure occurs, so that only quorum assignment at higher levels in $TB(V_0, X, s)$ can be used, then we apply "inflation" as in DQA on $TB(V_0, X, s)$.

If there is no level in the table of $TB(V_0, X, s)$ at which a transaction can be executed as a result of partition failures, then X may choose to do a view update as in GVP based on the view as projected by the partition.

Changing Views:

View update is done on a per object basis. The copy of each object X at site s keeps *current_Vid*(X, s), *high_Vid*(X, s), and *update_Vid*(X, s), which are initially all 0. In addition, each site s keeps *high_Vid*(s) which is the maximum of *high_Vid*(X, s) for all X's that have a copy at s. When the copy of X at s initiates a view-update, it creates a view with a unique view-id > $high\_Vid(s)$. If we imagine that each object copy resides on a virtual site, then the view-update transaction follows the view-update in GVP, with the addition that the DQA table associated with the view is also established. After view-update, the copy of X *might* send a message to inform every other copy of X in the new view that it should also adopt the new view. (This step only affects efficiency but not correctness.)

User Transactions

Each user transaction T at site s in view V executes read/write operations using quorums at a certain level l of the DQA table $TB(X,V,s)$. The same value of l and V will be used for operations of all data objects that T reads/writes. (T is said to execute at level l in view V.) The schedule for all transactions that execute at the same level in the same view must be 1C-serializable. To achieve this, either the timestamping mechanism in DQA or the mechanism in GVP using version number can be used.

Since a user transaction executed in view V with view-id V_id at site s may access more than one data object, say X and Y, if $current\_Vid(X,s) = V\_id > current\_Vid(Y,s)$, then Y must adopt the view $current\_Vid(X,s)$. Note that at this point, $current\_Vid(X,s) < high\_Vid(s)$ may hold.

When a transaction executes at level l of the table $TB(V,X,s)$, it follows the rules of both DQA and GVP.

(1) **Logical Read**: If multiple versions are used (the most up-to-date version for each level of $TB(V,X,s)$), then the procedure is the same as in DQA except that the read quorum accessed must reside on sites that has $current\_Vid = V\_id$. If only a single-version is used (the most up-to-date in all levels of $TB(V,X,s)$), then a logical read accesses a read quorum at level l and succeeds only if none of the accessed copies was last written by a transaction at a level $> l$. When a transaction with a read commits, each ratchet lock in the read quorum is set to the maximum of its current value and the transaction's level.

(2) **Logical Write**: If multiple versions are used, then the procedure is the same as DQA except for the following. If a logical write on X by a transaction executing in a view with view-id V_id accesses a copy of X at site s' and $V\_id < update\_Vid(X, s')$, then the logical write is aborted. This is analogous to the ratchet lock mechanism. If only a single version is used, then, in addition to the above requirements, none of the copies accessed should have been last written by a transaction at a level $> l$.

Quorum Inflation: A transaction executing at level l of $TB(V, X, s)$ that cannot access a write quorum because of failures or ratchet locks may abort and restart at a higher level in $TB(V, X, s)$.

Note that quorum deflation is possible but not necessary, since we can apply view-update instead. If indeed deflation of $TB(V, X, s)$ is performed, then it is very similar to deflation in DQA except that the coquorum of read or write is restricted to a set of copies within V.

In this combined strategy, the initial quorum assignment table may be fine-tuned to the most probable pattern of partition failures. The advantage of this strategy over GVP is that inflation (which incurs much less overhead than view-update) can be used when a partition failure creates partitions that contain quorums belonging to higher levels of the quorum assignment table. The advantage of this strategy over DQA is that deflation can be replaced by view-update, which we have seen in Section 4.8.3 to have some advantages over deflation. The extra restriction compared to DQA is the requirement that of each write quorum set in $TB(V, X, s)$ intersect groupwise the global read quorum set $RQ(X)$. It remains for future research to compare the performance of these systems by simulation.

4.8.5. Correctness of the GVP-DQA Combination

Theorem 4.2: Any schedule α, consisting of the operations of user transactions, generated by the GVP-DQA combination of Section 4.8.4 is equivalent to (i.e., has the same read-from relation as) a 1C-serial schedule σ.

Proof: Let $\{T_i^{GV} \mid 1 \le i \le m_v\}$ be the set of all transactions that are executed and committed in GVP view GV, that is, each T_i^{GV} submitted at site s uses the quorums at some level of $TB(GV, X, s)$ when it operates on logical data X. By the conditions of DQA, the schedule α_{GV} generated in GV is 1C-serializable.

All transactions that execute at the same level in the same view are 1C-serializable, and across different levels in the DQA table associated with a view, the DQA mechanism ensures that the resulting schedule is 1C-serializable. Hence, the schedule α_{GV} generated in GV is 1C-serializable.

Without loss of generality, let α_{GV} be equivalent to a serial schedule $\sigma_{GV} = T_1^{GV} T_2^{GV} ... T_m^{GV}$, a prefix of which consists of all view-update transactions in GV. We claim that α is equivalent to $\sigma = \sigma_{GV_1} \cdots \sigma_{GV_m}$, where m is the number of views and $GV_1\_id < GV_2\_id < ... < GV_m\_id$.

To prove this claim, we point out that all the conditions for GVP are enforced here if we imagine that each data object copy resides on a virtual site of its own. Hence the conditions for view-updates (e.g. assignments of new view-id), those for logical read (e.g. a read quorum must reside on sites with *current_Vid* = view-id of the transaction executed), and those for logical write are all satisfied (e.g., all the sites accessed must have *update_Vid* $\le$ view-id of the transaction executed). Therefore, any schedule generated by the combined protocol will be 1C-serializable. □

4.9. SEQUENCE VECTORS AND GLOBAL VIEWS

We define the **sequence vector** of a transaction T as follows: the sequence vector, Vn, of a transaction T is an array of integer **sequence numbers**, $Vn(X)$, one for each data object X that T operates on. If a data object Y is not operated on by T, then $Vn(Y)$ is undefined. The set of sequence numbers for a data object X, of the form $Vn(X)$ for some Vn, are totally ordered by the "less than" relation "<". We say that

$Vn(X)$ precedes (equals) $Vn'(X)$ if $Vn(X) <$ (=) $Vn'(X)$. Given two sequence vectors Vn and Vn' for two committed transactions, if there is a data object X such that both $Vn(X)$ and $Vn'(X)$ are defined and $Vn(X) < Vn'(X)$, then we say that Vn precedes Vn', written $Vn \ll Vn'$.

4.9.1. Total Ordering of Sequence Vectors in GVP and DQA

The sequence number of a data object corresponds to the view-id of a data object in GVP. Since a transaction always executes in a view in GVP, the sequence vector of a transaction always consists of sequence numbers which have an identical value (which is the view-id). We say that each view corresponds to a **global view**.

For DQA, a sequence number $Vn(X)$ corresponds to a "view of data object X", which is the quorum assignment at a certain level of a DQA table for the data object as defined in Section 4.8.1, and the "view-id"s can be regarded as the sequence number of X. Then a global view is a combination of views of different data objects. In order to show the uniquc ordering on all sequence vectors that can occur in an execution, we need to show the following.

Lemma 4.2: In any log generated by DQA, for any two sequence vectors Vn and Vn' of committed transactions T and T', respectively, if $Vn(X)$, $Vn'(X)$, $Vn(Y)$ and $Vn'(Y)$ are defined for different X and Y, then $Vn(X) < Vn'(X)$ implies $Vn(Y) \leq Vn'(Y)$.

Proof: Since each transaction executes at a certain level, all the levels are totally ordered by <, and since the "view-id"s are consistent with <, we need only consider the case where T and T' are executed at the same level l. If $Vn(X) < Vn'(X)$, then the quorum assignment corresponding to $Vn'(X)$ must have been bound to level l by a deflation transaction T_d after that corresponding to $Vn(X)$ was bound. Since T_d is atomic and it updates the quorum assignment table at a coquorum for both read and write operations of the previous assignment at level l, T must have been committed before T_d, which in turn must have been committed before T', so that the resulting schedule of user transactions and deflation transactions is 1C-serializable. Assume that $Vn'(Y) < Vn(Y)$. Then using an analogous argument, we conclude that T' must have been committed before T, a contradiction. □

It follows that the set of sequence vectors used by all committed transactions of a log generated by either GVP or DQA is totally ordered by the precedence relation $\ll$. A sequence vector is thus like a timestamp that helps us to schedule transactions.

4.10. UNIFYING GVP AND DQA

In this section, we derive a paradigm for any protocol that deals with partition failures by using the following principles, which have been extracted from GVP and DQA.

4.10.1. Common Features of GVP and DQA

Let Vn be the sequence vector of any transaction T.

(1) virtual partition: For all objects X accessed by T, $Vn(X) = v$ = view-id of the view (i.e. virtual partition) in which T executes. The set of sequence vectors of all committed transactions are totally ordered by the precedence relation $\ll$ defined in Section 4.9.

(2) quorum consensus: For each data object X, there is a quorum assignment *corresponding to* $Vn(X)$.

(3) any output schedule has an equivalent 1C serial schedule in which the transactions are serialized according to the precedence order $\ll$.

We have seen earlier that the set of sequence vectors used by all committed transactions of a log generated by either GVP or DQA is totally ordered by the precedence relation $\ll$.

4.10.2. Paradigm for General Quorum-Based Protocols

Consider two sequence vectors Vn and Vn' of two committed transactions in a log L generated by GVP or DQA. If $Vn \ll Vn'$ and there is no committed transaction in L with sequence vector Vn'' such that $Vn \ll Vn'' \ll Vn'$, then we say that Vn **immediately precedes** Vn' in L. For a given replicated data object X, let $rq(X,Vn)$ and $wq(X,Vn)$ denote the read quorum set and the write quorum set for X corresponding to sequence vector Vn, respectively. This notation is justified, since a sequence vector uniquely specifies a global view. The following are the criteria of our paradigm. We show that they are satisfied by GVP and DQA.

Criterion A: A correct quorum consensus based concurrency control protocol is used for concurrency control of all transactions with the same sequence vector. For any committed transaction with sequence vector Vn in a log L, and for each data object X, there is a unique **final** value of X denoted by $final(X,Vn)$.

The final value is defined recursively as follows:

(1) if X is written by a transaction with sequence vector Vn, then $final(X,Vn)$ is the unique value read by a fictitious final transaction with Vn that reads X after all user transactions with Vn have committed (as decided by the concurrency control protocol for the transactions with Vn), and before reading by transactions with Vn has been disabled by view-updates of higher view-id;

(2) if X is not written by a transaction with Vn, then $final(X,Vn)$ equals $final(X,Vn')$ where Vn' immediately precedes Vn in L; and

(3) $final(X, V_{-1})$, where V_{-1} is a fictitious sequence vector that immediately precedes the first sequence vector, equals the initial value of X.

Criterion B: When a transaction with sequence vector Vn reads X, and if no transaction with sequence vector Vn writing X has committed, then at least one copy in each read quorum corresponding to Vn has the **final** value $final(X,Vn')$, where Vn' immediately precedes Vn.

Hence the final value of a sequence vector is the "initial" value seen by transactions with the next higher sequence vector. One way to enforce Criterion B is to make a value read "final" by disabling further writes on the data object with preceding sequence vectors. Hence we can break this criterion into 2 parts as follows:

Criterion B1: When a transaction with sequence vector Vn reads X, at least one copy in each read quorum corresponding to Vn has the most up-to-date value written by transactions with sequence vectors $\ll Vn$, provided that no transaction with sequence vector Vn writing X has committed.

This can be satisfied in the following ways.

(1) The most up-to-date value written by a transaction with sequence vector Vn is stored at all copies in a write quorum corresponding to Vn. In GVP, each logical write executed in view V on data object X always increments the version numbers of the copies in a write quorum in $wq(X,V)$. A similar measure is taken in DQA. This makes sure that none of the copies in the write quorum which are updated by the most recently committed transaction with a logical write on it will be updated by a logical write operation executed in a view with a smaller view-id.

(2) During view-update of a site s in GVP, copies in a global read quorum in $RQ(X)$ are accessed for each X so that each copy of inheritable data object X at s gets the most up-to-date value from the previous view (since $RQ(X)$ intersects each write quorum in $wq(X,V)$ for all V). In step (2) of quorum deflation in DQA, the latest version is copied to all the copies in a new write quorum, which intersects each new read quorum. Hence the most up-to-date value is also obtained by operations extraneous to steps in executing user transactions.

(3) The alternate reading rule in GVP requires that a global read quorum in $RQ(X)$ be accessed, which intersects each write quorum in previous views. For DQA, each read quorum at level l intersects

each write quorum at all preceding levels in which writes are enabled (i.e., transactions may execute at such levels in the future).

Criterion B2: No transaction with sequence vector Vn' is allowed to write on X if, for some sequence vector Vn where $Vn' \ll Vn$, X is read by a transaction with Vn.

Write is disabled in two ways:

(A) *Implicit disable*[15]. If, by the quorum intersection rule or invariant, each write quorum in $wq(X,Vn')$ intersects each read quorum in $rq(X,Vn)$, then, when a reading operation accesses a read quorum in $rq(X,Vn)$, X can no longer be written by transactions with Vn'. For example, this happens when a ratchet lock disables writes while inflating in DQA, and the alternate reading rule in GVP requires that a global read quorum $RQ(X)$ be accessed, which intersects each view write quorum of X.

(B) *Explicit disable.* If some write quorums in $wq(X,Vn')$ do not intersect some read quorums in $rq(X,Vn)$, then we must explicitly disable writes on X by transactions with Vn'. One way is to access a set S of sites that contains enough copies to intersect each write quorum in $wq(X,Vn')$ and *inform* each site s in S that write is now disabled. Such a method may disable writes even if X has not yet been read by any transaction with sequence vector Vn.

For example, in GVP, data object X is inheritable in view V only if V contains a global read quorum in $RQ(X)$. Since the global read quorum intersects each previous write quorum, once a data object is initialized by the view-update transaction, writing initiated at an old view cannot proceed because it will come across a copy at a site s with *update_Vid*(s) higher than the old view. In steps (1) and (2) of quorum deflation in [Her87], *ratchet locks* are advanced to inhibit writing, where appropriate.

[15] By implicit disable, we refer to the disabling of writes by the normal steps of read and write in quorum consensus. By explicit disable, we refer to extra steps that are not part of the usual quorum consensus, such as view-update transactions and deflation transactions.

In step (3) of quorum deflation, the assignment tables at a coquorum of an old write quorum are updated, so that no transaction can write with the old write quorum.

Note that any logical read $R_T[X]$ accesses a read quorum rq of X, and any logical write $R_W[X]$ accesses a write quorum wq of X, and we often require that rq intersects wq. This intersection property of each read quorum with each write quorum of a data object achieves two objectives. First, $R_T[X]$ can always read the most up-to-date value because the read quorum that $R_T[X]$ accesses has at least one copy in common with the write quorum of copies written by the most recent logical write. Second, the read quorum that $R_T[X]$ accesses intersects each write quorum to be written by each logical write that follows the read, so that the $R_T[X]$ can inhibit writes by marking (e.g., by means of ratchet locks in [Her87]) the copies in the read quorum if necessary.

4.11. CONCLUDING REMARKS

We have defined a family of virtual partition protocols, GVP, which generalizes and improves on the virtual partition protocol VP [ElT89] by allowing two concurrent partitions to read and write the same logical data under certain conditions. We proved the correctness of the members of GVP. We have also considered three members of GVP: VP, SP and FP, each having its own characteristics and suitable for different applications. The optimization methods, such as *demand tracking* [ElT89], are also applicable to the members of GVP.

A replica control protocol interacts to some degree with concurrency control protocol. For example, if **2-phase locking** is used together with GVP, then we must specify what a view-update transaction should do when it accesses a data object copy locked by a transaction. The transaction that is holding a lock on it may now be in a separate partition as a result of a partition failure. Such details have been worked out.

We compared GVP with the dynamic quorum adjustment (DQA) in [Her87], and extracted some common principles and mechanisms shared by both.

One way to compare VP, SP, and FP would be to compute the expected data availability under a given distribution of partition failures. Such analysis is left for the future.

CHAPTER 5

BICOTERIES

5.1. MOTIVATION

As we have seen in our review on replicated database systems, *quorum consensus* is one major approach to concurrency control. However, there has not been much study on how to select quorums to implement the quorum consensus approach. We see from the previous chapter that the characteristics of protocols like VP, SP and FPP depend very much on what quorums are used. It would be useful to distinguish good sets of quorums from the rest and this would require a study of the properties of quorums. In this chapter, we make such an attempt by relating sets of read/write quorums to an abstract structure which we shall call a **bicoterie**.

Among the first solutions proposed to the concurrency control problem based on quorum consensus was the *vote assignment* or majority voting technique [Gif79,Tho79], where each copy of each data object X is assigned a number of *votes*, and a logical operation on X is allowed only if it can lock enough copies that contain a majority of votes. As we have seen in previous chapters, vote assignment is a special case of quorum consensus, where a quorum is a set of copies which contains a certain number of votes. This idea was first introduced in [Lam78a], where each logical data is assigned a set of *groups* of copies. Each pair of groups of copies should intersect at at least one copy. As a simple example, suppose we have 3 copies $\{1, 2, 3\}$ of a data object X. $\{ \{1, 2\}, \{2, 3\}, \{1, 3\} \}$ is a set of groups with the above intersection property. An operation on X is allowed only when either both 1, 2 are locked, or both 2, 3 are locked, or both 1, 3 are locked.

In [GaB85], a set of groups of copies such as $\{ \{1, 2\}, \{2, 3\}, \{1, 3\} \}$ in the above example is called a **coterie** (see Definition 5.1 for a precise definition). Essentially, a coterie P is a set of groups of elements, such that each group in P intersects each other group in P. The intention is to use P to model the read quorum set and the write quorum set in quorum consensus.

In the above example, the conditions for permitting operations are equivalent to a vote assignment that assigns a vote of one to each copy. It is shown in [GaB85] that some vote assignment can be improved if the total number of votes is even. The following example is given. Suppose we have a system with copies a, b, c, and d of X, and a vote of one is assigned to each copy. Then three copies are needed for a majority, and the assignment is equivalent in effect to the following coterie:

$$S = \{ \{a,b,c\}, \{a,b,d\}, \{a,c,d\}, \{b,c,d\} \},$$

which is the set of all groups of three copies.

Consider now the following coterie:

$$R = \{ \{a,b\}, \{a,c\}, \{a,d\}, \{b,c,d\} \}.$$

Each group still intersects each other group, but now three of the groups have one less element than before. This means that to achieve mutual exclusion, one less element need to be locked in three of the four choices. This is a better solution than the vote assignment in terms of both overhead of logical read/write and availability.

In [GaB85], R is said to "dominate" S (see Definition 5.2). There is another way that a coterie "dominates" another. Consider a logical data X with three copies $\{a, b, c\}$ and the coterie

$$S = \{ \{a,b\}, \{b,c\} \}.$$

We observe that the following coterie is an improvement over S in terms of availability:

$$R = \{ \{a,b\}, \{b,c\}, \{a,c\} \}.$$

The reason is that we have one more group to choose from. So in case we have a partition failure where we have copies a and c in one partition, we will be able to operate on X with coterie R, but not with S. (In terms of overhead, R is a bigger set and it would require more memory to store R, but we shall consider improving availability a more important objective.)

Therefore, if a coterie R "dominates" another coterie S, then it is an improvement in availability if R instead of S is taken as the set of read/write quorums in quorum consensus.

Let us define more formally the terms coteries and domination.

Definition 5.1: [GaB85] Let U be the set of copies (sites)[16] that compose the system. A set S of groups of elements from U is a **coterie** under U iff

(1) $G \in S$ implies that $G \neq \phi$.

(2) (**Intersection property**) If $G, H \in S$, then G and H must have at least one common element.

(3) (**Minimality**) There are no $G, H \in S$ such that $G \subset H$. □

Definition 5.2: Let R, S be coteries (both under U). R **dominates** S if $R \neq S$ and, for each $H \in S$, there is a $G \in R$ such that $G \subseteq H$. □

Definition 5.3: A coterie S (under U) is **dominated** if there is another coterie (under U) which dominates S. If there is no such coterie, then S is **nondominated** (ND, for short). □

The time complexity of the problem of deciding if a given coterie is dominated has been left as an open problem by [GaB85]. (It is believed to be NP-complete (see Section 5.5).) There is a brute-force algorithm for NDness, which tries to reduce the size of each group in the given coterie P without violating the intersection property, as well as generates all possible groups that intersect all groups in P and see if each one can be added to the given coterie without violating the minimality property. The time complexity of this algorithm is in general exponential in the input size.

[16] In GVP, data object copies are considered, while in some other protocols, e.g., VP in [ElT86], sites are considered. We shall use the term "elements" to denote either data object copies or sites.

However, as shown by M.Yannakakis in [GaB85], there are at least $2^{2^{cn}}$ nondominated coteries under a universe of n elements for some constant c. Therefore, it is not enough to know how to distinguish between a dominated coterie and a nondominated coterie. The more practical and also challenging problem is how to choose "good" nondominated coteries from among the more than $2^{2^{cn}}$ nondominated coteries. We leave this problem for future research. Instead, we would like to address another issue that is not attended to in [GaB85].

In the most general setting of the concurrency control problem, we assume only two conflicting operations on data: read operation and write operation. (With concurrency control for *abstract data type* [Her86, HeW88], other kinds of operations can be defined with corresponding properties of conflicts.) This is why in the quorum consensus protocol (Chapter 4), we have a read quorum set and a write quorum set. By defining only a coterie, [GaB85] has actually restricted the solutions to those that use identical read quorum set and write quorum set. However, we have quorum consensus protocols that make use of different read quorum set and write quorum set. One example is the read-one write-all protocol, where the read quorum set consists of all groups of one element, and the write quorum set consists of the single group that contains all elements.

Therefore, we shall generalize the concept of coterie to that of **bicoterie**. A bicoterie is essentially a pair of sets $\{P, Q\}$, of groups of elements, where each group of elements in P intersects each group of elements in Q. The intention is to model the read quorum set and write quorum set by P and Q, respectively.[17] A coterie is then equivalent to a bicoterie $\{P, Q\}$ such that $P = Q$.

[17] Imagine that instead of two conflicting operations of read and write, we have 3 operations Z_1, Z_2, and Z_3 that conflict with each other, which require quorums (groups of elements) in Q_{Z_1}, Q_{Z_2}, and Q_{Z_3}, respectively. Then we require that each group in Q_{Z_i} to intersect with each group in Q_{Z_j}, where $j \neq i$. Hence we have something that we may call a *tri-coterie*. This could be generalized to N operations, whence we shall have *N-coteries*. It could also be generalized to cases where the conflicts are between certain pairs of operations instead of any pairs of operations. A boolean relation then maps each pair of operations to true (they conflict) or false (they do not conflict).

In the following sections we shall give more formal definitions and prove some properties of bicoteries. We shall define domination of bicoteries similar to that of coteries, and we shall relate the results to the virtual partition protocols in the previous chapter.

5.2. DEFINITION AND PROPERTIES OF BICOTERIES

Definition 5.4: Let U be a set of elements and let $A = (P, Q)$ be an ordered pair of sets of groups of elements from U. A forms a **bicoterie** under U if

(1) for each group G in P or Q, $G \neq \phi$,

(2) (**Intersection property**) for each group G in P and each group H in Q, $G \cap H \neq \phi$.

(3) (**Irredundancy**) for any two groups G and $H \in P$, $G \not\subset H$, and for any two groups G and $H \in Q$, $G \not\subset H$. □

Definition 5.5: A bicoterie $A = (P, Q)$ is **dominated** by bicoterie $B = (R, S)$ (or B **dominates** A) if the following hold:

(1) for every group G in P, there exists a group H in R such that $H \subseteq G$,

(2) for every group G in Q, there exists a group H in S such that $H \subseteq G$, and

(3) $(P, Q) \neq (R, S)$. □

The following lemma shows that whenever a bicoterie $A = (P, Q)$ is dominated, we can find a bicoterie $B = (R, S)$ dominating A, where either $R = P$ or $S = Q$.

Lemma 5.1: For a bicoterie $A = (P, Q)$, if there exists a bicoterie $B = (R, S)$ dominating A and such that $P \neq R$ and $Q \neq S$, then

(i) (P, S) is a bicoterie dominating A.

(ii) (R, Q) is a bicoterie dominating A.

Proof: (i) By definition, each group in R intersects each group in S. Since each group G in P has a subset H in R, and H intersects each group in S, G also intersects each group in S. Hence, (P, S) is a bicoterie. It dominates (P, Q) because (R, S) dominates (P, Q).

Similarly for (ii). □

The following theorem, which can be proved as Theorem 2.1 in [GaB85], will provide us with a way to show whether a bicoterie is dominated or not.

Theorem 5.1 A bicoterie $A = (P, Q)$ under U is dominated iff there exists a group G $(\subset U)$ such that

(i) G is not a superset[18] of any group in P and G intersects every group in Q; or
(ii) G is not a superset of any group in Q and G intersects every group in P

Proof: First we prove that the existence of G satisfying (i) or (ii) implies that A is dominated. Suppose there is a group G satisfying (i). If G is a subset of some groups $H_1, H_2, \cdots H_n$ in P, then ($\{P - \{H_1, H_2, ..., H_n\}\} \cup \{G\}, Q$) dominates A. Otherwise ($P \cup \{G\}, Q$) dominates A. Similarly for (ii).

Second, we prove that A is dominated implies (i) or (ii). From Lemma 5.1, A is dominated either by bicoterie $B = (R, Q)$ or bicoterie $C = (P, S)$, where $R \neq P$, $S \neq Q$. Suppose A is dominated by $B = (R, Q)$. Since $R \neq P$, we must have either of the following:

(1) there exists G in R such that $G \not\subseteq H$ and $H \not\subseteq G$ for any H in P, or

(2) there exists G in R and H in P such that $G \subset H$.

[18] By definition, a set is a superset of itself.

In either case, G must satisfy (i) or else B would not be a bicoterie.

Similarly if A is dominated by $C = (P, S)$, except that (ii) instead of (i) will be satisfied □

We say that A is **dominated** by G in the above theorem.

Definition 5.6: A **transversal** of a set of groups P is a group which intersects each group in P. A **minimal transversal** of P is a transversal of P such that no proper subset of it is a transversal of P. □

If M and N are 2 sets of minimal transversals of P, then clearly $M \cup N$ is also a set of minimal transversals of P.

Lemma 5.2 (cf. Lemma 2.2 in [GaB85]): Let $A = (P, Q)$ be an ND bicoterie and G a transversal of P (Q), then G is a superset of a group in Q (P).

Proof: If G is not a superset of a group in Q (P), then G can be added to Q (P) and we get a bicoterie that dominates A. □

Definition 5.7: $MT(P)$ denotes the set of all minimal transversals of P. □

Now we can define a way to check for domination of a given bicoterie in terms of minimal transversals.

Theorem 5.2: A given bicoterie $A = (P, Q)$ is ND iff $P = MT(Q)$ and $Q = MT(P)$.

Proof: Follows directly from Theorem 5.1. □

5.2.1. Modeling by Boolean Functions

In order to prove properties of the transformation MT formally, we represent a set of groups by a boolean expression. Let universal set $U = \{ g_1, g_2, ..., g_n \}$, and consider each element of U as a boolean variable. Any group G of elements from U can be uniquely represented as the boolean product term of the elements of G. Let $P = \{ G_1, G_2, ... G_m \}$, where $G_i = \{ g_{i1}, g_{i2}, ..., g_{ik_i} \}$, be a set of groups of elements of U. Group G_i is thus represented by product $PD_i = g_{i1} \cdot g_{i2} \cdot \ldots \cdot g_{ik_i}$. (We also say that G_i is the group corresponding to PD_i.) We now represent P as a sum-of-products expression $P = PD_1 + PD_2 + \ldots + PD_m$. Observe that this representation is unique. Namely, for any set P of groups of elements of U, there is a unique positive[19] boolean sum-of-products expression, and conversely. Applying formula $A \cdot A = A$ or $A + AB = A$ to a boolean expression E is called a **reduction** step (E gets **reduced** by such a step). A sum-of-products boolean expression is said to be **irreducible** if it cannot be reduced further. Clearly an irreducible positive sum-of-products represents an irredundant set of groups.

Two boolean expressions are said to be **equivalent** if they represents the same function.

Lemma 5.3: For any given positive boolean expression E, there is a unique irreducible sum-of-products expression E' equivalent to E up to a permutation of product terms.

Proof: Suppose there are two distinct irreducible sum-of-products expressions E_1 and E_2 equivalent to E. Clearly both E_1 and E_2 are positive, since applying $A \cdot A = A$ or $A + AB = A$ to E does not generate any complemented variable. Let PD be a "smallest" product term (i.e., a product term consisting of the smallest number of variables) that is in one of E_1, E_2 but not in the other. Without loss of generality let PD be a product term in E_1. Assign value 1 to all variables in PD and 0 to all other variables, and evaluate E_1 and E_2. For this assignment E_1 yields 1 while E_2 yields 0, a contradiction to the fact that E_1 and E_2 are equivalent. □

In boolean algebra, the **dual** of a boolean expression E (denoted by *dual*(E)) is obtained from E by interchanging the OR (+) and AND (·) operations and the constants 0 and 1, respectively.

Let P be an irredundant set of groups as defined in Section 5.2 and let E be an irreducible boolean sum-of-products expression representing P. We now interpret the transformation $MT(P)$ in terms of operations on boolean expressions.

[19] A boolean expression is positive if no complemented variable appears in it.

Consider the dual of E, i.e., $(g_{11}+g_{12}+\cdots+g_{1k_1})\cdot(g_{21}+g_{22}+\cdots+g_{2k_2})\cdot\cdots\cdot(g_{m1}+g_{m2}+\cdots+g_{mk_m})$. If we expand this expression, a sum-of-products expression SP results. Note that each term in SP contains one element from G_i, $i=1,\ldots,m$. Therefore, the group corresponding to any product in SP intersects group G_i for any i. Now reduce SP using the formulae $A\cdot A=A$, and $A+A\cdot B=A$. Intuitively, the reduction using $A\cdot A=A$ corresponds to getting sets of elements instead of bags[20] of elements. Applying $A+A\cdot B=A$ corresponds to obtaining the irredundant set of groups. Making the above arguments more formal, we can prove the following lemma.

Lemma 5.4: Let E be an irreducible positive boolean expression representing an irredundant set P of groups under a universal set U. Let $R=red(dual(E))$, i.e., the irreducible sum-of-products expression obtained from $dual(E)$ by reducing $dual(E)$. Then each term in R represents a set in $MT(P)$ and vice versa. □

The above relation between duality and minimal transversal leads to a proof for the following theorem, which will in turn lead to a way to derive some ND bicoteries that dominate a given non-ND bicoterie.

Theorem 5.3: If P is an irredundant set of groups then $P=MT(MT(P))$.

Proof: We borrow the *principle of duality* ([Har65] p.56, [Koh78] p.45) from boolean algebra, which says that the dual of a valid boolean statement is also valid. (The principle of duality stems from the symmetry of the postulates and definition of boolean algebra with respect to the two operations, + and $\cdot$, and the two constants, 0 and 1.) Let $MT(P)=Q$, and EP and EQ be the irreducible boolean expressions representing P and Q, respectively. From Lemma 5.4, $MT(P)$ is represented by $red(dual(EP))$. Since $MT(P)=Q$, we have $red(dual(EP))=EQ$. The equality here means that both

[20] A bag is a multiset. Thus a bag of elements may contain more than one sample of an element of the same identity.

sides are identical up to a permutation of product terms. Since *red* preserves equivalence of expressions, $dual(EP)$ and EQ are equivalent. By the principle of duality, therefore, it follows that EP is equivalent to $dual(EQ)$. Finally, by Lemma 5.3, we have $EP = red(dual(EQ))$. Substituting $MT()$ for $red(dual())$ and sets of groups for boolean expressions in the above boolean statements, we obtain $P = MT(Q) = MT(MT(P))$. □

Corollary 5.1: For each irredundant set P of groups (see Def. 5.4), there is an ND bicoterie (P, Q). This exhausts all ND bicoteries.

Proof: The first part follows directly from Theorem 5.3 since $(P, MT(P))$ is an ND bicoterie. By definition, for any bicoterie $A = (P, Q)$, P is an irredundant set of groups. Hence all irredundant sets of groups exhaust all possible choices for P. By Theorem 5.2, A is ND iff $P = MT(Q)$ and $Q = MT(P)$. Suppose (P, Q_1) and (P, Q_2) are both ND, where Q_1 and Q_2 are distinct sets of groups. This contradicts Lemma 5.2, which says that $MT(P)$ is unique for P. □

Corollary 5.2 A given bicoterie $A = (P, Q)$ is ND iff $P = MT(Q)$ or $Q = MT(P)$.

Proof: This follows from Theorems 5.2 and 5.3. □

The following is a way to find an ND bicoterie that dominates a given bicoterie,

Given a non-ND bicoterie $A = (P, Q)$,

Choice (a): Find $(P, MT(P))$.

Choice (b): Find $(MT(Q), Q)$.

We now show that either of the above choices indeed finds an ND bicoterie that dominates A. For Choice (a): By Theorem 5.3, $(P, MT(P))$ is ND. By Lemma 5.2, each group in Q is a superset of a set in $MT(P)$. Therefore, for each group G in Q, there exists a group H in $MT(P)$ which is a subset of G, i.e., A is dominated by $(P, MT(P))$.

Similarly for Choice (b).

Example 5.1: Let $A = (P, Q)$ be a non-ND bicoterie, where $P = \{ \{a, b\}, \{a, c\} \}, Q = \{ \{a\} \}$. With Choice (a), we get $B = (P, S)$ where $S = \{ \{a\}, \{b, c\} \}$, and with Choice (b), we get $C = (R, Q)$ where $R = \{ \{a\} \}$. $\square$

The above method does not derive all possible ND bicoteries that dominate a given non-ND bicoterie. This can be seen from the following example.

Example 5.2: Consider a non-ND bicoterie $A = (P, P)$ under a universe $U = \{ 1, 2, 3, 4, 5, 6, 7 \}$, where P consists of groups represented by rows of the following array.

1	2	3
1	4	5
1	6	7
2	5	6
3	4	6
3	5	7

Using either Choice (a) or (b), we first derive $Q = MT(P) =$

1	2	3
1	4	5
1	6	7
2	5	6
3	4	6
3	5	7
1	3	5
2	4	7
1	3	6
1	5	6
3	5	6

Then $B = (P, Q)$ or $C = (Q, P)$ dominates A.

However, the ND bicoterie $D = (R, R)$ where R is shown below, also dominates A:

1	2	3

1	4	5
1	6	7
2	5	6
2	4	7
3	4	6
3	5	7

and D is not derived by either choice (a) or (b). □

Theorem 5.4 A bicoterie $A = (P, P)$, where P is a coterie, is ND iff P is ND.

Proof: This follows from the fact that a coterie P is ND iff $P = MT(P)$. □

5.3. FPP BICOTERIES

In Section 4.6.4, we defined *finite projective plane (FPP)* and introduced the Finite Projective Plane Based Protocol (FP), in which the lines (sets of points or elements) on a finite projective plane are taken as the read/write quorums. Here we shall define a class of bicoterie based on FPP.

Definition 5.8. Let P be a set of lines (groups)[21] in a finite projective plane of order m. The coterie under a universal set U of $m^2 + m + 1$ elements corresponding to P is called a **FPP coterie of order** m. A bicoterie $A = (P, P)$ under the universal set U is called a **FPP bicoterie of order** m

From Theorem 5.4, a bicoterie composed of two ND coteries is ND. Therefore, to show that a FPP bicoterie $A = (P, P)$ is ND, we need only to show that the FPP coterie P is ND.

Theorem 5.5: A FPP bicoterie of order m is not dominated by any line (group) with $\leq m + 1$ points (elements).

Proof: In a FPP Π of order m, each point lies on exactly $m+1$ lines, and each line contains exactly $m+1$ points (Theorem 4.1). If the FPP coterie Π is dominated, then there exists an element (a point) in every group (line) which together form a group (line) L_s which is not a superset of any group (line) in P. Let us find the minimal number of points on such a line L_M. Each point intersects $m+1$ lines, and there

[21] With FPP bicoteries, lines (sets of points) of a FPP corresponding to (sets of elements) in a coterie. We

are in total $m^2 + m + 1$ lines (Theorem 4.1). Therefore, there must be at least $\lceil (m^2+m+1)/(m+1) \rceil = m+1$ points on L_M.

It remains to show that a line containing $m+1$ points cannot dominate Π. Assume that a line L_M containing $m+1$ points dominates Π. Since any two points must lie on exactly one line in Π, there are at least two points x_1 and x_2 common to lines L_k and L_M for some line L_k in Π. Since L_M is not equal to any existing line in Π, there is at least one point y on line L_k which is not on L_M. Since Π is a FPP of order m, y lies on $m+1$ lines, therefore, there are lines $L_1, ..., L_m$ in addition to L_k, which pass through y. Each of these lines must have at least one point in common with L_M, the point must be distinct for each line, and the point cannot be x_1 or x_2 (else the line becomes L_k). However, there are only $m+1$ points on L_M and we cannot find such a distinct point for each of $L_1, ..., L_m$. We conclude that our assumption cannot be true. □

Theorem 5.6: A FPP coterie of order m may be dominated by groups (lines) with more than $m+1$ points.

Proof: We prove by giving an example of a FPP coterie of order 3. Suppose the elements (points) of a FPP coterie are denoted by numbers 1 to 13, and let the groups (lines) in the FPP coterie be:

1	2	3	4
1	5	6	7
1	8	9	10
1	11	12	13
2	5	9	13
2	8	12	7
2	11	6	10
3	5	12	10
3	8	6	13
3	11	9	7
4	5	8	11

sometimes use the terms, lines and groups, interchangeably.

4	6	9	12
4	7	10	13

Then the above FPP is dominated by the group (line) { 5, 6, 7, 2, 8, 11 }. □

Theorem 5.7: A FPP coterie of order 2 is ND.

Proof: Let Π_2 be a FPP coterie of order 2. From Theorem 5.5, any line dominating Π_2 has more than 3 points. Suppose Π_2 is dominated by a line L_d of 4 points. Since every two points on a FPP lie on a line, there are $\binom{4}{2} = 6$ lines that have two points on L_d and one point outside. Since Π_2 has 7 lines, the remaining line must intersect with L_d at only one point and intersect at two points with lines other than L_d. There are 3 points outside L_d. From the above discussion, each of 6 lines goes through exactly one of these points, and one line goes through two points. The sum of *degrees* (i.e., the number of distinct lines that pass through a point) of these points is therefore 8. However, for Π_2, each point should lie on 3 lines, and the sum of degrees of 3 points should be 9, a contradiction. It is not possible for a line containing y points, $y > 4$, to dominate Π_2 because the number of lines that it intersects at two points would be $\binom{y}{2}$ which is greater than 7 for $y > 4$. □

5.4. VOTE ASSIGNMENTS

In Section 4.6.1, we introduced the vote assignment protocol (VAP), in which the read/write quorums were sets of copies with a total number of votes ≥ read/write threshold. Here we define a more general vote assignment independent of "views" of GVP.

Definition 5.9: Let U be the set of copies of a logical data object X. A **vote assignment** is a function $v : U \rightarrow N$. (N is the non-negative integers.) $v(a)$ denotes the number of votes assigned to copy a.
□

Definition 5.10: For a vote assignment v over U, $TOT(v)$ is defined by

$$TOT(v) = \sum_{a \in U} v(a). \ \square$$

Definition 5.11: For a vote assignment v over U, we call $RT(v)$ and $WT(v)$ a **read accessibility threshold** and a **write accessibility threshold**, respectively, if they satisfy $RT(v) + WT(v) = TOT(v) + 1$. $\square$

Definition 5.12: Let v be a vote assignment over U. We say that v **can realize** bicoterie (P, Q) if there are read and write accessibility thresholds, $RT(v)$ and $WT(v)$, respectively, such that P and Q are the minimal groups[22] of P', Q', respectively, where

$$P' = \{ G \subset U \mid \sum_{a \in G} v(a) \geq RT(v) \} \text{ and}$$

$$Q' = \{ G \subset U \mid \sum_{a \in G} v(a) \geq WT(v) \}.$$

$\square$

Definition 5.13 (cf. Definition 3.3. in [GaB85]): We say that a vote assignment v over U **can realize** a coterie Z if there is a value $TH(v)$, $MAJ(v) \leq TH(v) \leq TOT(v)$[23], where $MAJ(v)$ is $\lceil (TOT(v)+1)/2 \rceil$, such that Z consists of the minimal groups of Z', where

$$Z' = \left\{ G \subseteq U \mid \sum_{a \in G} v(a) \geq TH(v) \right\}.$$

$\square$

In general, a given bicoterie may be realized by more than one vote assignment. For example, vote assignment v, $v(a) = 1$, $v(b) = 1$, $v(c) = 1$, with $RT(v) = WT(v) = 2$, as well as vote assignment v',

[22]Namely, $G \in P'$ (Q') if G contains no other group of P (Q). $\{(P, Q)\}$ is a bicoterie since it satisfies the properties of Definition 5.4.

[23] In [GaB85], $TH(v)$ is fixed to be $MAJ(v)$.

$v'(a) = 2$, $v'(b) = 2$, $v'(c) = 3$, with $RT(v') = WT(v') = 4$ realize the same bicoterie.

The following theorem shows that some ND bicoteries may not be realized by any vote assignment.

Theorem 5.8: No vote assignment can realize a FPP bicoterie of order $m \geq 2$.

Proof: Each FPP bicoterie A of order m (≥ 2) consists of two identical coteries P whose groups correspond to the lines in the corresponding FPP. In general, if $G = \{ a, b, c \} \cup G'$ is in coterie P, then $H = \{ a', b, c \} \cup G'$ is not in P. This is because any two elements (points) appear together in one and only one group (line), hence b, c cannot appear in two groups (lines) in P.

If the coterie can be realized by a vote assignment v, then we must have $v(a) > v(a')$, so that $\sum_{x \in G} v(x) > \sum_{x \in H} v(x)$, and it is possible for the first sum to be $\geq$ RT(v) or WT(v) while the second sum is not.

Since each point in the FPP of P lies on $m+1$ lines, there must be a line (or group) to which a' belongs, hence there exists a group $J = \{a', d, e\} \cup J'$ in P. Then $K = \{ a, d, e \} \cup J'$ cannot be in P, for the same reason as above. However, $v(a) + v(d) + v(e) + \cdots > v(a') + v(d) + v(e) + \cdots$ and it is not possible for the total vote of J to exceed a threshold value while that of K does not, a contradiction. □

Theorem 5.9: There are ND bicoteries that cannot be realized by any vote assignment.

Proof: From Theorem 5.7, FPP bicoteries of order 2 are ND, and from Theorem 5.8, such bicoteries cannot be realized by vote assignment. □

Theorem 5.10: Let $A = (P, Q)$ be a bicoterie under U realized by a vote assignment v, with read and write accessibility thresholds of $RT(v)$ and $WT(v)$. Then A is nondominated.

Proof: From Lemma 5.1, we can assume without loss of generality that A is dominated by $B = (P, S)$. By Theorem 5.1, there is a group $G \in S$ that intersects all groups in P but is not a superset of a group in Q. Since $G \notin Q$, by definition, G has less than $WT(v)$ votes. Since $RT(v) + WT(v) = TOT(v) + 1$, $U - G$ must have more than $RT(v)$ votes. Thus, $U - G$ or a subset of it must be in P. This is a contradiction since G does not intersect $U - G$ or its subsets, and hence does not intersect all groups in P. $\square$

In [GaB85], $TH(v)$ is always set to be $MAJ(v)$, both read and write must gather a minimum vote of $TH(v) = TOT(v)/2 + 1$ when $TOT(v)$ is even. $TH(v)$ cannot be used as $RT(v)$ and $WT(v)$ because then $RT(v) + WT(v) \neq TOT(v) + 1$. Theorem 3.3 in [GaB85] says that in this case, if vote assignment is augmented to v' so that $TOT(v')$ is $TOT(v) + 1$, then the resulting vote assignment is nondominated. With Theorem 5.10, we can explain this fact because v' is a vote assignment and $RT(v)=TH(v)$ and $WT(v)=TH(v)$ satisfy $RT(v) + WT(v) = TOT(v) + 2 = TOT(v') + 1$.

5.5. RECTANGULAR BICOTERIES

In the read-one write-all approach in replicated database concurrency control, implementing logical read is cheap but implementing logical write is costly. There is a trade-off between the overheads of logical read and logical write. With the read-one write-all approach, the relative overheads of the two operations are 1 to n, where n is the number of copies. Now we want to generalize this ratio to c to $\lceil n/c \rceil$, for any integer constant $c \leq n$. To this end, we now introduce a family of **rectangular bicoteries**. A rectangular bicoterie which realizes a trade-off factor of c to n/c between read and write will be called a rectangular bicoterie **of order** c.

In general, if we have a set of data copies $U = \{ a_1, a_2, a_3, ..., a_n \}$, then a rectangular bicoterie $A = (P, Q)$ of order c is constructed as follows:

$P = \{ \{ a_{i+1,} a_{i+2}, \cdots a_{i+c} \} \mid i = 0, c, ..., k \cdot c \}$, where $k = \left(\lceil n/c \rceil - 1 \right)$,

where if n is not a multiple of c, then the elements a_j, where $j > n$, in the above are taken as the null element, i.e., the last group (with $i = k \cdot c$) in P has less than c elements. Q is defined by

$Q = MT(P)$.

Since P is a set of disjoint groups, Q is the set of all combinations of choosing exactly one element from each group of P. That is, if $P = \{ G_1, G_2, ..., G_{\lceil n/c \rceil} \}$, then $Q = \{ \{ b_1, b_2, ..., b_{\lceil n/c \rceil} \} \mid b_i \in G_i \}$. As an example, if $U = \{a_1, a_2, a_3, a_4, a_5\}$, then a rectangular bicoterie of order 2 is $A = (P, Q)$, where $P = \{ \{ a_1, a_2 \}, \{ a_3, a_4 \}, \{ a_5 \} \}$ and $Q = MT(P) = \{ \{ a_1, a_3, a_5 \}, \{ a_1, a_4, a_5 \}, \{ a_2, a_3, a_5 \}, \{ a_2, a_4, a_5 \} \}$.

Q has at least $c^{\lfloor n/c \rfloor}$ groups of $\lceil n/c \rceil$ elements each.

Theorem 5.11: Each rectangular bicoterie is non-dominated.

Proof: If $A = (P, Q)$ is a rectangular bicoterie, then Q is constructed from P by $Q = MT(P)$, hence by Corollary 5.2, A is nondominated. □

Theorem 5.12: No vote assignment can realize a rectangular bicoterie.

Proof: Let $A = (P, Q)$ be a rectangular bicoterie of order c, and let groups $G_1 = \{ a_1, a_2, ... a_c \}$, $G_2 = \{ b_1, b_2, ... b_c \}$ be groups of P. Assume that A is realized by a vote assignment v. If $v(a_1) \geq v(b_1)$, then the group $H = \{ a_1, b_2, ..., b_c \}$ would have at least as many votes as G_2, i.e., at least $RT(v)$ votes, and hence a subset H' of H should be a group in P. H' cannot contain a_1 since no element appears in two groups in P by definition. But then $H \subset G_2$, and thus P is not irredundant, a contradiction. □

For a set P of groups of elements, let the **size** of P be the sum of the number of elements in each group of P. By definition of a rectangular bicoterie $A = (P, Q)$ under U, we see that size of P is equal to n, the number of elements in U, whereas the size of Q is exponential in n. Hence it will be expensive to remember the groups of Q by storing them in memory. One solution would be to store only P and to derive groups in Q algorithmically. In quorum consensus, it is enough if one group in Q is accessible (e.g., within a partition during a partition failure). The algorithm that searches for any one group in Q given P runs in linear time because it only has to check if there exists an accessible element in each group of P.

5.6. COUNTING BICOTERIES AND VOTE ASSIGNMENTS

Theorem 5.13 (M.Yannakakis, Theorem 4.1 in [GaB85]): There are at least $2^{2^{cn}}$ ND coteries under a universe of n elements, where c is some constant.

Corollary 5.3: The above value is a lower bound on the number of ND bicoteries under a universe of n elements.

Proof: It follows from the above theorem and from Theorem 5.4, which says that P is a ND coterie iff (P, P) is a ND bicoterie. □

Theorem 5.14: The following formula gives a lower bound on the number of ND bicoteries under a universe of n elements.

$$2^{\binom{n}{1}} + 2^{\binom{n}{2}} + \cdots + 2^{\binom{n}{\lceil (n+1)/2 \rceil}} + \cdots + 2^{\binom{n}{n}} - n.$$

Proof: From Theorem 5.3, the number of ND bicoteries under n elements equals the number of irreducible sum-of-products boolean expressions formed from a set of n positive literals. We count the number of sum-of-products boolean expressions where all products contain exactly the same number of positive literals. There are $\binom{n}{i}$ different products of exactly i positive literals, and each sum-of-products expression has the choice of including or not including each of these products. Hence there are $2^{\binom{n}{i}} - 1$ different sum-of-products boolean expressions with products of size i. Summing over all possible sizes (1 to n) gives the lower bound. □

Since $\begin{bmatrix} n \\ \lceil (n+1)/2 \rceil \end{bmatrix}$ is of the order 2^{cn}, this lower bound is at least as good as the one in Theorem 5.13. The advantage of above lower bound is that it does not contain an unknown constant c.

Theorem 5.15 (M.Yannakakis, see Theorem 5.3 in [GaB85]): At most 2^{n^2} coteries can be realized by vote assignments, in a universe of n elements.

Proof: See the proof in [GaB85]. □

Theorem 5.16: At most $2^{n^2} \cdot 2^{n^2}$ different bicoteries can be realized by vote assignments, in a universe of n elements.

Proof: The argument is similar to the proof of Theorem 5.3 in [GaB85]. We can view the possible subsets of the universal set U as the nodes of the n-dimensional unit *hypercube*. An n dimensional vector represents a *hyperplane* that cuts the cube in two halves, and each vote assignment with a read (write) accessibility threshold corresponds to one such hyperplane. Hypercube nodes on one side of the vector represent the groups of nodes that have total votes ≥ read (write) accessibility threshold, and nodes on the other side are the remaining groups. Hence we have two hyperplanes, corresponding to the read threshold and the write threshold, respectively. If we move a hyperplane as much as possible without crossing any node, it will end up sitting on n nodes. For each hyperplane, there are $\begin{bmatrix} 2^n \\ n \end{bmatrix} < 2^{n^2}$ choices for the sets of nodes of the hypercube on which to rest the hyperplane. For two hyperplanes, we have less than $2^{n^2} \cdot 2^{n^2} = 2^{2n^2}$ choices. □

5.7. COMPLEXITY OF RECOGNIZING BICOTERIE DOMINATION

Lemma 5.6: If recognizing coterie domination is NP-complete, then recognizing bicoterie domination is also NP-complete.

Proof: This follows from Theorem 5.4 which says that a bicoterie $A = (P, P)$ (where P is a coterie) is nondominated iff P is nondominated. Therefore, we can transform the domination recognition problem of a coterie P into that of bicoterie $A = (P, P)$. $\square$

In [GaB85], it is conjectured that the above recognition problem for coteries is NP-complete. They relate a coterie to a hypergraph where the coterie groups are the hyperedges (a *hyperedge* contains a set of points). It is shown that a coterie is dominated iff the corresponding hypergraph is 2-colorable, i.e., the nodes of the hypergraph can be colored by 2 colors so that every hyperedge has at least two colors. (Intuitively, all elements of one color can form a transversal of the coterie.) Recognizing 2-colorable hypergraphs is known to be NP-complete [Lov73], which is a reason why it is believed that the same is true for recognizing coterie domination. However, since the properties of intersection and irredundancy must be present, coteries correspond to special hypergraphs, hence the result cannot be directly extended to the problem of coterie domination, and it remains open.

5.8. APPLICATION TO GVP

In GVP, we have three types of quorum sets: $RQ(X)$, the global read quorum set, and for each view V, $rq(X,V)$ and $wq(X,V)$, the view read and write quorum sets, respectively. The intersection requirement for these sets are such that each quorum in $RQ(X)$ intersects with each write quorum in $wq(X,V)$ for all V, and each read quorum in $rq(X,V)$ intersects each write quorum in $wq(X,V)$. Since the copies residing at the sites of V form a subset of the set U of all copies, we can satisfy the above requirement if $(RQ(X), wq(X,V))$ forms a bicoterie under U for each V, and $(rq(X,V), wq(X,V))$ forms a bicoterie under V.

Recall that all the protocols described in Chapter 4 make use of $RQ(X)$, $rq(X,V)$ and $wq(X,V)$ having the above properties.

CHAPTER 6

THE TRANSACTION REPLICATION SCHEME (TRS)

6.1. INTRODUCTION

Most replicated database management schemes execute each transaction once, and broadcast the results. We explore the alternative of "replicated transaction processing", meaning that a transaction is executed more than once. The basic idea is to broadcast the transactions instead of their updates; a set of transactions submitted in a short time interval are grouped together and broadcast in one single message and are executed at all the receiving sites. It turns out that replicated transaction processing has some features which may make it attractive in some applications [PiG87, PiG89].

Here, we propose a distributed concurrency control scheme based on replicated transaction processing, called **Transaction Replication Scheme**, or **TRS** for short. TRS possesses the desirable property that read-only transactions can read from local data copies, making it suitable for read-intensive databases.

The other main features of TRS compared with conventional concurrency control schemes are the following:

(1) simplicity of global concurrency control,

(2) reduction in the number of messages, and

(3) faster response time.

Section 6.2 describes the network model, followed by Section 6.3 which describes the data and transaction model. Section 6.4 presents the synchronous broadcasting scheme. Section 6.5 proves the correctness of the scheme and Section 6.6 contains comments on the scheme.

6.2. THE NETWORK MODEL

Our system model consists of a set of processing **sites** (or **nodes**) connected through a communication **network.** All processing needed by distributed applications is performed at sites, while any processing needed for communication (e.g., routing) is performed by the network.

We make the following assumptions about the network and timing.

(1) Each site has its own memory and there is no shared memory.

(2) Each site has a unique ID.

(3) Message size is bounded by *M-size* (bytes).

(4) There is an upper bound[24] of *M-delay*[25] (sec) on the maximum message transmission delay between any two sites, measured by the clock of any site in the system.

(5) Each site has a local clock and the local clocks at any two sites are synchronized to within *C-diff* (sec).[26] Note that a tight value of *C-diff* depends on a tight value of *M-delay* if clock synchronization is achieved via messages (see [Lam78b]).

(6) An upper bound of *X-time* (sec) on the longest execution time of any "batch" of transactions at any site, measured by the clock of any site (see Section 6.4). This bound is merely to guarantee that no transaction batch will execute forever.

[24] We may treat message delay greater than this bound as an omission failure.

[25] In "The Cost of Messages" [Gra88] by Jim Gray, *M-delay* depends on the following :

$$\text{Communication\_Delay} = \text{Transmit\_Delay} + \frac{\text{Message\_Size}(\textit{bytes})}{\text{Bandwidth}(\textit{bytes/second})} + \text{CPU time},$$

where CPU time is computation time in communication protocols for message handling.

[26] In practice, a signal from a very reliable clock can be periodically broadcast, e.g., by radio or dedicated lines, to synchronize the clocks. Alternatively, a reliable clock synchronization protocol can be used [DHS84,LaM84,LuL84]. Failures may invalidate this assumption, but we assume that a failure recovery mechanism will restore the clock difference before normal transaction execution is resumed after a failure.

6.3. THE DATA AND TRANSACTION MODELS

6.3.1. Data Model: Two Types of Fully Replicated Data

We assume that a distributed database system consists of *n* processors at *n* sites, any two of which can communicate via the network with each other when there is no failure.

The ultimate goal is to give each user the illusion that he/she is the only user accessing a centralized database. (We shall see later that the user may be aware that other users can have access to the database and so data can be "private" or "public", but the user should not have to worry about concurrency when other users are accessing the data at the same time.) More formally, let us call the data objects as seen by the user the **logical data objects**. We allow the system to keep more than one physical copy of any logical data object, usually at different sites. Such logical data objects are called **replicated data objects**, and each copy a **physical data object**. Replication should be transparent to the user. We consider only fully replicated data objects, i.e., logical data objects that have a copy at every site.

In most replicated database systems, uscrs might want to have two types of data. (See below for possible reasons.) Each data object of the first type is **private** to a particular site, meaning that only transactions from that site can modify the data object, but transactions from other sites may read it (possibly their local copy of the data object). The second type of data is **public**, meaning that these data objects can be read and modified by transactions from every site.

As an example, consider an airline database system. There may be a site for accounting, a site for flight scheduling, and many sites for seat reservation. The accounting and flight scheduling data would be "private" to the accounting site and the flight scheduling site, respectively. The reservation sites may want to read flight schedules but they need not modify them. Such data are private data. The seat plan of each flight should be public among the reservation sites since every such site can book seats and modify the data. (We assume that customers can pick their seats at reservation sites instead of at the airport.) Such data are public data.

Our objective in considering private data is not to provide privacy, but rather to achieve efficiency. Since private data is updated by only one site, we might expect that simpler concurrency control is sufficient. In our system, we shall actually show that transactions accessing only private data can be executed and committed upon submission without waiting for any communication with other sites.

Previous work on replicated distributed database systems did not distinguish or accommodate the above two types of data. Most popular methods using locking or timestamping assume only the public data, so transactions from any site can read and write any data object. Such systems cannot take advantage of the property of private data that only transactions from one site can modify some data. More arguments that support private data can be found in [ClS80, GaK88, LiS80]. Some researchers have proposed having only private data, e.g., the fragmented database model in [KoG87]. However, systems with only private data have limited applications. For example, they cannot handle concurrent seat reservation from multiple reservation sites.

6.3.2. Transaction Model: Local Transactions and Global Transactions

We assume that each transaction originates from a single site. A user transaction may access a data object X by operations $READ(X,y)$ and $WRITE(X,v)$. A $READ(X,y)$ operation reads the value of X and returns it in variable y. $WRITE(X,v)$ changes the value of X to that of v. The set of logical data objects that the transaction reads (writes) is called its **readset (writeset)**. In our model, each transaction predeclares supersets of its readset and writeset.

We assume that the execution of each transaction is *deterministic*. That is, when a transaction T is executed under a certain initial state of the database and certain interaction with user, there is only one possible outcome: only one possible return value for T and one possible final state of the part of the database affected by T. In other words, transactions do not make random choices which may be different in different runs starting in the same initial database state.

We can now explain private data and public data in terms of transaction operations. Let a logical data object X be replicated at all the sites. The data object X can then be of either one of the following two types:

- **Private data** X -- only transactions submitted at one particular site s can perform $WRITE(X,v)$ on them; transactions submitted at other sites may perform $READ(X,y)$. We say that s is the *owner site* of X.[27]
- **Public data** X-- transactions submitted at any site can perform $WRITE(X,v)$ as well as $READ(X,y)$ on X.

We now identify two main types of transactions:

- **Local transaction**- a transaction initiated at a site s accessing (i.e., reading and/or writing) only logical data objects that are private to site s.
- **Global transaction**- any non-local transaction.

Assumptions and Objective

Let us assume that we have only two types of global transactions:

(1) **type 1** global transactions: read-only global transactions,

(2) **type 2** global transactions: global transactions that may write only the public data.

There can be other types of global transactions (e.g., global transactions that also write data private to the transaction submission site), but they would require more complicated management. We believe that it is realistic to build systems with only the above two types of global transactions. If other types of global transactions are required, we suggest to make all data public. The only disadvantage of this

[27] As we pointed out previously, replication of data should be transparent to the users. However, in the case of private data, we do not assume transparency of the network, which is a different issue. The owner site s of a private data object X understands that there are other sites who might read X but not write X.

approach is that local transactions become global and cannot enjoy the privilege of being executed without waiting (see the description of our protocol in Section 6.4).

It is possible that in some applications, most of the global transactions are of type 2. For example, in an airline reservation system, a transaction may read a flight schedule private to another site, may read/write a public seat plan, but may not modify any private data.

In some systems, local transactions may be more numerous than global transactions. For example, in a banking database system, stationary customers (i.e., customers who do not travel) need only to access private data. The transactions of such customers will be local and they may constitute the majority of transactions. In case such a customer travels, the system can send his/her transactions to the owner site to be executed as local transactions. The processing of local transactions may or may not be replicated; their results are broadcast. Note that for private data, the replication of data could be for the purpose of backup.

We shall devise a protocol based on timestamping that gives priority to local transactions over global transactions, by letting local transactions preempt global transactions.

6.4. SYNCHRONOUS BROADCASTING SCHEME

Define Δ to be *C-diff* + *M-delay*. Let D be the average communication delay (or message transfer time) between two sites measured by the average clock rate over all clocks. The idea behind the scheme we propose in this section is as follows. Essentially, it uses the timestamps as a serialization order. We assume that the timestamps of all transactions in a set (batch) submitted during any period of time are greater than those in the batch submitted in any preceding period of time. Consider a short period of time, δ (sec), and consider the batch B of transactions submitted during this period at all sites. Among the transactions in B, suppose that each global transaction is given a timestamp larger than that of any local transaction. The local transactions are executed at their origin sites and their updates (private data) are broadcast to all other sites. The batch of global transactions submitted at each site, on the other hand, is sent to all sites (including the sending site) and executed at the receiving sites when they arrive there. This means that a local transaction and a global transaction in B are executed at different times, as much as Δ time units apart.

Consider, for example, two sites s_1 and s_2, and suppose that B consists of only two transactions, T_1 and T_2, where T_1 is a local transaction submitted at s_1, and T_2 is a global transaction submitted at s_2. T_1 is executed immediately during the current period of length δ, while T_2 is broadcast by s_2 and will arrive at s_1 later. If T_1 writes a private data object X and T_2 reads X, then T_2 must read X from T_1 (since T_1 is the only transaction in B that writes X). This implies that s_1 must remember the value of X updated by T_1 until T_2 arrives. This value of X may become an old version in case it is overwritten by another local transaction T_3 submitted at a later time at s_1 after T_1, and before T_2 arrives. In general, our scheme requires remembering old values (versions) of private data (updated by local transactions). Moreover, the updated value of X must be sent from s_1 to s_2, so that T_2 can be executed at s_2 as well.

We now discuss the details of our protocols. It will turn out that the local transactions can be scheduled without using timestamps.

6.4.1. Basic Protocol

6.4.1.1. Broadcasting

The basic step of our protocol for each site s is: *accumulate global transactions initiated at s for a time period of* $\delta = \Delta/q$, *where* q *is a design parameter (integer),*[28] *and then broadcast them at the end of the period.*

[28] Theoretically, Δ/q can be made arbitrarily small, but in some cases, smaller Δ/q will require the storage of more versions of some private data, as will be explained later. A small Δ/q could also lead to an excessive number of "null messages".

At the same time, the other sites send in their batches of global transactions from a previous period, which must be collected by s. After every Δ/q (sec), s starts the next period of global transaction accumulation and broadcast. For example, if $\Delta/q = 0.5$, then each site broadcasts at times 0.5, 1.0, 1.5, 2.0, ..., of its local clock. If we attach the unique site ID to the end of the submission time of each transaction, then we get a globally unique timestamp for each transaction. The global transactions are assigned timestamps generated in this way. The timestamps provide a total ordering on all the global transactions.

Before time $t+\Delta$, probably around $t+D$, s should have received from all sites (including itself) the messages which were broadcast at time t according to each broadcasting site's clock. Once s has received a message from every site, it totally orders all the received global transactions according to their timestamps, and executes them using this order as the serialization order (i.e., makes sure that the execution is equivalent to a 1C serial schedule (see definition in Chapter 2) which orders the transactions according to the timestamps).

6.4.1.2. Transaction Scheduling and Execution

Local Transactions

At each site s, during the first q periods of Δ/q (sec) each (e.g., after a system restart and before s receives a message from every site), only local transactions are executed. s may have received some global transactions broadcast by some other sites by the end of the qth period, but there is no guarantee that all such broadcasts have been received. In general, *local transactions are executed immediately upon submission according to some local concurrency control scheme. At the end of each period, s broadcasts the final values of private data updated in that period together with the batch of global transactions accumulated during that period at s.*

The value of a private data object X that is sent by s in each broadcast is stored as a **version** of X at s. At the end of the q th period, up to q versions of s 's private data may be stored at s.

Type 2 Global Transactions

After the first broadcast from every other site has arrived (probably at around time D and by time Δ/q at the latest), site s examines the messages which have been received from the other sites, namely, the new versions of the senders' private data as well as a new batch of global transactions totally ordered by their timestamps. Site s then executes the type 2 global transactions in the batch. The execution can be concurrent as long as it is equivalent to a 1C serial schedule that follows the timestamp order. To ensure 1C serializability, we enforce the following rule. (We shall shortly see why this rule needs to be followed.)

When a global transaction T, which is broadcast by a site s_2 at time t and executed at site s_1 later, reads a private data object X, T should read the version of X that was broadcast at time t by some site s_3, where $s_3 = s_2$ or $s_3 = s_1$ is possible.

This version is discarded when the execution of the batch of global transactions is completed. This implies that, in general, the oldest undiscarded versions of private data are consistent with the newest versions of global data. Note that if $s_3 \neq s_2$ and $s_3 \neq s_1$, then the version of X to be read by T arrived at s_1 in the same *batch* as T. If $s_3 = s_2$, on the other hand, then the version of X to be read by T arrived at s_1 in the same *message* as T. Otherwise, i.e., if $s_3 = s_1$, the version of X (a private data object *local to* s_1) to be read by T is the oldest version of X kept at s_1.

The situation is illustrated in Figure 6.1. In the figures in this chapter, a horizontal axis represents time measured by site s's clock, and the transmission of messages is indicated by a slanted arrow. The tail of an arrow rests on the time axis at the message sending time, and the head of the same arrow, when projected on the time axis, corresponds to the time (s 's clock) at which every message broadcast from every other site s_i at time t (according to s_i's clock) has arrived. In these figures, it is assumed that the clocks keep the same time and each broadcasting takes exactly Δ time units. Figure 6.1 shows that if each period lasts Δ/q time units, and if the execution of a batch of transactions can be completed in Δ/q time units, then in the worst case, $q+2$ versions of some *local* private data may be required. (In ideal timing, no extra versions of *remote* private data need to be stored; global transactions and the remote private data they may read arrive in the same batch.) The worst case is where message delay is Δ, and this requires the largest number of versions. If messages travel faster, some versions of local private data can be discarded earlier, and fewer versions may be needed. Local transactions can be executed any time; they may interrupt broadcasting or the execution of type 1 and 2 global transactions.

In Figure 6.1, we assume a site s owns a private data object X with consecutive versions, $x_1, x_2, x_3, x_4, x_5, x_6$. When site s broadcasts a new version x_i (as shown by a slanting arrow), it also broadcasts a new batch of transactions B_i submitted at s. In the figure, we have assumed that the clocks tic at the same speed and each broadcasting takes exactly Δ time units. A system with average transmission time D can complete broadcasting in around D ($< \Delta$) time units. In such a case, execution can be started once all the broadcast messages of a period are received at a site. For example, in Figure 6.1, a site can start executing a batch of global transactions B_2 earlier than time t_2, as long as it has received the message from every site s_i broadcast at t_1 (s_i's time).

Type 1 Global Transactions

So far, we have described our scheme as if all global transactions were broadcast. However, type 1 global transactions are not actually broadcast to other sites, since these transactions are read-only and do not affect the database. If such a transaction T needs to read local private data, then it reads their oldest undiscarded versions. Thus, T needs not wait before it executes and commit. As commented earlier, the oldest undiscarded versions of local private data are consistent with the newest versions of public and remote private data. Therefore, T reads a consistent set of private and public data. Note, however, that in effect we have changed T's timestamp so that T is considered as old as the local transactions that created the oldest undiscarded versions of local private data.

Important Subtype of Type 2 Global Transactions

An important subtype of type 2 global transactions are those which read/write only public data. It is important because, if all global transactions are of this subtype, there is no need to store multiple versions of private data. Thus, we may set the time period Δ/q to be very small without the penalty of having to store many versions of private data. The advantage of introducing private data is that local transactions can be executed without waiting, giving them a faster response than global transactions, which must wait for some multiples of communication delay time.

6.4.2. Timestamp Ordering Algorithm without Transaction Abortion

We propose to adopt a scheduling technique based on timestamp ordering. We need a protocol which does not rely on transaction abortion. For, if a transaction is aborted at one site, it has to be aborted at every other site, which means that we cannot commit a transaction until we are sure that no other site will abort it later on. We would thus need some commit protocol like *two-phase commit* or *three-phase commit protocol* [Ske82b], and end up with poor performance.

CONSERVATIVE TIMESTAMPING TECHNIQUE

As is well-known, concurrency control using **conservative timestamp ordering** [BeG81] does not require transaction abortion. We shall see that the synchronous broadcasting of transactions makes this approach easier because little waiting (at most Δ/q time units) is necessary for a site to make sure that no transactions with older timestamps from other sites will arrive. The crucial point is that each transaction T predeclares its readset and writeset, denoted by **readset**[T] and **writeset**[T], respectively.

Our approach is to preprocess all the transactions in each batch B to detect read/write and write/write **conflicts**. Let T_1, T_2, ..., T_n be the transactions in B in the timestamp order. To simplify explanation we shall add an imaginary transaction T_0 at the beginning. T_0 writes every public data object.

Our algorithm constructs the set $PRECEDE[T_i, X]$, which contains all transactions that should be executed before T_i. For each data object X, it examines each transaction T_i that reads or writes X. If T_i writes X, then the algorithm looks for the closest preceding transaction (in terms of timestamps) T_j that writes X and puts it in $PRECEDE[T_i, X]$. All the transactions with timestamp between those of T_j and T_i that read X are also placed in $PRECEDE[T_i, X]$. For each transaction T_j in $PRECEDE[T_i, X]$, T_i is inserted into the set $INFORM[T_j, X]$, so that T_j can inform T_i about the completion of T_j when it *finishes execution* (see the next paragraph). If X is only read by T_i, then the algorithm looks for the closest preceding transaction (in terms of timestamps) T_j that writes X. T_j is then placed in $PRECEDE[T_i, X]$, and T_i is inserted into the set $INFORM[T_j, X]$. Note that the transaction manager (TM) carries out these operations locally at one site. When T_j "finishes execution", TM erases it from $PRECEDE[T_i, X]$. T_i cannot access data object X unless $PRECEDE[T_i, X] = \emptyset$. A formal description of our algorithm is given in Figure 6.2, which is started when a set of global transactions has been received from every site, forming a new batch $\{T_1, T_2, ..., T_k\}$.

An important point in the above paragraph is the meaning of the phrase "finishes execution", which means "commits". We keep only one version of each public data object; a transaction executes by accessing a copy in its temporary work area, and when the execution is completed the copy in the work area is written into stable storage, from where later transactions will read the value of the data object. We assume that writing into stable storage is an atomic action, and is considered as part of the committing action. The above scheme enforces timestamp order, therefore, no abortion is necessary.

```
Assume batch $\{T_1, T_2, ..., T_k\}$ is ordered by timestamps.
Preprocessing before execution:
  for all transactions $T_i$ and data objects $X$ in readset[$T_i$] or writeset[$T_i$]
    $PRECEDE[T_i,X] := \varnothing$
    $INFORM[T_i,X] := \varnothing$
  for each data object $X$ and each transaction $T_i$,
  if $X \in$ writeset[$T_i$]
  then
     $j := i - 1$
     COUNT := 0
     while $X \notin$ writeset[$T_j$]
     do
       if $X \in$ readset[$T_j$]
       then
          $PRECEDE[T_i,X] := PRECEDE[T_i,X] \cup \{T_j\}$
          $INFORM[T_j,X] := INFORM[T_j,X] \cup \{T_i\}$
          COUNT := COUNT + 1
       $j := j - 1$
     if ($j \neq 0$) and (COUNT = 0)   * no read between two writes
     then
       $PRECEDE[T_i,X] := PRECEDE[T_i,X] \cup \{T_j\}$
       $INFORM[T_j,X] := INFORM[T_j,X] \cup \{T_i\}$
  else
  if $X \in$ readset[$T_i$]
  then
     $j := i - 1$
     while $X \notin$ writeset[$T_j$] do $j := j - 1$
     if $j \neq 0$
     then
       $PRECEDE[T_i,X] := PRECEDE[T_i,X] \cup \{T_j\}$
       $INFORM[T_j,X] := INFORM[T_j,X] \cup \{T_i\}$
Necessary condition before a transaction $T_i$ accesses $X$:
  $PRECEDE[T_i,X] = \varnothing$
After execution of a transaction $T_i$:
  For each $T_j \in INFORM[T_i,X]$
     $PRECEDE[T_j,X] := PRECEDE[T_j,X] - \{T_i\}$
```

Figure 6.2 Preprocessing.

Note that when we add a transaction T_j to *PRECEDE* $[T_i, X]$, it is similar to *locking* the data object X so that T_i cannot access X until T_j "finishes execution". We say that X is **virtually-locked** by T_j in respect to T_i when X cannot be accessed by T_i in this sense. If T_j reads (writes) X, then we say that T_j has a **virtual read (write) lock** on X against T_i. Releasing a virtual lock in the above example corresponds to the removal of T_j from *PRECEDE* $[T_i, X]$.

6.4.3. Large Transactions and Long Transactions

6.4.3.1. Large Transactions

We assume an upper bound of *M-size* (bytes) on message size. This will be a problem for large transactions, where the size of the transaction > *M-size.* One solution is to break up a large transaction T into consecutive sub-transactions $\{ T_1, T_2, ..., T_k \}$, so that each T_i is short enough to fit into one message. We assume that a large transaction T can predeclare its entire readset and writeset. Therefore, with the first subtransaction T_1, it declares its readset and writeset. The subtransactions are broadcast in consecutive message broadcasts. T is given a timestamp which is picked at the beginning of the period just preceding the broadcast of T_k. $T_1, T_2, ..., T_k$ are not executed until all of them are received. Therefore, we are treating T as if it were submitted in its entirety at about the same time as those transactions which are broadcast in the same batch as T_k.

6.4.3.2. Handling Overdue Global Transactions

We have seen in Section 6.4.1.2 that $q+2$ versions of some private data may be needed if the execution of a batch of global transactions can be completed within Δ/q seconds. Here we consider the case where the execution of batch global transactions requires more than Δ/q seconds.

If a site s cannot finish the current batch of global transactions within a period of Δ/q seconds, it may still broadcast the next batch of global transactions collected during the current period when the broadcasting time comes. However, site s will have to try to finish the uncompleted transactions belonging to the current batch in the next period. In order to do this, s might have to keep one more version for some private data. We can see the reason from Figure 6.3, where an old transaction a is delayed from period $[t_1, t_2]$ to $[t_2, t_3]$. If a reads version x_1 of private data X, then we must keep it during the period $[t_2, t_3]$ until a commits at site s. From Figure 6.4, it is seen that if a site cannot finish a batch for k periods, it may have to keep k more (in addition to the $q+2$ versions which are normally required) versions of some data.

Moreover, since transactions are broadcast as usual, site s will be storing more and more global transactions to be executed. We therefore look for ways to reduce overdue transactions.

There are two causes for overdue global transactions:

(A) Too Many Global Transactions

One possible reason why a site s takes so long to finish transactions is that there are too many global transactions arriving within a short period. One solution to this problem would be to *limit the size of the batch* in each period. For example, we may set an upper bound MB (maximum broadcast size) on the number of global transactions that a site can broadcast to others. If a site s has received N ($>MB$) global transactions from users within a period, then it must only broadcast the first MB transactions and store the rest for the next broadcast. If MB is set properly, then the total batch accumulated at each round is likely to be finished in one round at most sites.

With this scheme, only site s stores delayed transactions submitted at site s. Without the limiting scheme, transactions would be stored at all sites, occupying much more storage space. If too many transactions arrive at s and threaten to overload its storage, s can give feedback to the users to request a

[29] Another reason why there are too many global transactions in a time period is that a site may have to resynchronize its clock by advancing it, in which case, the current period is shrunk, and not long enough to execute all transactions for that period.

slow-down on their side.

(B) Too Many Local Transactions

There is another possible reason why a site s could not finish global transactions in time. There may be too many local transactions arriving at s within one period, or some local transaction taking up a lot of computation resources. Since local transactions have priority over type 2 global transactions, they would be able to dominate the use of computation resources. Therefore we can simply *raise the priority level*[30] *of long overdue global transactions* at site s, *in order that they get to use the computation resources.*

6.5. CORRECTNESS OF THE PROPOSED SCHEME

Before presenting a formal proof of correctness of our scheme, we shall illustrate the idea behind the proof by an example.

Example: Consider the execution shown in Figure 6.5, where each period is $\Delta/3$ time units long. In the top part of the figure, the transactions are plotted on the horizontal time axis in the order they were executed. All local transactions in this example (indicated by x in the figure) access only one local private data object X, and $\{ x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8 \}$ are consecutive versions of X. The local transactions a_1, a_2, for example, were generated between time t_1 and t_2, and they modified the value of X from x_4 to x_5. The global transactions a_3, a_4 were executed in the same time period as a_1 and a_2, but they had been broadcast at time $t_1 - \Delta$. The global transactions e_2, e_3 were generated between t_1 and t_2 and broadcast at t_2.

[30] So far we have assumed that local transactions have priority over global transactions (see Section 6.4.1.2).

What we want to show is that the replicated log (or rd log, see Sec. 2.3) representing this execution is 1C serializable by finding a serialization order corresponding to an equivalent 1C serial log. Since global transactions $a_3, a_4, b_2, b_3, c_2, c_3, d_3, d_4$ use values, x_1, x_2, x_3, x_4, they must be serialized before a_1 and a_2. Hence the local transactions between t_1 and t_2 are serialized after all global transactions broadcast at time t_1 (i.e., d_3, d_4). (See the serialization order shown in the bottom half of Figure 6.5.) Global transactions e_2 and e_3 access the value x_5, and thus should be serialized after a_1 and a_2, but before local transactions b_1 and b_4 which modify x_5 to x_6. Hence, the local transactions executed between t_2 and t_3 (i.e., b_1 and b_4) are serialized after all global transactions broadcast at time t_2 (i.e., e_2 and e_3) and before all global transactions broadcast after time t_3 (i.e., f_1, etc). □

The crucial point to be observed in the above example is the fact that the serialization order among the (type 2) global transactions is the same as their timestamp order. As for the local transactions (e.g., a_1 and a_2), they are serialized just before the (type 2) global transactions (e.g., e_2 and e_3) which were generated in the same time period. Note that, in the above example, we only considered serialization order among the local transactions at one site and the (type 2) global transactions. Of course, we have also to consider the local transactions at other sites and type 1 global transactions. As for the latter, we have already shown that they read a consistent set of data values, which means they can be serialized at appropriate places in the serialization order. As for the former, local transactions generated at different sites do not conflict by definition. Therefore, it is easy to integrate them in the serialization order defined by the timestamps of the type 2 global transactions.

We shall call the global transactions actually executed at each site accessing physical data copies the **transaction replicas** (or simply replicas) of the **logical global transaction** which is intended by the user to operate in the same way but on the corresponding logical data objects.

We say that a TRS schedule (schedule of the TRS system) α is equivalent to a 1C-serial schedule β if β contains all the logical global transactions and local transactions in α and the following conditions hold.

(1) A replica T_r of logical global transaction T reads from T'_r (a replica of the logical transaction T') in α iff T reads from T' in β, and T_r reads from local transaction T'' in α iff T reads from T'' in β.

(2) The read-from relation among local transactions are the same in α and β.

If a TRS schedule α is equivalent to a 1C-serial schedule, then α is said to be 1C-serializable.

Theorem 6.1: *If no failure occurs, then each committed schedule (consisting of transactions which have committed) generated by the above scheme is equivalent to a 1C-serial schedule in which all the local transactions committed in the period* $[t_i, t_i+\Delta/q]$ *are scheduled after all the global transactions broadcast at time* $< t_i$ *and before all global transactions broadcast at time* $\geq t_i+\Delta/q$*, where all the time values are measured by the clocks of the corresponding transaction submission sites.*

Proof: In the conservative timestamp ordering scheme in Section 6.4.2, when a global transaction is committed at one site, it will also be committed at every other site, and all committed global transactions are committed in their timestamp order.

Next we show that, in general, for any schedule α generated by the TRS system, there is an equivalent 1C-serial schedule in which the local transactions (LT) that commit in any scheduling period of the form $[t_i, t_i + \Delta/q]$ in α appear after all global transactions (B_1) in α broadcast at time t_i and before all global transactions (B_2) in α broadcast at or after $t_i + \Delta/q$. Suppose local transactions in LT update the value of private data object X from x_1 to x_2. Then x_2 is broadcast with the global transactions in B_2 and when the transactions in B_2 are executed, they read the value of x_2, that is, transactions in B_2 may read from transactions in LT. Hence the resulting execution of B_2 and LT will be equivalent to a 1C-serial schedule in which the transactions in LT appear before those in B_2. Also, before x_2 and B_2 arrive, B_1 will be executed and if transactions in B_1 read X, they will read the value x_1. Therefore, the resulting execution of B_1 and LT is equivalent to a 1C-serial schedule where the transactions in B_1 appear before those in LT. □

6.6. COMMENTS ON THE SCHEME

At the beginning of this chapter we listed some nice features of TRS. Here we can justify those claims with the details of the scheme.

(1) Simplicity is a key requirement for practical systems both for implementation and for correctness. For TRS, there is no complicated global concurrency control; the broadcasting of transactions and synchronization of execution (enabled at each site if it receives broadcast from all other sites) are the means to ensure correctness. It is simpler than 2-phase locking because there is no explicit remote locking (i.e., explicit request of locks from a remote site), and there is no need for global deadlock detection.

(2) TRS can provide savings on the number of messages for systems with heavy workload. At each site, at regular intervals, a set of transactions are broadcast in one single message to every other site. This compares favourably with the number of messages required for the usual locking schemes, where each reading or writing operation of a transaction requires its own locking, acknowledgement and unlocking messages. The number of messages in such a system may be reduced by concatenation of messages at lower levels of the network. However, concatenation of messages at lower level would require more total overhead then that at higher level.
When the size of a transaction is smaller than the size of data update, and computation is not heavy, we also have an advantage in the size of transmitted data. A more detailed comparison by analysis or simulation is left for future research.

(3) When there are no conflicts, a transaction under conventional locking or strict timestamp ordering scheme will need 2 message transmission delays before it can commit. A transaction in the TRS scheme waits for a short time period before broadcasting and then requires one message transmission delay before it can commit. This waiting time before broadcasting can be made small, so that the TRS scheme outperforms the above schemes in terms of response time.

(4) There is one major problem with distributed (non-centralized) database systems with the currently known concurrency control schemes. It is the availability of data that is under great contention of updates by transactions, sometimes referred to as *hot spot* data[31]. Most currently known schemes use either 2-phase locking or timestamping. We can see that conventional timestamping schemes which avoid cascading aborts also "virtually lock" data because, if an update has not committed, other transactions should not see the new value. (See an overview of locking and timestamping mechanisms in Chapter 3.) With distributed systems, (explicit or virtual) locking lasts at least twice the message transfer delay time. One message delay is required to acknowledge the lock-request, another message delay is to confirm the commit of the transaction (possibly piggybacked with the update data). With "hot spot" data, we cannot afford to have each updating transaction hold the data for twice the message transfer delay time. This may be a reason why the existing systems like airline reservation systems still adopt the centralized approach, although the centralized approach has drawbacks like dependence of reliability on the central site.

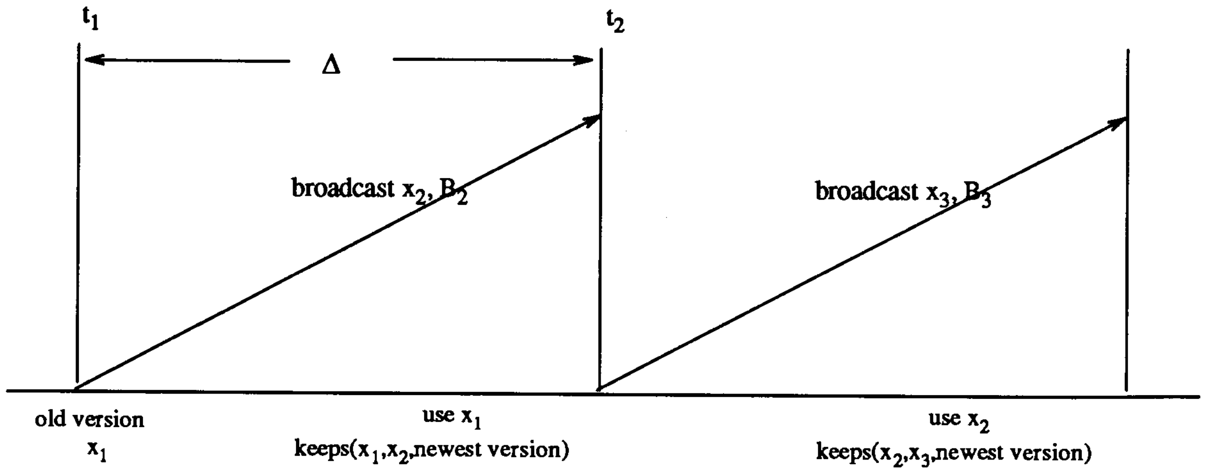
In TRS, conflicts between transactions are resolved with a timestamping method that will not cause abortion. Any locking of data in the general sense (i.e., in the sense that some transaction cannot access some data object until another transaction commits) is local rather then global. The key reason why we can achieve this is because each site knows about the complete set of transactions (i.e., all possible conflicts on accessing the data) in each time period, and we assume that each transaction declares its readset and writeset.

Unfortunately, TRS also has disadvantages:

[31] If Writes are used to add and subtract from a data object, then we can replace Writes by increment and decrement operations, which will be commutable, and hence set weaker locks than Writes (see [GaK85, PeR88, Reu82]).

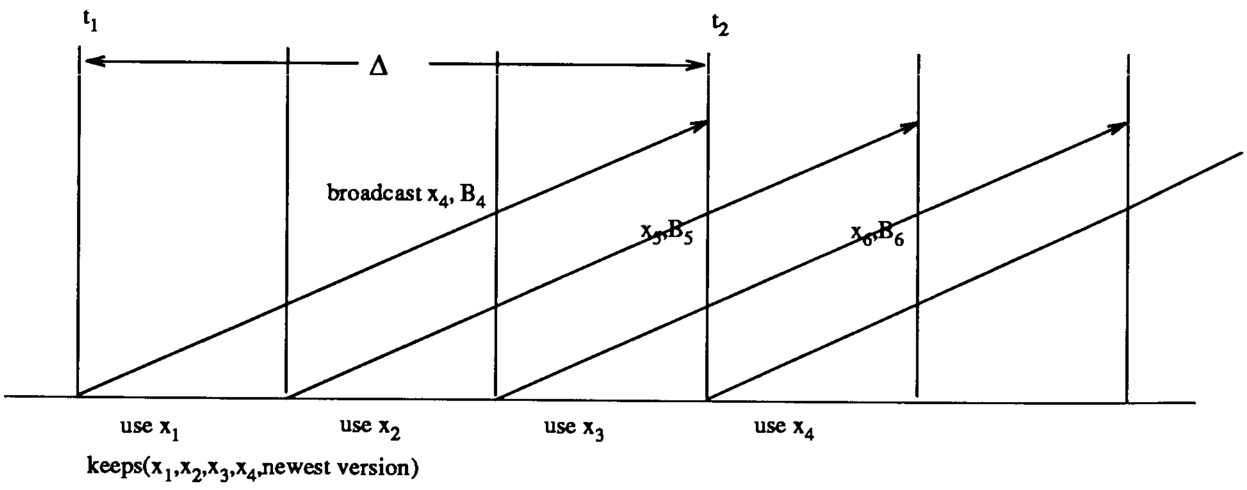
(1) When transactions involve intense computation, the savings on messages may not cover the expense in replicated computation. However, we have ways to get around this problem. In essence, for such a transaction, we broadcast the operations and data involved, but only one site will do the execution. Similarly, if the size of a transaction T is much greater than the update size, then we broadcast only the *writeset* (data to be updated), but only the origin site s will do the execution. Other sites then know the data references and can prevent conflicting operations on those data from other transactions. In particular, the writeset of T will be "locked" (see Section 6.4.2.1) at every site (including s, even when s has finished executing T) throughout the period during which other sites do not know the outcome of T. The results are then broadcast to update every site along with the next batch of transactions $B(s)$ (portion of batch B submitted at s). The lock on the writeset of T at a site is released before the execution of B. However, with this approach, there is a tradeoff between computation cost and reliability whose choice can be decided by the user.

(2) In the scheme, each site collects and broadcasts transactions regularly. If no transaction has been collected during a period, a null message has to be broadcast anyway. The synchronous broadcasting scheme of TRS may cause excessive null messages (messages that have nothing to do with user transactions to be sent) if system workload is low. However, we argue that to enhance performance in an asynchronous broadcasting scheme, frequent null messages are also inevitable.



each round = Δ time units

requires 3 versions

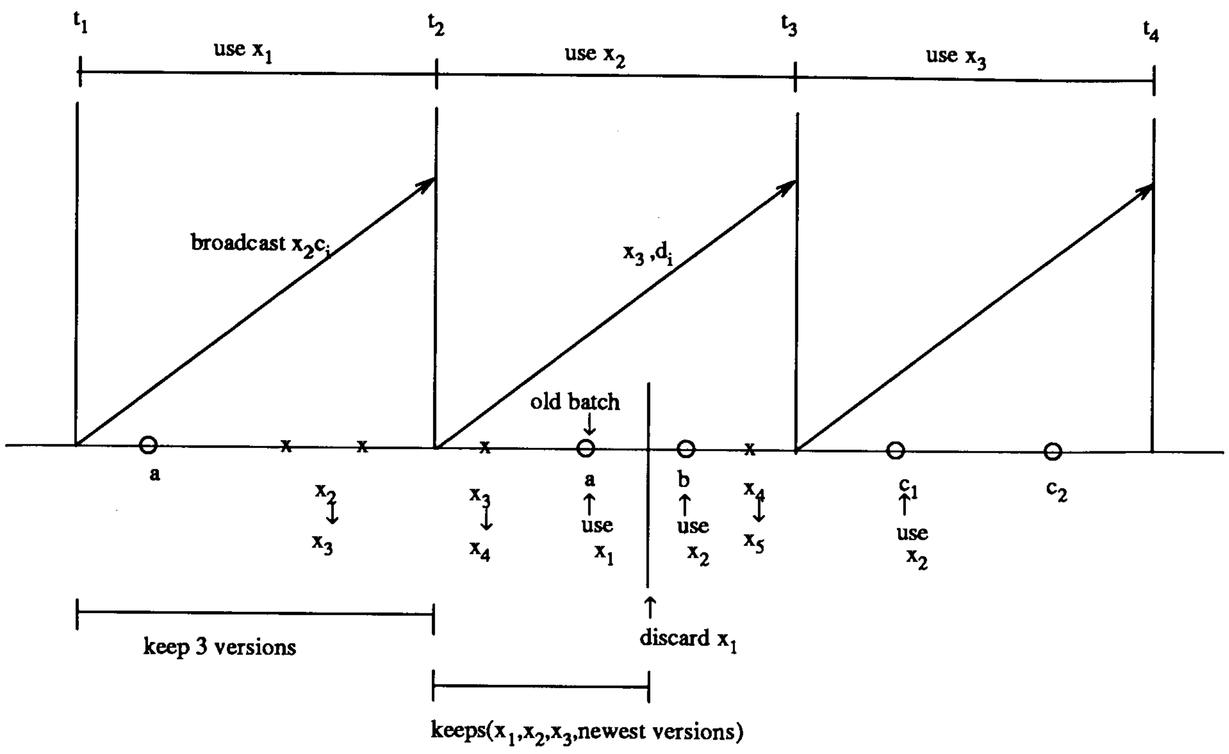


each round = 1/3 Δ time units

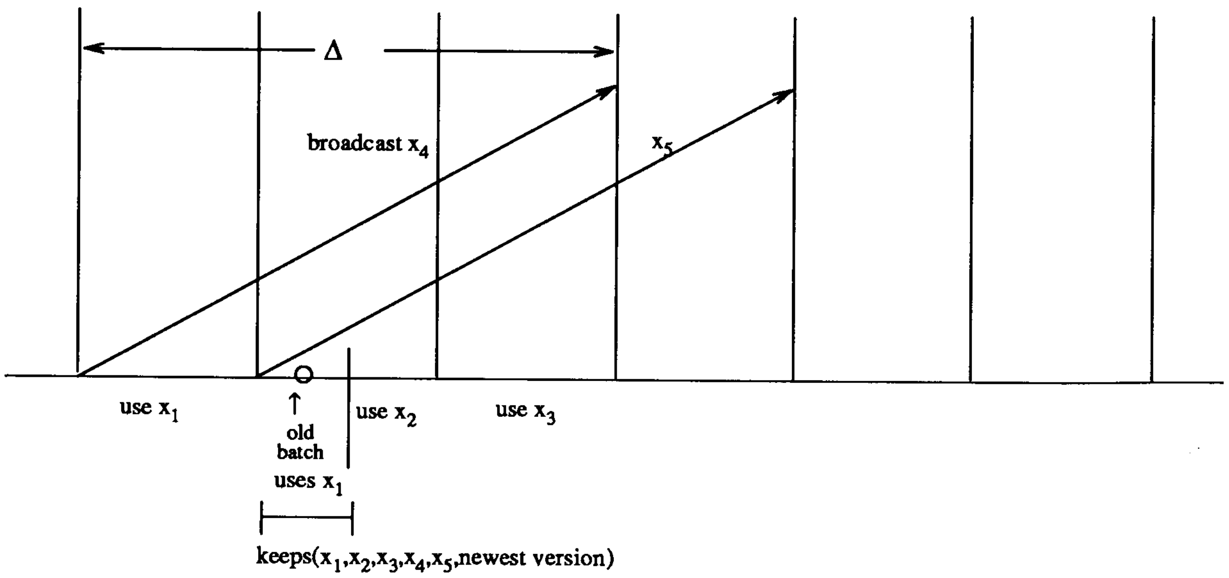
requires 5 versions

In general, if a round = 1/k Δ time units then requires 2+k versions

Figure 6.1 Transaction broadcast and multiple versions of private data

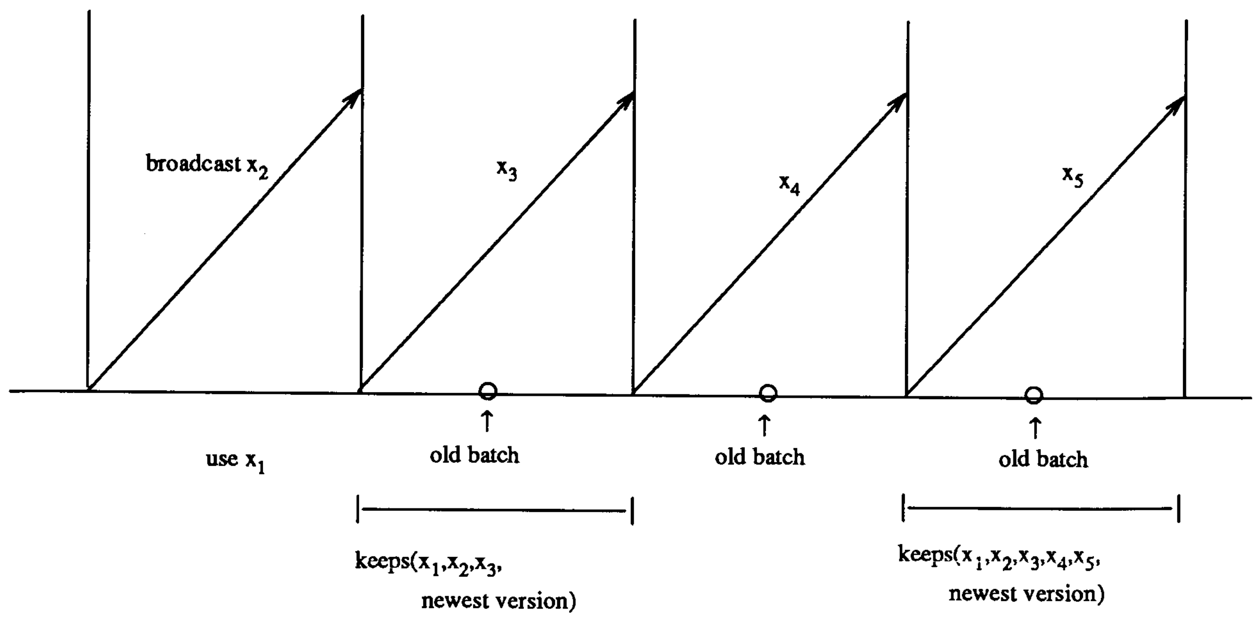


For time period = Δ, a transaction overdue requries 4 versions



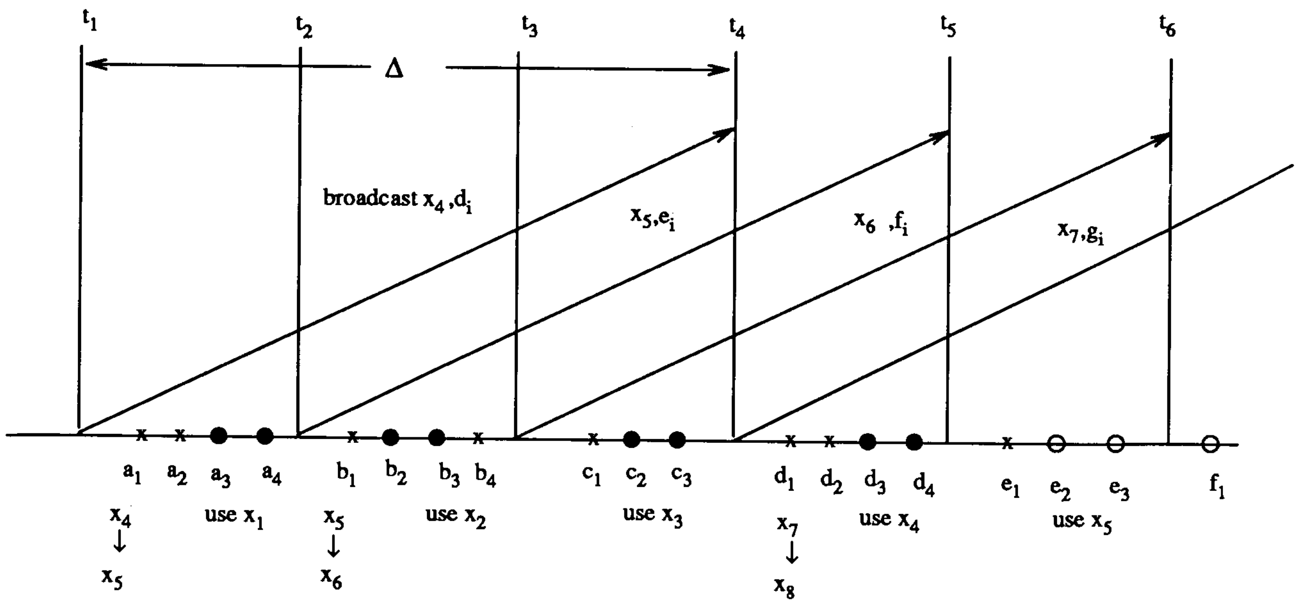
In general, one more version may be needed for overdue transaction

Figure 6.3 Overdue Transactions

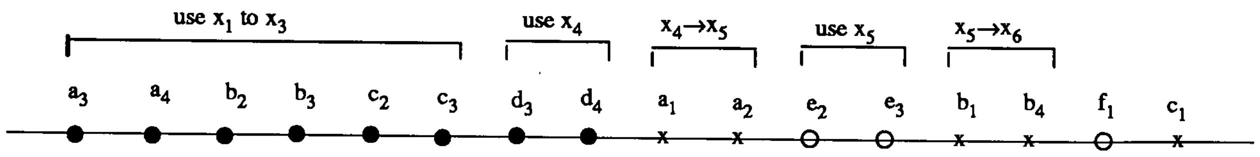

broadcast x_2
x_3
x_4
x_5
use x_1
old batch
old batch
old batch
keeps(x_1,x_2,x_3, newest version)
keeps(x_1,x_2,x_3,x_4,x_5, newest version)

When a transaction is overdue k periods, it may requires k more versions of some private data.

Figure 6.4 Long Overdue Transactions



An Equivalent Serial Schedule :



use x_1 to x_3
use x_4
$x_4 \rightarrow x_5$
use x_5
$x_5 \rightarrow x_6$
a_3 a_4 b_2 b_3 c_2 c_3 d_3 d_4 a_1 a_2 e_2 e_3 b_1 b_4 f_1 c_1

● } type 2 global transaction
○

x local transaction

Figure 6.5 Type 2 Global Transactions

CHAPTER 7

CONCURRENT NESTED TRANSACTIONS ACCESSING B-TREES

7.1. INTRODUCTION

Work has been done to enhance concurrency by using the semantics of particular data structures such as B-tree, hashed file, etc. [KuL80], [LeY81], [Sag86], [El187], [GoS85], [ShG88]. However, most such **semantically-based** concurrency control schemes make the simplifying assumption that a transaction consists of a single **decisive operation** [ShG88], such as read, insert or delete. (Search, split and merge operations are non-decisive.) Here we deal with a more realistic and more complicated model, where a database is a collection of search structures (B-trees), and each transaction may be nested and may perform more than one decisive operation.

Among the known concurrency control schemes for B-trees, the one due to Sagiv [Sag86] probably can achieve the highest degree of concurrency since each read/write operation has to lock only one node in the B-tree simultaneously [JoS90]. He makes use of the $\mathbf{B^{link}}$ **tree** proposed in [LeY81]. A B^{link} tree is obtained from a B-tree by adding to each vertex a pair (k, p), where k is highest key stored in the subtree rooted at the vertex, and p is a pointer to the next vertex at the same level, called a *link* (See Figure 7.1). With this added structure, only a single vertex needs to be locked when an overflowed vertex is being split into two vertices. Suppose that a vertex v with parent u is split into two vertices v' and v'', where v and v' have the same "address" as seen by u, and that a concurrent search for a key in v has visited u before the splitting operation and follows the pointer to v' after splitting. Even if the key to be accessed now resides in v'', the link from v' to v'' can be followed to read the desired key. Therefore, no backtracking is necessary for read accesses. In [Sag86], there is no need for **2-phase locking** to ensure serializability, since his user transactions have at most one read/write operation on data in the B-tree, which is called a decisive operation.

We consider **nested transactions** [LyM86] accessing B-trees, combining the B^{link} tree algorithm with *strict* 2-phase locking [EGL76]. When transactions are nested, they can be represented by a transaction tree. As in [Mos81], a transaction can acquire a lock only if all the holders of the locks conflicting with it are its ancestors in the transaction tree. (Initially, the root transaction holds all locks.) Therefore, when a transaction A requires a lock on a data object X and a conflicting lock on X is held by another transaction B, A has to wait until B and all ancestors of B that are proper descendents of the least common ancestor of A and B have committed. In applying 2-phase locking, in order to take advantage of the B^{link} tree algorithm, we want to lock the individual vertices and keys of a B-tree.

Serializability is a widely accepted criterion for correctness for conventional schedules. But when semantics are considered, some non-serializable schedules can be considered "correct". Another complication is that two B-trees may contain the same set of data, although there is no one-to-one mapping between the links and vertices of the two B-trees. We consider two states of a B-tree to be "equivalent" if they contain the same set of data. We propose a correctness criterion, called "strongly-serially correct", which basically says that a schedule α is correct if there is a serial schedule β such that no "important" (user visible) transaction T can tell the difference between α and β from the data read by T. For such α and β, we show that each data object storing the value of a key is updated by α and β in exactly the same sequence. (Cf. [ShG88].) In other words, we ignore the non-decisive operations such as searching and vertex-splitting as well as the structures of B-trees, as long as they are equivalent.

We use the **I/O automaton model** [LyM86] to formally describe our system and in proving its correctness. Both transactions and data objects are represented by I/O automata. I/O automata can naturally model object-oriented databases: each automaton can be seen as an object in which both data (constituting a part of an automaton's state) and allowable operations are defined. I/O automata interact by their output and input operations.

How to apply the I/O automaton model to describe and prove the correctness of semantically-based concurrency control schemes was given as an open problem in [FLM88, LyM86]. Our work is an attempt to address this open problem. It can also be considered as an extension to [HaH88], which considers concurrency control schemes for nested transactions accessing object bases. Our work also extends [FLM87]. (Due to the complexity of the system, parts of our discussion are going to be fairly informal.)

7.2. THE MODEL

Nancy Lynch, et al., at MIT have constructed the "theory of nested transactions", in which they formalized the description of nested transaction systems in terms of I/O automata. We shall adopt their model. In this section we give a brief and informal review of I/O automata and nested transaction systems based on [FLM87], [FLM88], [GoL87], and [LyM86].

7.2.1. Nested Transaction

While a (conventional) transaction is just a partially ordered set of primitive operations (e.g., reads and writes on independent[32] data objects) that are executed as a unit, a nested transaction has a hierarchical structure: each nested transaction consists of either primitive operations or **subtransactions** which are themselves nested transactions. In a nested transaction system, each transaction instance T may create subtransaction instances, which become the child transactions of T. The transaction instances and their parent-child relationship form a **transaction tree**. A transaction can commit only if each of its descendents has either committed or aborted. For the many advantages of nested transactions, the reader is referred to [LyM86] and [Mos85].

[32]I.e., not organized into a structure such as a B-tree.

7.2.2. I/O Automaton Model

An **I/O automaton** A is a 5-tuple (*states*(A), *start*(A), *out*(A), *in*(A), and *steps*(A)). *states*(A) is the set of states of A, of which a subset *start*(A) is the set of start states. *out*(A) is the set of A's output operations, and *in*(A) is the set of A's input operations. *steps*(A) is the **transition relation** of A, which consists of triples of the form $<s', \pi, s>$, where $s', s \in states(A)$, and $\pi \in in(A) \cup out(A)$. This triple means that in state s', automaton A can indivisibly perform operation π and change to state s. An element of the transition relation is called a **step** of A. A state s' may satisfy the **preconditions** of more than one operation, i.e., $<s', \pi, s>$ and $<s', \pi'', s''>$ may both exist in *steps*(A), in which case the system **non-deterministically** executes one of the enabled operations.

An **execution** of automaton A is an alternating sequence $s_0, \pi_1, s_1, \pi_2, \ldots \pi_n, s_n$ of states and operations of A, such that $s_0 \in start(A)$ and $<s_i, \pi_{i+1}, s_{i+1}>$ is a step of A for each i $(0 \leq i \leq n-1)$. A **schedule** of A is the subsequence of an execution of A consisting only of operations.

We describe a **system** in terms of interacting components, each of which is an I/O automaton. A set of I/O automata may be composed to create a system S, if the sets of output operations of the automata in the set are pairwise disjoint. (Thus, every output operation in S will be triggered by exactly one component.) A state of the composed I/O automaton is a tuple of states, one for each component, and the start states are tuples consisting of start states of the components. The operations of the composed automaton are those of component automata. Each output operation of a component automaton is connected with the identically named input operation of another component automaton. In the resulting system, an output operation is generated autonomously by one component and is thought of as being instantaneously transmitted to the components having the same operation as an input; the input step is simultaneous with the output step.

Let α be a schedule of a system with a component automaton A. The **projection** of α on A, denoted $\alpha | A$, is the subsequence of α consisting of all the operations of A. Clearly, $\alpha | A$ is a schedule of

A.

7.2.3. Nested Transaction Systems

To model nested transaction systems, a **system type**, which is a four-tuple $(T, parent, O, R)$ is used. T is the set of transaction names organized into a tree by the mapping *parent*: $T \to T$, with T_0 as the root. In referring to this tree, we use the traditional terminology, such as child, leaf, least common ancestor (lca), ancestor and descendent. (Note that a transaction is its own ancestor and descendent.) The leaves of this tree are called **access transactions** or simply **accesses**. The set O denotes the set of (data) **objects**; it partitions the set of accesses, where each partition block contains accesses to one particular object. R is the set of **return values** of the transactions.

A system of a given system type is the composition of a set of I/O automata. This set contains a transaction automaton for each internal (i.e., non-access) node of the transaction tree, an object automaton for each element of O and a **scheduler** automaton. A transaction automaton has just one start state, while an object automaton may have many start states, depending on the initial value of the data object. (We sometimes use an automaton to represent a set of data objects, instead of one automaton for each data object.) **Serial systems** ([LyM86], [FLM87]) are those in which the scheduler makes sure that only **serial executions** are allowed. In a serial execution, a transaction is created only when all its elder siblings have been aborted or have requested to commit. Each execution hence corresponds to depth-first traversal of the transaction tree.

We investigate two systems that manage search structures. They are the **flat file locking (FFL) system** and the **B-tree locking (BTL) system**. Of main interest to us is the BTL system which controls concurrent execution of nested transactions in a database system with B-trees. The FFL system is introduced as a means to prove the correctness of the BTL system. The FFL system is the same as the R/W Locking system defined in [FLM87], except for the naming of objects (see the next section). We shall define a stronger correctness condition than that used in [FLM87] and prove the stronger correctness of the FFL system, and then establish the correctness of the BTL system by showing that it "simulates" the FFL system.

7.3. SERIAL SYSTEM

The serial system is identical to the serial system described in [FLM87]. Readers familiar with the system may skip to Section 7.4.

7.3.1. Transactions

A non-access transaction T is modelled as an I/O automaton, with the following operations.

Input Operations:
CREATE(T)
REPORT_COMMIT(T',v), for T' a child of T, and v a value
REPORT_ABORT(T'), for T' a child of T
Output Operations:
REQUEST_CREATE(T'), for T' a child of T
REQUEST_COMMIT(T,v), for v a value

The CREATE input operation "wakes up" the transaction. The REQUEST_CREATE output operation is a request by T to create a particular child transaction. The REPORT_COMMIT input operation reports to T a successful completion of one of its children, and returns a value recording the results of that child's execution. The REPORT_ABORT input operation reports to T an unsuccessful completion of one of its children, without returning any other information. The REQUEST_COMMIT operation is an announcement by T that it has finished its work, and includes a value recording the results of that work.

7.3.2. Basic Objects

Since access transactions model abstract operations on a shared data object, we associate a single I/O automaton with each object, rather than one for each access. Thus, a basic object X is modelled as an automaton, with the following operations.

Input Operations:
CREATE(T), for T an access to X
Output Operations:
REQUEST_COMMIT(T,v), for T an access to X

Although we give these operations the same names as the operations of non-access transactions, it is helpful to think of the operations of access transactions in other terms also: a CREATE corresponds to an invocation of an operation on the object, while a REQUEST_COMMIT corresponds to a response by the object to an invocation.

7.3.3. Serial Scheduler

The serial scheduler is also modelled as an automaton. While the transactions and basic objects have been specified to be any I/O automata whose operations and behavior satisfy simple syntactic restrictions, the serial scheduler is a fully specified automaton, particular to each system type. It runs transactions according to a depth-first traversal of the transaction tree. The serial scheduler can choose nondeterministically to abort any transaction after its parent has requested its creation, as long as the transaction has not actually been created. Each child of T whose creation was requested must be either aborted or run to commitment with no siblings overlapping its execution, before T can commit. The operations of the serial scheduler are as follows.

Input Operations:
REQUEST_CREATE(T)
REQUEST_COMMIT(T,v)
Output Operations:
CREATE(T)
COMMIT(T), $T \neq T_0$
ABORT(T), $T \neq T_0$
REPORT_COMMIT(T,v), $T \neq T_0$
REPORT_ABORT(T,v), $T \neq T_0$

T_0 in the above definition is the root transaction in the transaction tree. The REQUEST_CREATE and REQUEST_COMMIT inputs are intended to be identified with the corresponding outputs of transaction and object automata, and similarly for the CREATE, REPORT_COMMIT and REPORT_ABORT output operations. The COMMIT and ABORT operations are internal, marking the point in time where the decision on the fate of the transaction is irrevocable.

A state s of the serial scheduler consists of the following sets:
(1) *s.create_requested*
(2) *s.created*
(3) *s.commit_requested*
(4) *s.committed*
(5) *s.aborted*
(6) *s.returned*

The set *s.commit_requested* is a set of (transaction, value) pairs. The others are sets of transactions. There is exactly one initial state, in which the set *create_requested* is $\{T_0\}$, and the other sets are empty.

The transition relation consists of exactly those triples (s',π,s) satisfying the preconditions and generating the effects given below, where π is the indicated operation. For brevity, we include in the effects only those components of state s which may change with the operation. If a component of s is not mentioned in the effects, it is implicit that the component is the same in s' and s.

* REQUEST_CREATE(T)

Preconditions:

$s.create\_requested = s'.create\_requested \cup \{T\}$

* REQUEST_COMMIT(T,v)

Effects:

$s.commit\_requested = s'.commit\_requested \cup \{(T,v)\}$

* CREATE(T)

Preconditions:

$T \in s'.create\_requested - (s'.created \cup s'.aborted)$

$siblings(T) \cap s'.created \subseteq s'.returned$

Effects:

$s.created = s'.created \cup \{T\}$

* COMMIT(T), $T \neq T_0$

Preconditions:

$(T, v) \in s'.commit\_requested$ for some v

$T \notin s'.returned$

$children(T) \cap s'.create\_requested \subseteq s'.returned$

Effects:

$s.committed = s'.committed \cup \{T\}$

$s.returned = s'.returned \cup \{T\}$

* ABORT(T), $T \neq T_0$

Preconditions:

$T \in s'.create\_requested - (s'.created \cup s'.aborted)$

$siblings(T) \cap s'.created \subseteq s'.returned$

Effects:

$s.aborted = s'.aborted \cup \{T\}$

$s.returned = s'.returned \cup \{T\}$

* REPORT_ABORT(T), $T \neq T_0$

Preconditions:

$T \in s'.aborted$

* REPORT_COMMIT$(T,v), T \neq T_0$

Preconditions:

$T \in s'.committed$

$(T,v) \in s'.commit\_requested$

The input operations, REQUEST_CREATE and REQUEST_COMMIT, simply result in the request being recorded. A CREATE operation can only occur if the corresponding REQUEST_CREATE has occurred and the CREATE has not already occurred. The second precondition on the CREATE operation says that the serial scheduler does not create a transaction until all its previously created sibling transactions have returned. That is, siblings are run sequentially. The preconditions on the COMMIT operation say that the scheduler does not allow a transaction to commit until all its children have returned. The preconditions on the ABORT operation say that the scheduler does not abort a transaction while there is activity going on on behalf of any of its siblings. That is, aborted transactions are dealt with sequentially with respect to their siblings. The result of a transaction can be reported to its parent at any time after the (purely internal) commit or abort has occurred. In particular, two siblings might run in one order and be reported to their parent in the opposite order.

7.3.4. Serial Systems and Serial Schedules

The composition of transactions with basic objects and the serial scheduler for a given system type is called a *serial system*, and its operations and schedules are called *serial operations* and *serial schedules*, respectively.

7.4. FLAT FILE LOCKING (FFL) SYSTEM

Let $X.k$ denote a key k in search structure X. Then $D = \{X.k$ for all X and k such that X is a search structure containing $k\}$ forms a database similar to a conventional database, in which each data item is directly accessible. "Searching" is thus a single step. We shall call each object $X.k$ in D a **FFL object**.

To simplify the system, we model both insert and delete operations on a key by write operations. We achieve this by assuming that a key can take a fictitious value of "nil", which indicates that the key is actually absent from the search structure. Insert is then writing into a key which initially has value "nil", delete is assigning the value "nil" to the key.

The generalized 2-phase locking mechanism [Mos85] is used on the objects in D, considered as independent data items.

A **FFL system** consists of transaction automata and automata representing FFL objects communicating with a scheduler. The scheduler (automaton) controls communications among the other components, thereby controlling the order in which the transactions create child transactions or access objects. A FFL system is the composition of a set of I/O automata, and is identical to the R/W locking system of [FLM87], with the FFL object automata corresponding to the automata for R/W locking objects. The following is a detail description of a FFL system. Readers familiar with the R/W locking system may skip to the Section 7.5.

7.4.1. FFL objects

For each object $X.k$ in D, we define a FFL object $M(X.k)$ which provides a resilient lock-managing variant of $X.k$. It receives operation invocations and responds like a basic object in the serial system, and also receives information about the fate of transactions so that it can maintain its locking and state restoration data.

$M(X.k)$ has the following operations.

Input Operations:
CREATE(T), for T an access to $X.k$
INFORM_COMMIT_AT($X.k$)OF(T), $T \neq T_0$
INFORM_ABORT_AT($X.k$)OF(T), $T \neq T_0$
Output Operations:
REQUEST_COMMIT(T,v), for T an access to $X.k$

A state s of $M(X.k)$ consists of the following five components:
(1) *s.write-lockholders*,
(2) *s.read-lockholders*,
(3) *s.create_requested*,
(4) *s.run*, which are sets of transactions,
(5) *s.map*, which is a function from *write-lockholders* to states of $X.k$

We say that two locks on the same object *conflict* if they are held by different transactions and at least one is a write-lock. The initial states of $M(X.k)$ are those in which *write-lockholders* $=\{T_0\}$ and *map* (T_0) is an initial state of the basic object $X.k$, and the other components are empty. The transition relation of $M(X.k)$ is given by all triples (s',π,s) satisfying the following preconditions and effects, given separately for each π. As before, any component of s not mentioned in the effects is the same in s as in s'.

* CREATE(T), T an access to $X.k$

Effects:

s.create_requested = *s'.create_requested* $\cup$ $\{T\}$

* INFORM_COMMIT_AT($X.k$)OF(T), $T \neq T_0$

Effects:

```
if $T \in s'$.write-lockholders then
 begin
  s.write-lockholder = (s'.write-lockholders - $\{T\}$) $\cup$ $\{parent(T)\}$
```

```
    s.map (U) = s'.map (U) for U ∈ s.write-lockholders - {parent(T)}
    s.map (parent (T)) = s'.map (T)
   end
  if T ∈ s'.read-lockholders then
   begin
    s.read-lockholders = (s'.write-lockholders - {T}) ∪ parent (T)
   end

* INFORM_ABORT_AT(X.k)OF(T), T ≠ T_0
 Effects:
  s.write-lockholders = s'.write-lockholders - {descendants(T)}
  s.read-lockholders = s'.read-lockholders - {descendants(T)}
  s.map (U) = s'.map (U) for all U ∈ s.write-lockholders

* REQUEST_COMMIT(T,v) for T a write access to X.k
 Preconditions:
  T ∈ s'.create_requested − s'.run
  s'.write-lockholders ∪ s'.read-lockholders ⊆ ancestors (T)
  (s'.map(least(s'.write-lockholders)), CREATE(T), t)
       and (t, REQUEST_COMMIT(T,v), t')
       are in the transition relation of basic object X.k, for some t
 Effects:
  s.run = s'.run ∪ {T}
  s.write-lockholders = s'.write-lockholders ∪ {T}
  s.map (U) = s'.map (U) for all U ∈ s.write-lockholders - {T}
```

s.map $(T) = t'$

* REQUEST_COMMIT(T,v) for T a read access to $X.k$

Preconditions:

$T \in s'.create\_requested - s'.run$

$s'.write\text{-}lockholders \subseteq ancestors(T)$

$(s'.map(least(s'.write\text{-}lockholders)), CREATE(T), t)$

and $(t, REQUEST\_COMMIT(T,v), t')$

are in the transition relation of basic object $X.k$, for some t

Effects:

$s.run = s'.run \cup \{T\}$

$s.read\text{-}lockholders = s'.read\text{-}lockholders \cup \{T\}$

When an access transaction is created, it is added to the set *create_requested*. A response, containing return value v, to an access T can be returned only if (a) the access has been requested but not yet responded to, (b) every holder of a conflicting lock is an ancestor of T, (c) every holder of a conflicting lock is an ancestor of T, and (d) v is a value which can be returned by basic object $X.k$ in the response to T from some state t, obtained by performing CREATE(T) in the state *map(least(write-lockholders)).* When a response is given, the access transaction is added to the set *run* and granted the appropriate lock, and if the transaction is a write access, the resulting state is stored as *map* (T). If the transaction is a read access, no change is made to the stored state of basic object $X.k$, i.e. to *map*.

When the FFL object is informed of the abortion of a transaction, it removes all locks held by the descendants of the transaction. When it is informed of a commit, it passes any locks held by the transaction to the parent, and also passes the version stored in *map*, if there is one.

7.4.2. Generic Scheduler

The generic scheduler is a very nondeterministic automaton. It passes requests for the creation of sub-transactions or accesses to the appropriate recipient, passes responses back to the caller and informs objects of the fate of transactions, but may delay such messages for arbitrarily long time or unilaterally decide to abort a subtransaction which has been created.

The generic scheduler has nine operations:

Input Operations:
- REQUEST_CREATE(T)
- REQUEST_COMMIT(T,v)

Output Operations:
- CREATE(T)
- COMMIT(T), $T \neq T_0$
- ABORT(T), $T \neq T_0$
- REPORT_COMMIT(T,v), $T \neq T_0$
- REPORT_ABORT(T), $T \neq T_0$
- INFORM_COMMIT_AT(X)OF(T), $T \neq T_0$
- INFORM_ABORT_AT(X)OF(T), $T \neq T_0$

These play the same roles as in the serial scheduler, except for the INFORM_COMMIT and INFORM_ABORT operations which pass information about the fate of transactions to the FFL objects.

Each state s of the generic scheduler consists of six sets:

(1) *s.create_requeste*,
(2) *s.created*,
(3) *s.commit_requested*,
(4) *s.committed*,
(5) *s.aborted*, and
(6) *s.returned*.

The set *s.commit_requested* is a set of (transaction, value) pairs, and the others are sets of transactions. All are empty in the initial state except for *create_requested*, which is $\{T_0\}$.

The operations are defined by preconditions and effects as follows:

* REQUEST_CREATE(T)

Effects:

$s.create\_requested = s'.create\_requested \cup \{T\}$

* REQUEST_COMMIT(T,v)

Effects:

$s.commit\_requested = s'.commit\_requested \cup \{(T,v)\}$

* CREATE(T), T a transaction

Preconditions:

$T \in s'.create\_requested - s'.created$

Effects:

$s.created = s'.created \cup \{T\}$

* COMMIT(T), $T \neq T_0$

Preconditions:

$(T,v) \in s'.commit\_requested$ for some v

$T \notin s'.returned$

$children(T) \cap s'.create\_requested \subseteq s'.returned$

Effects:

$s.committed = s'.committed \cup \{T\}$

$s.returned = s'.returned \cup \{T\}$

* ABORT(T), $T \neq T_0$

Preconditions:

$s.aborted = s'.aborted \cup \{T\}$

$s.returned = s'.returned \cup \{T\}$

* REPORT_COMMIT(T,v), $T \neq T_0$

Preconditions:

$T \in s'.committed$

$(T, v) \in s'.commit\_requested$

* REPORT_ABORT(T), $T \neq T_0$

Preconditions:

$T \in s'.aborted$

* INFORM_COMMIT_AT(X)OF(T), $T \neq T_0$

Preconditions:

$T \in s'.committed$

* INFORM_ABORT_AT(X)OF(T), $T \neq T_0$

Preconditions:

$T \in s'.aborted$

7.4.3. FFL Systems

The composition of transactions with FFL objects and the generic scheduler is called a *FFL system.*

The main result of [FLM87] is a proof of serial correctness for the R/W locking system: Every schedule of the R/W locking system is serially correct for every non-orphan (see Section 7.6), non-access transaction.

The serial correctness of the FFL system follows directly from the above result. In Section 7.6, we shall define a stronger correctness condition and prove that schedules generated by the FFL system (and hence the R/W locking system) are correct in this new sense.

7.5. B-TREE LOCKING SYSTEM

7.5.1. $\mathrm{B}^{\mathrm{link}}$ tree

A B*-tree [Wed74] serves as a *dense index*, i.e., the leaves contain pairs (k, p), where p points to the record with key k.[33] The internal vertices have i keys and $i+1$ pointers, where $c \le i \le 2c$ for some fixed $c > 1$, e.g., $p_0 k_1 p_1 k_2 \cdots k_i p_i$. where $k_1 < k_2 < ... < k_i$. During a search for a key k, we follow the link p_j provided that $k_j < k \le k_{j+1}$, and we may assume that k_0 is $-\infty$ and k_{i+1} is $+\infty$. For each pair (p_i, k_{i+1}), k_{i+1} is the highest key in the vertex pointed to by p_i.

The $\mathrm{B}^{\mathrm{link}}$ tree [LeY81] are obtained from the B*-tree by adding to each vertex an additional pair (k_{i+1}, p_{i+1}). The key k_{i+1} is the *high key* of the vertex, i.e., the highest key in the subtree rooted at that vertex. The pointer p_{i+1} points to the next vertex at the same level, and we call it a *link*. Thus, starting at the leftmost vertex at any level, we can traverse all the vertices at that level by following the links. Another modification, proposed in [Sag86], is to allow internal vertices to have less than $c+1$ children temporarily. Figure 7.1 shows an example of $\mathrm{B}^{\mathrm{link}}$ tree taken from [Vid87], where M is a marker indicating a leaf vertex.

We have taken some of the following definitions from [GoS85].

We assume the keys come from a possibly infinite, totally ordered set of keys called *KeySpace(X)*, and values that are stored with keys come from a possibly infinite set called *ValueSpace(X)*. A *state* of a

[33] We interpret p as the *value* associated with key k.

B^{link} tree consists of a five tuple $(V, E, root, contents, edgeset)$, where V is a set of vertices,[34] E is a set of directed edges, and *root* is a distinguished member of V. *contents* is a function from V to subsets of $KeySpace(X) \times ValueSpace(X)$. *edgeset* is a function from the edges in E to subsets of $KeySpace(X)$. The global contents, *GlobalContents(X)*, is the union of the contents in all the vertices, i.e., $GlobalContents(X) = \{contents(v) \mid v \in V\}$. Let *GCSpace(X)* be the set of all possible *GlobalContents* (X)'s, and *StateSpace(X)* be the set of all possible states of X.

Informally, $contents(v)$ is a set of key-value pairs and tells us what is stored in vertex v, $edgeset(e)$ tells us what range of keys can be found in the subtree reached by traversing edge e, and *GlobalContents* (X) gives what is stored in the entire B^{link} tree X. In the next section, we give a description of the B^{link} tree scheme in [Sag86].

7.5.2. B^{link} tree Algorithm

7.5.2.1. Concurrent Searches and Insertions

The procedure for a search in a B^{link} tree is given in Figure 7.2 (it is taken from [LeY81] and [Sag86]). The function $next(A, k)$ accepts the data of a vertex A and a value k, and returns either a pointer to the next level or the link of A (if k is larger than the high key of A). The procedure *movedown* starts at the root, and moves down the levels of the tree (using pointers and links) until a leaf is reached. The procedure *moveright* follows links until the leaf where k belongs is reached.

An insertion of a new record r, having a key value k, begins with a search for the leaf where k belongs (see Figure 7.3). The procedure *movedown_and_stack* is similar to *movedown* with the addition of stacking a pointer to the last vertex visited at each nonleaf level. Once we reach a leaf, we lock it and check whether k should be inserted into that leaf. Insertion is done only if k is neither in that leaf nor

[34]To avoid confusion between the transaction tree and a B-tree, we use "vertex" instead of "node", which is reserved for the transaction tree.

```
procedure movedown;
begin
current := root;
A := get(current);
while A is not a leaf do
    begin
    current := next(A,k);
    (* the function next(A,k) returns either a pointer to the next level
      or the link of A. *)
    A := get(current)
    end;
end;

procedure moveright;
begin
while t :=next(A,k) is a link do
    begin
    current := t;
    A := get(current);
    end;
end;

procedure search (k:keyvalue)
begin
movedown;
moveright;
if k is in A then return pointer to record
else return nil        (* not found *)
end;
```

Figure 7.2 Procedure for a Search (Read)

larger than its high key. If k does not belong to that leaf, than we have to unlock it and call the procedure *moveright* again. Whenever *moveright* finds the leaf, A, where k belongs, we lock A and check again whether k belongs to A, since A might be split between the time we first read it and the moment we lock it.

When writing into A, three cases are possible and they are handled by the procedures in Figure 7.4. If A is *safe*, i.e., has fewer than $2c$ pairs, then we call the procedure *write_into_safe*. If A is *unsafe*, then splitting has to be done, creating a new vertex, and we have to insert another pair at the next higher level to point to the new vertex. A special case is where a new root has to be created, and this is handled by the procedure *write_into_unsafe_root*. In this case we have to lock A until the new root is created, in order to avoid the creation of two roots simultaneously. If the next higher level already exists, then we call the procedure *write_into_unsafe*. The vertex where the next insertion takes place is either the parent vertex of A at the time of splitting (i.e., the one through which we came down, we store it on a stack) or further to the right as a result of vertex splitting. Thus, we pop the stack and repeat the main loop of the procedure *write*. There is one minor detail of handling the case where the stack is empty although an insertion at a higher level is required. This may occur when the number of levels in the tree has been increased while our process is running. Thus, if the stack is empty, then at the end of the procedure *insert_into_unsafe* we assign *current* a pointer to the leftmost vertex at the next higher level.

7.5.2.2. Concurrent Deletion

Deletions are handled by removing the key value and the pointer of the deleted record from the leaf where they are stored. There is a special *compression process* that redistributes the data in the tree so that each vertex has at least k pairs. This process can run concurrently with searches, insertions and deletions.

The idea is to lock a vertex F and two adjacent children of F, say A and B. If both A and B have fewer than c pairs, then the data in A and B are rearranged (and possibly B is deleted), and some required changes are made in F. Immediately after A and B are examined, we unlock F, A, and B, before locking F again and examining the next pair of children. This is done to ensure that insertion processes that have procedure *compress_level*(i), given in Figure 7.5, visit each vertex, F, at level $i+1$,

```
procedure movedown_and_stack;
begin
initialize stack;
current := root;
A := get(current);
while A is not a leaf do
   begin
   t := current;
   current := next(A ,k);
   if current is not a link
   then push (stack ,t);
   A := get(current)
   end;
end;

procedure write(r :key);
begin
p := pointer to record with key k;
completed := false;
movedown_and_stack;
repeat
   repeat
      found := true;
      lock(current);
      A := get(current);
      if k > highkey(A) then begin
                             unlock(current);
                             found := false
                             moveright;
                             end;
    until found;
    if A is safe then write_into_safe
    else if A is not the root
         then write_into_unsafe
         else write_into_unsafe_root
until completed;
```

Figure 7.3 Procedure for Write

```
procedure write_into_safe;
begin
write the pair (k,p) into A;
(* that is, if k was not in A, then insert (k,p);
  if (k,p') was in A, then replace p' with p *)
unlock(current);
completed := true;
end;

procedure write_into_unsafe;
begin
B := a new vertex;
q := pointer to B;
A,B := rearrange old A, adding (k,p);
link of B := link of A;
link of A := pointer to B;
unlock(current);
p := pointer to B;
k := highkey(A);
if stack is not empty
then current := pop(stack);
else current := pointer to leftmost vertex at next higher level;
end

procedure write_into_unsafe_root;
begin
B := new vertex
q := pointer to B;
A,B := rearrange old A, adding (k,p);
link of A := pointer to B;
k:=highkey(A);
u:=highkey(B);
R := new root;
R := (current,k,q,u,nil)
update the information about the number of levels in the tree;
unlock(current);
completed := true;
end;
```

Figure 7.4 Three cases in writing

and examine pairs of adjacent children of F. (If F has an odd number of children, then the last one will not be compressed even if it has fewer than c pairs.) The complete process consists of applying *compress_level* to all the levels of the tree, except for the root, starting at level 0 (i.e., the leaves). As for the root, after applying *compress_level* to the level below it, we examine the root and if it has only one child, then the root is removed and its child becomes the new root.

Suppose that a tree T becomes empty as a result of deletions. One pass of *compress_level* over all the levels of T is not going to reduce the tree to a single vertex; rather, $\log_2 n$ passes over the tree are required, where n is the number of leaves in T. In a typical environment the number of insertions far exceeds the number of deletions, and running the compression process in the background as a low priority job is expected to allow only a small percentage of the vertices to be less than half full.

Some changes in the data structure are required. First, an additional bit is needed for each pointer stored in the tree. When the vertices A and B are examined, we mark the bits for the pointers to A and B that are stored in their common parent F. Secondly, in each vertex we stored the high key of its left neighbor (denoted k_0), and each vertex has a deletion bit indicating whether the vertex is deleted.

The procedure *compress_level*(i) locks a vertex, F, at level $i+1$, and reads it. If F has a tail (on the right) of unmarked pointers, then *compress_level* chooses the leftmost pointer in the tail, locks the vertex, A, it points to, and then reads A. The third vertex to be locked, B, is the one pointed to by the link of A. If F does not have a pointer to B, then two cases are possible depending on whether the pointer to B should be stored in F. If it should not, then we unlock all three vertices and move to the next vertex at level $i+1$. If it should, then all three vertices are unlocked and *compress_level* waits until the pointer to B is inserted into F. In practice, *compress_level* is stopped for a while before locking the three vertices again and checking that F has a pointer to B. It is possible that *compress_level* will be waiting forever, because of constant splitting of vertex A, but it is expected that the chances of that happening are minuscule. Once the pointer to B is in F, we check whether A and B have to be rearranged. If so, then two

```
procedure compress_level(i);
begin
current := pointer to leftmost vertex at level i+1;
while current ≠ nil do
   begin
   lock(current);
   F := get(current);
   if F has an unmarked tail then
   (* unmarked tail is sequence of unmarked pointers to children,
     where a pointer is marked if the corresponding child is examined *)
      begin
      P_1 := leftmost pointer in the unmarked tail of F;
      lock(P_1);
      A := get(P_1);
      P_2 := link of A;
      if P_2 = nil then return;
      lock(P_2);
      B := get(P_2);
      if P_2 is in F then
           begin
           rearrange A and B if necessary;
           unlock(current);
           unlock(P_1);
           unlock(P_2);
           end
      else begin
           unlock(current);
           unlock(P_1);
           unlock(P_2);
           if P_2 should be stored in F
           then wait
           else current := link of F;
           end
      end
  else begin  (* F has no unmarked tail *)
         unlock(current);
         current := link of F;
         end;
  end;
end;
```

Figure 7.5 Procedure for the compression process

cases are possible.

(1) If there are no more than $2k$ pairs in A and B, then all the pairs from B are moved to A (including the high key and link of B), the deletion bit in B is set, the pointer and high key for A in F are updated, and the pointer to A in F is marked. Finally, F, A, and B are rewritten and unlocked in that order.

(2) If there are more than $2c$ pairs in A and B, then A and B are rearranged so that each one will have at least c pairs, the high key of A is updated in A, B and F, and the pointers to A and B in F are marked. Finally, F, A, and B are rewritten and unlocked in that order.

Immediately before any one of A, B and F is rewritten and after all of them are rewritten, the tree has a valid *B–Tree* structure. Further, insertion into or deletion from any one of A, B and F does not interfere with the rewriting of these vertices, since it requires a lock. Thus, we only have to consider what happens when a process reads one of A, B and F before it is rewritten and another process (that may be the same as the first process) reads one of A, B and F after it is rewritten. Several cases are possible, but the only two that cause a problem are as follows.

(1) A process reads a deleted vertex.

(2) A process reads a vertex in search of a value k, and k should be left to that vertex, i.e., $k \leq k_0$.

Obviously, both of these cases are easy to detect. If either one happens to any process, then the simplest solution is to backtrack to the previous vertex visited, and only if we cannot resume the search from that vertex, we should restart at the root. A process must be extremely slow compared to the compression process in order for the backtracking to the previous vertex to fail (because the previous vertex may be compressed again, but the compression of the vertex takes place after all other vertices have been visited once by the compression process).

A minor detail is the following: When a vertex is deleted, we cannot remove it, because other process may have to read it. One solution is to record in the vertex the time of its deletion, and also store for each running process its starting time. A deleted record can be released when all the currently running processes have started after its deletion time. Another solution is to use a constant t_0, and restart a process that does not finish within t_0 seconds (an insertion process is restarted as described earlier). In this way, a deleted vertex can be release after t_0 seconds.

7.5.3. Automaton Model of the B-link Tree Algorithm

The B^{link} tree scheme in the previous section handles only simple operations of read/write of a single data object (key). The algorithm that we model using the I/O automata is a modified version which handles nested transactions accessing B-trees. We have included a 2-phase locking scheme for subtransactions in the transaction tree. The resulting system is called a B-tree locking (BTL) system, which consists of a set of I/O automata. This set contains a transaction automaton for each internal vertex (which represents a "non-access" transaction) of the transaction tree, a BTL object automaton for each vertex in each B^{link} tree, and a scheduler. In this system, there are locks on individual keys stored in vertices as well as locks on vertices.

7.5.3.1. Transactions

As in [FLM87] the I/O automaton modelling a transaction T has the following operations.

Input Operations:
- CREATE(T)
- REPORT_COMMIT(T', r), where $T' \in children(T)$
- REPORT_ABORT(T'), where $T' \in children(T)$

Output Operations:
- REQUEST-CREATE(T'), where $T' \in children(T)$
- REQUEST-COMMIT(T, r)

The states and state transitions of a transaction depends on the function of the transaction and is left unspecified.

A transaction trying to access some key in a B^{link} tree creates a child transaction, which is either a read access manager or a write access manager. These transactions will in turn create access transactions to vertices of the B-tree (see Figure 7.6(a)). In the figure, E at the root stands for "environment", U stands for "user transaction", *W-AM* (*R-AM*) stands for write (read) access managers, and *VM* stands for vertex manager.

The definition of the well-formed schedules of transaction automata is the same as that for a FFL system (or R/W locking system [FLM87]), except for a read or write access manager. A schedule of a read or write access manager is **well-formed** if it is *serial* in the sense of [FLM87] and all search accesses are created before the read or write access.

7.5.3.2. Read Access Manager

Let X be a B^{link} tree. The purpose of a **read access manager,** *R-AM(X)*, is to perform a rcad acccss to some key, $X.k$, in X. A *R-AM(X)*, T, invokes *search-read* (search to read) accesses to BTL objects (i.e., vertex managers, see Section 7.5.3.4), $VM(v_1)$, ..., $VM(v_m)$ for some vertices $v_1, ..., v_m$ in X, where v_1 is the root of X and the last vertex v_m is the one that holds the value of $X.k$ if k is contained in X. Accesses to $v_1, ..., v_{m-1}$ are non-decisive search operations and the read access to v_m is a decisive operation. Note that (k, nil) in a vertex, if any, represents the fact that key k is not in X.

R-AM(X) has the following operations.

Input Operations:
CREATE(T), with $root(T)$ = root of X
REPORT_COMMIT(T',r), with *type(T')=search-read*
REPORT_ABORT(T')
Output Operations:
REQUEST_CREATE(T'), with *type(T')=search-read* and *key(T')=k*
REQUEST_COMMIT(T,r)

In the above, when *type(T')=search-read* and *key(T') = k*, we mean that *T'* is a "search-read" access (there are other types of accesses) to some vertex *v*; *T'* is searching for a key in vertex *v* and the key to look for is equal to *k*. Here we have adopted a different convention of expressing parameters of an operation so that when we state an operation as in REQUEST_CREATE(*T'*), with *type(T')=search-read* and *key(T')=k* it is same as if *key(T')* is passed as a parameter as in REQUEST_CREATE(*T'*,*key*) where *T'* is a "search-read" access and *key* = *k* appears as a precondition.

We assume that the scheduler maintains a list of pointers to the roots of all B-trees, which can be passed as a parameter *root(T)*. I.e., a state component of the scheduler is *s.root_of_* (*X*) for each B-tree X, and when the scheduler outputs CREATE(*T*), where *T* is a *R-AM(X)*, the precondition that *root* (*T*) = *s'.root_of_* (*X*) must hold. Transactions can modify B-trees but do not introduce new B-trees.

A state *s* of transaction *R-AM(X)* consists of the following components:
(1) *s.data*
(2) *s.phase*, which can take one of "*idle*", "*searching*", or "*finished*".
(3) *s.nextvertex*
(4) *s.created*, a boolean variable.

s.data will store the return value for the read access. *s.nextvertex* keeps track of the next vertex during the search for the desired key *X.k*, starting at the root of B^{link} tree *X*. For the initial state s_0 of *R-AM(X)*, s_0.*created* = *false* and the other components are undefined. The transition relation of *T* = *R-AM(X)* is given by triples <*s'*, π, *s*> having the following preconditions and effects, given separately for each π, where *T'* is an access that is a child transaction of *T*.

* CREATE(*T*) with *root(T)*=root of *X*.

Preconditions:

s'.created = *false*

Effects:

s.phase := *"searching"*

s.nextvertex := *root(T)*

s.data := *nil*

s.created := *true*

* REQUEST-CREATE(*T'*) with *type(T')=search-read* and *key(T')=k*

(Each access *T'* with *type(T')* = *search-read* has an associated parameter of *key*(*T'*), which is the key to be searched for.)

Preconditions:

$T' \in$ *accesses(s'.nextvertex)*

(* *accesses(v)* is the set of all accesses to vertex v *)

s'.phase = *"searching"*

k = *key(T)*

* REPORT_COMMIT(*T'*, *r*) with *type(T')* = *search-read* and *r* = *(r(1), r(2))*

(*r*(1) and *r*(2) are returned by a "vertex manager" (see Section 7.5.3.4). *r*(1) = "*found*" means the key has been located.)

Preconditions:

s'.phase = *"searching"*

Effects:

```
if r(1) = "found"
  then { s.phase := "finished" ; s.data := r(2) }
if r(1) = "not-found" then s.nextvertex:= r(2)
```

* REPORT_ABORT(*T'*)

Effects: *none*

* REQUEST-COMMIT(*T*, *r*)

Preconditions:

s'.phase = *"finished"*

$r = s'.data$

Effects:

$s.phase :=$ *"idle"*

7.5.3.3. Write Access Manager

The purpose of a **write access manager,** *W-AM(X),* for search structure X is either to store a new key or to change the value of an existing key in X. Each *W-AM(X)*, T, has associated parameters $key(T)$ and $data(T)$. To delete $key(T)$, $data(T) = nil$ is stored with $key(T)$. When the value of a key is *nil*, then the key is not counted in the "size" of the vertex (size is the deciding factor in the splitting and merging of vertices). A *W-AM(X)* invokes *search-write* (search to write) accesses to $VM(v_1)$, ..., $VM(v_m)$ for some vertices $v_1, ..., v_m$ in X, where v_1 is the root of X and $key(T)$ is to be stored or updated in the last vertex v_m. Accesses to $v_1, ..., v_{m-1}$ are non-decisive, and the write access to v_m is decisive.

The operations of *W-AM(X)* are similar to those of *R-AM(X)* except that search-write accesses instead of search-read accesses appear in the REQUEST_CREATE and REPORT_COMMIT operations. The state s of a *W-AM(X)* consists of the following components:

(1) *s.phase*, which takes one of *"idle"*, *"searching"*, *"finished"* or *"split"*
(2) *s.nextvertex*
(3) *s.created*

For the initial state s_0 of *W-AM(X)*, $s_0.created = false$ and the other components are undefined. The preconditions and effects of each transition $<s', \pi, s>$ for each operation π are similar to those for *R-AM(X)*, except that value to be written is a parameter, $data(T')$, of REQUEST__CREATE(T') and $r = nil$ is returned in REQUEST_COMMIT(T', r). We omit the details.

7.5.3.4. B-tree Locking Objects (Vertex Managers)

In our system both vertices and keys of B^{link} trees are objects. Individual keys stored in a vertex, as well as vertices are lockable. Each vertex is represented by an I/O automaton but individual keys in it are not represented by automata. Instead, the automaton for a vertex also manages the keys in the vertex. Keys will be called **user-visible** objects and vertices **user-invisible** objects. (The notion of user-visibility will be important in our definition of correctness.) We define, for each vertex v in a B^{link} tree X, a vertex manager automaton $VM(v)$, which provides resilient lock management for the keys in the vertex. $VM(v)$'s for leaves v contain key-value pairs (k,c).

Since a key $X.k$ is a "resilient object" [LyM86], different transactions may "see" different **versions** of the same key (i.e., values associated with the key). However, there is only one version of each B^{link} tree structure. Therefore, at any time, the set of all keys contained in X is partitioned among the leaves of the current B^{link} tree, and each key in a leaf has a set of versions associated with it. When a key moves from a vertex to another as a result of splitting or merging operation (see Sections 7.5.3.4.2 and 7.5.3.4.3), all its versions move with it.

Most frequent accesses to vertex v are of type search-read or search-write, and they interact with automaton $VM(v)$ via the scheduler. (Later we shall see other accesses for vertex splitting and merging.) A search-read or search-write access T with $key(T)=k$ to vertex v is also an access to key k if v is a leaf and key k currently resides in v.

Unlike previous I/O automaton systems [FLM88], some objects, i.e., vertices, in a BTL system are not permanent. We allow search-write transactions to perform operations which may *remove* existing vertices and *construct* new ones.

$VM(v)$ has the following operations.
Input Operations:
CREATE(T), where $T \in accesses(v)$
INFORM-COMMIT-AT($X.k$)OF(T), $T \neq T_0$
INFORM-ABORT-AT($X.k$)OF(T), $T \neq T_0$
Output Operations:
REQUEST-COMMIT(T,v), where $T \in accesses(v)$

where T_0 is the root transaction in the transaction tree and $accesses(v)$ is the set of accesses to vertex v.

A state s of $VM(v)$ for a non-leaf v consists of the following components:

(1) *s.created*: contains accesses to v that have been created.

(2) *s.run*: contains accesses to v on whose behalf $VM(v)$ has output a REQUEST_COMMIT.

(3) *s.status* includes the following information:

s.edges: consists of the edges emerging from v, it is sufficient to record the identities of vertices connected by outgoing edges from v,

s.edgesets: consists of the edgesets of edges emerging from v,

s.outdeg: the number of edges emerging from v,

s.parent: the parent vertex of v,

s.next−neighbour: the vertex pointed to by the link.

For each non-leaf vertex v that exists initially, s_0.*status* reflects the initial state of v, and the other components are empty or undefined. A state s of $VM(v)$ for a leaf v consists of the following components:

(1) *s.created*: contains accesses to v that have been created.

(2) *s.run*: contains accesses to v on whose behalf $VM(v)$ has output a REQUEST_COMMIT.

(3) *s.status* includes the following information:

s.keyset: keys that are in v (value of a key can be *nil*),

s.next−neighbour: the vertex pointed to by the link,

s.edgeset: the edgeset of the edge (i.e., link) to the right neighbour,

s.parent: the parent vertex of v.

(4) *s.map*, which maps transactions to *contents*(v)

(if $(k,c) \in s.map(T)$, then we sometimes write $s.map(T,k) = c$; $s.map(U,k) = nil$ means that in state s, transaction U finds that key k is not present in v)

(5) *s.map-size*: maps each transaction T to the number of non-*nil* keys in v.

(6) *s.readlock-holders*(k), where $k \in s.keyset$,

(7) *s.writelock-holders*(k), where $k \in$ *s.keyset*,

For each leaf vertex v that exists initially, s_0.*status* reflects the initial state of v, i.e., s_0.*readlock-holders(k)* = s_0.*writelock-holders(k)* = $\{T_0\}$ for all $k \in$ s_0.*keyset*, s_0.*map* (T_0) and s_0.*map-size* reflect the initial contents of v and the other components are empty or undefined.

A new vertex v may be constructed by some search-write transactions T_a, as a result of splitting (see below). Then the initial state of $VM(v)$, s_0, is defined after the REQUEST_COMMIT of T_a. The transition relation of $VM(v)$ is given next.

7.5.3.4.1. Transitions of VM(v)

In the above, we defined the components of a state of $VM(v)$ for v being a vertex in a B^{link} tree. Here we define the transition relation of $VM(v)$. The transition relation of $VM(v)$ is given by all tuples $<s', \pi, s>$ satisfying the following preconditions and effects, given separately for each π.

* CREATE(T) with $key(T) = k$

Preconditions: $T \in accesses(v)$

Effects: *s.created* := *s'.created* $\cup$ *{T}*

To simplify the presentation, we define below two disjoint sets of preconditions for the same REQUEST_COMMIT with type *search-read.* It should be understood that an automaton triggers any transition whose preconditions are satisfied.

* REQUEST-COMMIT(T, r) with *type(T)=search-read* and $key(T) = k$

Preconditions:

v is a leaf

$T \in$ *s'.created* $-$ *s'.run*

$k \in$ *s'.keyset*

s'.writelock-holders(k) $\subseteq$ *ancestors(T)* *r(1) = "found"*

(* *r(1) = "found"* when the key required is in leaf vertex v;

r(2) then returns the value associated with the key. *)

r(2) = s'.map(least(s'.writelock-holders(k))

(* *least(s'.writelock-holders(k))* refers to the least ancestor of T in the transaction tree among *s'.writelock-holders(k).* *)

Effects:

s.readlock-holders(k) := s'.readlock-holders(k) $\cup$ *{T}* /* readlock given to T

s.run := s'.run $\cup$ *{T}*

* REQUEST-COMMIT(T, r) with *type(T)=search-read* and *key(T) = k*

Preconditions:

v is not a leaf or $k \notin$ *s'.keyset*

r(1) = "not-found"

(If v is not a leaf, then r (2) returns the vertex whose key range k is in.)

r(2) = v' if $k \in$ *s'.edgeset(e)* and *e = [v, v']*

Effects:

s.run := s'.run $\cup$ *{T}*

Again, to simplify the presentation, we define below two disjoint sets of preconditions for the same REQUEST_COMMIT with type *search-write.*

* REQUEST-COMMIT(T, r) with *type(T)=search-write*, *key*(T)=k, *data*(T)=d and r = (r (1), r (2))

Preconditions:

v is a leaf

$T \in$ *s'.created* – *s'.run*

$k \in$ *s'.keyset*

s'.map-size(T) < maxsize /* if not, see Splitting.

```
    $s'.writelock\text{-}holders(k) \cup s'.readlock\text{-}holders(k) \subseteq ancestors(T)$

    $r(1) = \text{"found"}$ and $r(2) = nil$

  Effects:

    if $s'.map(T,k)=nil$ then $s.map\text{-}size(T):=s'.map\text{-}size(T)+1$

    $s.map(T) := s'.map(least(s'.writelock\text{-}holders))$

    $s.map(T,k) := d$

    $s.writelock\text{-}holders(k):=s'.writelock\text{-}holders(k)\cup\{T\}$

    $s.run := s'.run \cup \{T\}$

* REQUEST-COMMIT($T$,$r$) with $type(T)=search\text{-}write, key(T) = k, data(T)=d$ and $r = (r(1), r(2))$

  Preconditions:

    $v$ is not a leaf or $k \notin s'.keyset$

    $r(1) = \text{"not-found"}$

    $r(2) = v'$

    where $k \in s'.edgeset(e)$, and $e = [v,v']$ is the leftmost edge from $v$ that can lead to $k$

  Effects:

    $s.run := s'.run \cup \{T\}$

* INFORM-COMMIT-AT($X.k$)OF($T$)

  Preconditions: $v$ is a leaf containing $X.k$

  Effects:

    if $T \in s'.writelock\text{-}holders(k)$ then

    begin

      $s.writelock\text{-}holders(k) := (s'.writelock\text{-}holders(k) - \{T\}) \cup \{parent(T)\}$   /* lock passed to parent

      $s.map(parent(T), k):=s'.map(T, k)$   /* "view" passed to parent

    end

    if $T \in s'.readlock\text{-}holders(k)$ then $s.readlock\text{-}holders(k) := (s'.readlock\text{-}holders(k)$
```

– {T}) ∪ {parent(T)} /* lock passed to parent

* INFORM-ABORT-AT(*X.k*)OF(*T*)

Preconditions: *v* is a leaf containing *X.k*

Effects:

s.writelock-holders(k):=s'.writelock-holders(k) – { descendants of T }

s.readlock-holders(k) := s'.readlock-holders(k) – { descendants of T }

7.5.3.4.2. Splitting

VM(*v*) may split itself in order to observe the size limit on vertices during insertion. Splitting a vertex *v* involves the construction of a new vertex *v'* containing all the keys of *v* higher than a certain "splitkey". The augmentation to the transition relation of *VM*(*v*) needed to account for splitting is given in the following.

* REQUEST-COMMIT(*T*, *r*) with *type(T) = search-write, key(T) = k, data(T)=d* and *T* ∈ *accesses(v)*

Preconditions:

T ∈ *s'.created – s'.run*

v is a leaf of *X*

s'.writelock-holders(k) ∪ *s'.readlock-holders(k)* ⊆ *ancestors(T)*

k ∈ *s'.keyset* and *s'.map(T,k) = nil*

s'.map-size(T) = maxsize

r(1) := "split"

r(2):=[s'.parent, splitkey, v']

(*splitkey* is a chosen key in *s'.keyset(T)*, and *v'* is a new vertex; if *v* is the root, then *s'.parent* is *nil*.)

Effects:

1) *VM(v')* for a new leaf *v'* is constructed with initial state s_0 as follows:

s_0.*status* is the initial status of vertex *v'* which contains the keys higher than *splitkey* in *s'.keyset* and the corresponding edges and edgesets from *s'.keyset*.

s_0*.created* and s_0*.run* inherit their values from *s'.created* and *s'.run* for transactions that access keys in s_0*.keyset*.

s_0*.readlock-holders* and s_0*.writelock-holders* also inherit their values from s'.

$s_0.map(T', k') := s'.map(T', k')$ for all $k' \in s_0.keyset$ and for all T'

s_0*.map-size(T')* := number of non-*nil* keys in s_0*.map* (*T'*) for all *T'*

```
if $k \in s_0.keyset$
then
   $s_0.map\text{-}size(T) := s_0.map\text{-}size(T) + 1$;
   $s_0.map(T, k) := d$
   $s_0.writelock\text{-}holders(k) := s_0.writelock\text{-}holders(k) \cup \{T\}$
   $s_0.run := s_0.run \cup \{T\}$
```

2) Changes to *VM* (*v*) :

s.status is the same as *s'.status* except that the keys higher than *splitkey* and the corresponding edges and edgesets are removed, and an additional *edge* [*v*, *v'*] and the corresponding *edgeset* are added.

s.created and *s.run* inherit their values from *s'.created* and *s'.run* for transactions that access the keys in the new *s.keyset*.

s.readlock-holders and *s.writelock-holders* also inherit their values from s'.

$s.map(T', k') := s'.map(T', k')$ for all $k' \in s.keyset$ and for all T'

s.map-size(T'):= number of non-*null* keys in *s.map(T')* for all *T'*

```
if $k \in s.keyset$
then
   s.map-size(T):=s.map-size(T)+1
   s.map(T,k):= d
```

s.writelock-holders(k):=s.writelock-holders(k)$\cup${T}

s.run:= s.run $\cup$ {T}

* REQUEST_COMMIT(*T*, *r*) with *type(T)=install-pointer, splitkey(T)=k,* and *newvertex(T) = v'*

Preconditions:

$T \in s'.created - s'.run$

$k \in s'.edgesets(e)$ for some emerging edge *e*

$r = nil$

$s'.outdeg < maxsize$

Effects:

$s.edges = s'.edges \cup \{[v,v']\}$

$s.edgesets := s'.edgesets \cup s.edgeset([v,v'])$

where edgeset([v,v']) is the set of keys $\geq k$

$s.outdeg := s'.outdeg + 1$

$s.run := s'.run \cup \{T\}$

* REQUEST_COMMIT(*T*, *r*) with *type(T) = split-write, split-key(T) = k*

$newvertex(T) = v'$ and $r = (r(1), r(2))$

Preconditions:

$T \in s'.create\text{-}requested - s'.run$

$s'.outdeg = maxsize$

r(1) = "split"

$r(2) = [s'.parent, splitkey, v'']$

where *splitkey* is a chosen key in $s'.keyset(T)$ and v'' is a new vertex; if v is the root, then $s'.parent$ is *nil*.

Effects:

1) new non-leaf vertex v'' is constructed with initial state of s_0:

s_0.*status* is the status of a vertex which contains the s'.*edges and* s'.*edgesets* with keys higher than k and includes the edge and edgeset for $[v'', v']$ if $k \geq splitkey$

s_0.*create–requested* and s_0.*run* inherit their values from the corresponding sets in s'.

if $k \geq splitkey$, then $s.run := s'.run \cup \{T\}$

2) for vertex v:

s.status is the status of s' after s_0.*edges* and s_0.*edgesets* are removed, and edge and edgeset for $[v, v']$ are included if $k < splitkey$.

s.create–requested and *s.run* are the values of s' after s_0.*create–requested* and s_0.*run* are removed.

if $k < splitkey$, then $s.run := s.run \cup \{T\}$

If vertex v is the root, then construct a new root v'' with initial status of a root vertex that has two child vertices v and v''.

Next we include the additional state components and transitions that are required for a *W-AM(X)* to handle the splitting process in the B^{link} tree algorithm.

Additional state components:

s.phase: can take additional value of "*split*"

s.split: a 3-tuple where *s.split* [1] and *s.split* [3] are vertices and *s.split* [2] is a key.

Additional transitions:

* REPORT_COMMIT(T', r), with *type(T')* = *search-write* or *split-write* and $r = (r(1), r(2))$ and $r(2)$ is a 3-tuple

Preconditions:

r(1) = "split"

Effects:

s.phase := "split"

s.split :=r(2)

* REQUEST_CREATE(T'), with $type(T) = split\text{-}write$, $split\text{-}key(T) = a$ and $newvertex(T') = b$

Preconditions:

$s'.phase$ = *"split"*

$T' \in accesses(s'.split[1])$

$a = s'.split[2]$

$b = s'.split[3]$

Effects:

$s.phase$:= *"split-in-process"*

* REPORT_COMMIT(T, r) with $type(T) = split\text{-}write$

Preconditions:

$r = nil$

$s'.phase$ = *"split-in-process"*

Effects:

$s.phase$:= *"finished"*

In the first REQUEST_COMMIT operation in the above, the preconditions test if an insertion is required at a leaf vertex which has reached the allowable maximum size (*maxsize*). If so, splitting is necessary. According to the B^{link} tree algorithm, v' is a new leaf with a link to the "right" neighbor of v. Vertex v must be "locked" while it is being linked to v'. This "locking" is implicitly enforced by the atomicity requirement on REQUEST_COMMIT by $VM(v)$, i.e., all its effects must be implemented indivisibly.

The parent vertex of v must be modified to include a new pointer to v'. For this purpose REQUEST_COMMIT(T,r) by $VM(v)$ returns $r(2)$ containing *s'.parent*, the *splitkey* and v'. However, the pointer updating is not urgent since the path via v to v' will serve as an indirect path for the time being. We give the responsibility for updating pointers to the root transaction T_0 as follows. The REPORT_COMMIT passes these three parameters returned in $r(2)$. T_0 intercepts

REPORT_COMMIT_AT($X.k$)OF(T) for the transaction T which caused splitting of v and issues a REQUEST-CREATE for transaction *IP-TM(X)* (*install-pointer* transaction manager) similar to the write access manager *W-AM(X)*. *IP-TM(X)* will take the parameters of the REPORT_COMMIT and search for the vertex in which to insert a pointer to the new vertex v'. *IP-TM(X)* will create child transactions which are accesses to vertex managers. However, instead of searching from the root vertex of X, the first child transaction accesses *s'.parent*, and the sibling transactions continue from there to search for the appropriate vertex to insert the new pointer. The appropriate vertex will be the one that contains an edgeset in which the *splitkey* lies. This searching is necessary because the parent vertex *s'.parent* may have been split and no longer be the correct vertex to insert the new pointer. This procedure is very similar to *search-write*, although the search is only horizontally moving right on the same level. It stops at the vertex where the pointer should be added to.

If level of the B-tree has been increased, there might be a need to add pointers to the newly installed levels near the root (a new root).

When the vertex u in which to insert the pointer to v' is found, the second REQUEST_COMMIT operation in the above inserts the pointer into the vertex.

If v is the root vertex, a new root has to be constructed which has v and v' as child vertices. The scheduler will be implicitly informed about the new root. I.e., when REQUEST_COMMIT(T,r) is input at the scheduler and T has constructed a new root, the effect at the scheduler is that *s.root_of_*(X) becomes the new root. Another REQUEST_COMMIT operation (not shown) is invoked when the insertion to the parent vertex requires the splitting of the parent vertex. It is analogous to the first REQUEST_COMMIT operation in in the above.

7.5.3.4.3. Vertex Merging

There is a background maintenance process similar to the *compression process* [Sag86], which reorganizes B^{link} trees when two adjacent vertices can be merged as a result of key deletions. We implement this process as transactions, called the **Merging Managers** (MM's), which are children of the root transaction. Informally, an MM searches through the whole tree, locking two adjacent vertices and their parent vertex at a time. If two adjacent vertices v and v' contain too few elements, then the contents of v' are appended to v, and the pointer to v' in the parent is removed. (v' is "*removed*" at this point.) The root should be merged with its child when the root vertex has only one child vertex left.

When a vertex is "removed", we cannot immediately "*discard*" it, because some transactions may have to read it. One solution is to record the time of its removal, and also record the starting time of each running transaction. A "removed" record can be discarded when all the transactions that started before the "removal" time have committed. We leave the job to a garbage collector.

Merging Manager MM(X)

We have for each B-link tree X a Merging Manager $MM(X)$. The root transaction periodically invokes the REQUEST_CREATE input to $MM(X)$. $MM(X)$ invokes accesses to the vertex managers of the vertices in X. These accesses include *merge-lock-parent, merge-find-size, merge-append, merge-remove, merge-update-parent,* and *merge-release,* which are discussed in the next section.

Input Operations:
 CREATE(T)
 REPORT_COMMIT(T', r), where $T' \in children$(T)
 REPORT_ABORT(T'), where $T' \in children$(T)
Output Operations:
 REQUEST_CREATE(T'), where $T' \in children$(T)
 REQUEST_COMMIT(T, r)

A state s of $MM(X)$ consists of the following components:

(1) s.created

(2) s.stage

(3) s.leftvertex, s.rightvertex and *s.parentvertex*

(*s.leftvertex* and *s.rightvertex* are two neighboring vertices in X and *parentvertex* is their parent)

(4) s.left, s.right and *s.parent*

(to record the status of what is being done on *leftvertex, rightvertex, and parentvertex*)

(5) s.outdeg (the total *outdeg* of *leftvertex* and *rightvertex*)

(6) s.count-release and *s.count-lock* (counters for locking and releasing locks)

(7) s.released (set of vertices whose locks have been released)

(8) s.append-elements (new elements to be appended to a vertex)

(9) s.new-neighbor (new neighbor to be attached to a vertex)

(10) s.ST (contains a set of transactions)

After the merging some of the contents of the *rightvertex* is migrated to *leftvertex*. There may still be transactions that are trying to access the *rightvertex*. For this reason, we do not discard the *rightvertex* immediately after merging, but leave the job to a garbage collector, which we assume runs periodically. The garbage collector will make sure that when it discards a vertex, no future transactions will access the vertex. We do not discuss the details.

The initial states s_0 of $MM(X)$ are those in which $s_0.created = false$ and the other components are empty. The transition relations of $MM(X)$ is given by all triples $<s', \pi, s>$ satisfying the following preconditions and effects, given separately for each π.

The merging process goes as follows: first a parent vertex and two adjacent children are locked through the transactions of types *merge-lock-parent* and *merge-find-size*. These also return the sizes of the child vertices. If the sizes are not too small, then *merge-release-lock* transactions are execution to release the locks and merging is started with another three vertices. Otherwise, *merge-remove* transaction is created to remove the right child, which at the same time will return the contents of the right child.

A *merge-append* transaction will then append such contents to the left child. At the same time, a *merge-update-parent* transaction can delete the pointer at the parent vertex to the right child. When all is done, a *merge-lock-release* transaction will release all locks and start merging with some other vertices.

*CREATE(T)

Preconditions:

s'.created = false

Effects:

s.created := true

s.count-lock := s.count-release := s.outdeg := 0

s.released := ∅

s.parentvertex, s.leftvertex and *s.rightvertex* are the first parent and children triple we come across when we traverse B-link tree X in breadth-first order

(*note that such a traversal requires some communications between $MM(X)$ and the scheduler and between the scheduler and the $VM(v')$ of some vertices v' in X. For simplicity, we omit the tedious and trivial description of such operations.*)

if such vertices exists, then *s.left := s.right := s.parent := "get-lock"*

if no such vertices exists, then *s.stage := "finished"*

*REQUEST_CREATE(T'), where *type*(T')=*merge-lock-parent* and *rightvertex*(T') = b

Preconditions:

s'.parent = "get-lock"

$T' \in$ *accesses(s'.parentvertex)*

b = *s'.rightvertex*

Effects:

s.parent := "locking-in-process"

*REQUEST_CREATE(*T'*), where *type* (*T'*)=*merge-find-size*

Preconditions:

T' ∈ *accesses(s'.leftvertex)*

s'.left = "get-lock"

Effects:

s.left := "locking-in-process"

*REQUEST_CREATE(*T'*), where *type* (*T'*)=*merge-find-size*

Preconditions:

T' ∈ *accesses(s'.rightvertex)*

s'.right = "get-lock"

Effects:

s.right := "locking-in-process"

*REQUEST_CREATE(*T'*), where *type* (*T'*)=*merge-remove* and *Trans-set*(*T'*) = TS

Preconditions:

s'.stage="merge-remove"

T' ∈ *accesses(s'.rightvertex)*

TS = s'.ST --*see* [1], [3]

(* TS is a set of transaction that may possibly access the removed vertex in the future,

[1] is where $s'.ST$ is updated (by return value r) , and [3] shows how r is acquired. *)

Effects:

s.stage := "remove-in-process"

*REQUEST_CREATE(*T '*), where *type* (*T '*)=*merge-append, append-edges*(*T '*)=*a* ,

append-edgesets(*T '*) = *b* and *new-neighbor*(*T '*)=*c*

Preconditions:

s'.stage="merge-append"

T' ∈ *accesses(s'.leftvertex)*

a = *s'.append-edges*

b = *s'.append-edgesets*

c = *s'.new-neighbor*

Effects:

s.stage := "append-in-process"

*REQUEST_CREATE(*T '*), where *type* (*T '*)=*merge-update-parent* and *merge-elements*(*T '*) = {*a* ,*b* }

Preconditions:

s'.stage = "merge-update-parent"

T' ∈ *accesses(s'.parentvertex)*

a = *s'.leftvertex*

b = *s'.rightvertex*

Effects:

s.stage := "update-parent-in-progress"

*REQUEST_CREATE(*T '*), where *type* (*T '*)=*merge-release-lock*

Preconditions:

s'.count-release > *0*

T' ∈ *accesses(s'.parentvertex or s'.leftvertex or s'.rightvertex)*

```
*REPORT_COMMIT(T ', r ), where type (T ')=merge-lock-parent

    Preconditions:

     s'.parent = "locking-in-process"

    Effects:

     s.parent := "locked"

     s.ST := r------------------------------------------------------------------------------------------------[1]

     s.count-lock := s'.count-lock + 1

     if s.count-lock = 3 ----------------------------------------------------------------------------------[2]

     then

         if s.size < minsize

         then s.count-release = 3    (* no merging *)

         else s.stage = "merge-remove"


*REPORT_COMMIT(T ', r ), where type (T ')=merge-find-size

    Preconditions:

     s'.left = "locking-in-process"

     T' ∈ accesses(s'.leftvertex)

    Effects:

     s.left := "locked"

     s.size := s'.size + r

     s.count-lock := s'.count-lock + 1

     if s.count-lock = 3

     then

         if s.size < minsize ... --------------------------------------------------------similar to [2]
```

```
*REPORT_COMMIT(T ', r ), where type (T ')=merge-find-size
    Preconditions:
     s'.right = "locking-in-process"
     T' ∈ accesses(s'.rightvertex)
    Effects:
     s.right := "locked"
     s.size := s'.size + r
     s.count-lock := s'.count-lock + 1
     if s.count-lock = 3
     then
         if s.size < minsize ... ----------------------------------------------------similar to [2]


*REPORT_COMMIT(T ', r ), where type (T ')=merge-release-lock
    Preconditions:
     T' ∈ accesses(v') and v' ∉ s'.released
    Effects:
     s.count-release := s'.count-release – 1
     s.released := s'.released ∪ {v'}
     if s.count-release = 0
       then
            s.leftvertex and s.rightvertex are assigned the next pair of sibling in the current level.
            (*The identities of these vertices can be obtained by accessing s.parentvertex and look for the
            next two outgoing edges to the right of s.rightvertex, we skip the details.*)
            If no such vertices exist then s.leftvertex and s.rightvertex are assigned the first pair of
            siblings in the next level that have a common parent, and s.parentvertex is the common
```

parent.

(*Again we skip the details of finding such vertices here.*)

If such vertices can be found, then *s.left* = *s.right* = *s.parent* =*"get-lock"*

If no such vertices can be found, then *s.stage* = *"finished"*

```
*REPORT_COMMIT($T'$, $r$), where $type(T')$= merge-remove and $r = (r(1), r(2), r(3))$
    Preconditions:
        $s'$.stage="remove-in-process"
    Effects:
        s.append-edges := r(1)
        s.append-edgesets := r(2)
        s.new-neighbor := r(3)
        s.stage := "merge-append"

*REPORT_COMMIT($T'$, $r$), where $type(T')$=merge-append
    Preconditions:
        $s'$.stage = "append-in-process"
    Effects:
        if $s'$.parent = "updated"
        then s.count-release := 3 and s.stage := "lock-release"
        else s.stage := "appended"

*REPORT_COMMIT($T'$, $r$), where $type(T')$=merge-update-parent
    Preconditions:
        $s'$.parent = "locked"
```

Effects:

if *s'.stage = "appended"*

then *s.count-release := 3* and *s.stage := "lock-release"*

else *s.parent := "updated"*

*REQUEST_COMMIT(*T*, *r*)

Preconditions:

s'.stage = "finished"

Effects:

s.stage := "commit-requested"

Augmentation to Vertex Manager *VM* (*v*)

When merging is performed concurrently with accesses, an access may need to read a "removed" vertex. This can happen to a read or write access invoked by a *R-AM* or *W-AM*, or an access by an *install-pointer* transaction. One solution is to backtrack to the parent of the removed vertex. Therefore, we need to augment the vertex managers to handle this backtracking.

The following are the additional accesses to vertex managers. The additional state variables are:
(1) s.merge-locked
(2) s.split-locked
(3) s.stage
(4) s.ST
(5) s.next-neighbor

Here we state an additional requirement for the splitting process: the parent vertex of the splitted vertex should be locked (*split-locked=true*) before splitting takes place. Merging always locks the parent and two child vertices before the merge of the child vertices (*merge-locked=true*) and split-locks conflict with merge-locks. This is to prevent concurrent deletion of the parent vertex by merging during splitting. The split-lock is released after the parent vertex is updated. Note that split-lock does not conflict with read-locks or write-locks. We omit the part of obtaining the split-lock. The following conditions of operations show how merging is done. To simplify the description, we shall omit the obvious precondition of REQUEST_COMMIT(T, r) that $T \in s'.create{-}requested - s'.run$ and the obvious effect that $s.run := s'.run \cup \{T\}$.

In the following, *merge-lock-parent* and *merge-find-size* transactions will lock a parent and 2 child vertices and also return the sizes of the child vertices. If the size is too small *merge-remove* will remove the right vertex, returning the contents of the vertex. Then *merge-append* will add these to the left vertex. *merge-update-parent* deletes the pointer from the parent to the right vertex and *merge-release* releases the locks on the relevant vertices.

*REQUEST_COMMIT(T, r), where *type* (T)=*merge-lock-parent* and *rightvertex*(T)=*b*

Preconditions:

s'.write-lockholders(k) $\cup$ *s'.read-lockholders(k)* $\subseteq$ *ancestors(T)*

r = set of all transactions $T' \in$ *s'.create-requested* and which were created at most t seconds ago

(* We assume that all transactions commit or abort within t seconds after being created *)

and *type* (T') = *search-read* or *search-write*

and *key*(T') $\in$ *s'.edgesets([v,b])* --[3]

Effects:

s.merge-locked := *true*

*REQUEST_COMMIT(T, r), where *type* (T)=*merge-find-size*

Preconditions:

s'.write-lockholders(k) $\cup$ *s'.read-lockholders(k)* $\subseteq$ *ancestors(T)*

s'.split-locked = *false*

$r = s'.outdeg$

Effects:

s.merge-locked := *true*

*REQUEST_COMMIT(T, r), where *type* (T)=*merge-remove*, *Trans-set*(T) = *ST*

and $r = (r(1), r(2), r(3))$

Preconditions:

$r(1) = s'.edges$

$r(2) = s'.edgesets$

$r(3) = s'.next\text{-}neighbor$

Effects:

s.stage := *"removed"*

s.ST := *ST*

(* *s.ST* holds the set of transactions that may access the removed vertex. The removed vertex is finally released when all the transactions in *s.ST* have informed the vertex about its commit (as follows). *)

*INFORM_COMMIT_AT(N)OF(T)

Preconditions:

s'.stage = *"removed"*

$T \in s'.ST$

Effects:

$s.ST := s'.ST - \{T\}$

if $s.ST \cap s.create\text{-}requested = \emptyset$ then *s.stage* := *"discarded"*

*REQUEST_COMMIT(T, r), where *type* (T)=*merge-append*, *append-edges*(T)=E,

append-edgesets(T)=ES and *new-neighbor*(T) = v'

Preconditions:

s'.merge-locked = true

Effects:

s.edges := s'.edges $\cup$ E $\cup$ *{v,v'} - { v, s'.next-neighbor }*

s.edgeset := s'.edgeset $\cup$ *ES*

s.edgeset(v') := s'.edgeset(s'.next-neighbor)

s.next-neighbor := v'

*REQUEST_COMMIT(T, r), where *type* (T)=*merge-update-parent*, *merge-vertices*(T)={ a ,b }

Preconditions:

s'.merge-locked = true

Effects:

s.edges := s'.edges - { [v,b] }

edgeset of [v,b] is deleted in *s.edgesets*

*REQUEST_COMMIT(T, r), where *type* (T)=*merge-release-lock*

Effects:

s.merge-locked := false

For the REQUEST_COMMIT(T, r) operations where the preconditions say that *write-lockholders* or *write-lockholders* $\cup$ *read-lockholders* must all be ancestors of T, an additional requirement is that the vertex not be *merge-locked.*

When a search-read or search-write accesses a removed vertex, it has to backtrack to the parent vertex through *s.parentvertex*.

*REQUEST_COMMIT(T, r), where *type* (T) = *search-write*

Preconditions:

s'.stage = "removed"

r(1) = "not-found" and *r(2) = s'.parent*

7.5.3.5. Scheduler

The B-tree locking (BTL) scheduler has the following operations:

Input Operations:
REQUEST_CREATE(T)
REQUEST_COMMIT(T,r)

Output Operations:
CREATE(T)
COMMIT(T), $T \neq T_0$
ABORT(T), $T \neq T_0$
REPORT_COMMIT(T), $T \neq T_0$
REPORT_ABORT(T), $T \neq T_0$
INFORM_COMMIT_AT($X.k$)OF(T), $T \neq T_0$
INFORM_ABORT_AT($X.k$)OF(T), $T \neq T_0$

These play roles analogous to those in the R/W locking scheduler [FLM87], except for the INFORM_COMMIT and INFORM_ABORT operations, which pass information about the fate of transactions to the vertex managers that currently hold the keys involved. In particular, if splitting takes place after an access T, so that the key that T accessed is now in a new vertex, INFORM_COMMIT or INFORM_ABORT messages must be directed to the correct vertex manager. Each vertex manager examines the AT($X.k$) field of the INFORM message and accepts a message if it currently holds key k. Also, the scheduler has the additional responsibility to keep track of the current root vertex of each B-tree in the system, as described above.

7.6. PROOFS OF STRONGLY-SERIAL CORRECTNESS

Serial correctness is defined in [FLM87, LyM86] as follows: a schedule α of a system is **serially correct for a transaction** T if its projection on T, $\alpha|T$, is identical to $\beta|T$ for some serial schedule β. α is **serially correct** if it is serially correct for every non-orphan, non-access transaction. (A transaction T is an **orphan** in a schedule α if the operation REPORT_ABORT(U) occurs in α for some ancestor U of T.) [FLM87] shows that any R/W locking schedule (i.e., schedule of a R/W locking system) is serially correct, which implies that any FFL schedule (i.e., schedule of a FFL system) is serially correct. Note that the above definition of serial correctness does not require the same serial schedule β satisfy $\alpha|T$ = $\beta|T$ for every non-access transaction T. We don't quite understand the intuitive significance of the above definition of correctness. Therefore, we shall adopt a stronger definition of correctness, which does require that β be the same for all T.

7.6.1. Strongly-Serially Correct Schedules

So far, we have considered two different systems, i.e., FFL system and BTL system. Notice that in the FFL system, each non-access transaction in the transaction tree is user-visible in the sense that all the subtransactions are invoked explicitly by the user within the user program. In the BTL system, however, the subtransactions which perform a search through a B^{link} tree, for example, is probably not explicitly invoked by the user. In his program, the user explicitly requests a read/write access to a key, $X.k$, say, but the system may provide a set of operations (subtransactions) for actually locating $X.k$. Therefore, this kind of (sub)transaction is **user-invisible**. Similarly, we call an object **user-visible** if it is explicitly specified in a user-visible transaction. One important condition that all user-invisible transactions must satisfy is that they should not have "side-effects", namely, they must not alter any user-visible object in such a way that user-visible transactions can observe the change.

We shall now address the question of how to define correctness for the schedules of a system (e.g., BTL system) with user-invisible transactions. We generalize the idea of an "extension" [GoL87] for our purpose. A system type $\Sigma' = (T', parent', O', R')$ is an **extension** of another system type $\Sigma = (T, parent, O, R)$ if the transaction tree $(T, parent)$ is a subtree of $(T', parent')$ rooted at its root such that each user transaction in T corresponds to a user transaction in T'. For each access transaction T with parent $parent(T)$ in T, there is a subtree in T' under the user transaction $parent(T)$. R is a subset of R'. T and O may not be subsets of T' and O', respectively. There is a natural mapping, i.e., the identity mapping, from Σ to Σ'. However, we find its inverse, $F_{\Sigma'\Sigma}$ more useful: $F_{\Sigma'\Sigma}(x) = null$ for any argument (i.e., transaction, object, return value, and *parent* of a transaction) x that is in Σ' but not in Σ. Our intention is to consider a BTL system as an extension of a FFL system and show that a FFL system can correctly be "mimiced" (or "simulated") by a BTL system. Then, intuitively, the correctness proof for a BTL system is "reduced" to that for a FFL system via a certain mapping.

Since we are interested in mapping systems, we define mappings for automata and their operations. Let A (B) be a system of system type $\Sigma' = (T', parent', O', R')$ $(\Sigma = (T, parent, O, R))$ such that all transactions and objects of B are user-visible and the subset T of T' consists of all user-visible transactions of A. We say that system A is an **extension** of system B if Σ' is an extension of Σ, and each non-access transaction of Σ is represented by the same automaton both in A and B. For system A which is an extension of B, we define the **standard mapping**, denoted $F_{A,B}$, from A to B. $F_{A,B}$ maps A's automata and their operations to those of B. Instead of giving a formal definition, we present an example to explain the standard mapping for the case where A = BTL system and B = FFL system.

In a BTL system, each object is not represented by a separate automaton, but instead an automaton represents a set of objects. However, in what follows, we pretend that there is an automaton for each object. Therefore, a vertex automaton plays the role of several automata, one for each key it contains, in addition to the role of the automaton for the vertex itself.

Example: Consider a FFL system whose transaction tree T is shown in Figure 7.6(b), where a node labeled "O" represents an access transaction. We have constructed this system, so that the BTL system with transaction tree T' of Figure 7.6(a) is an extension of it. For each non-access transaction (labeled U (user) or E (environment)) in T, there is a corresponding transaction in T'. The main idea is to make the interfaces shown by curved lines in Figures 7.6(a) and (b) the same as far as the user-visible transactions (U) are concerned. Therefore, we want to map a *R-AM* or *W-AM* T' in Figure 7.6(a) to the corresponding read or write access transaction T (labeled "O") in Figure 7.6(b), i.e., $F_{A,B}(T') = T$.

All operations of user transactions in Figure 7.6(a) are mapped to the corresponding operations in Figure 7.6(b). All operations of transaction T', which is a *R-AM* or *W-AM*, except for CREATE(T') and REQUEST_COMMIT(T'), are mapped to the *null* operation. CREATE(T') and REQUEST_COMMIT(T') correspond to those of an access transaction in Figure 7.6(b), i.e., they represent an interaction with an object. In particular, REQUEST_COMMIT of a *W-AM* updating a key $X.k$ is mapped to REQUEST_COMMIT of an access T which is a write access to $X.k$. We shall call both a *W-AM* and a write access **write transactions** on $X.k$.

As for the operations of an object, INFORM_COMMIT_AT($X.k$)_OF(T') and INFORM_ABORT_AT($X.k$)_OF(T') of an object $X.k$ in the BTL system are mapped to INFORM_COMMIT_AT($X.k$)_OF(T) and INFORM_ABORT_AT($X.k$)_OF(T), respectively, where an access transaction T is the image of a *R-AM* or *W-AM* T'. COMMIT and ABORT operations of the BTL system are mapped to those of the FFL system. The remaining operations of the BTL system which are not mapped so far are mapped to the *null* operation. □

In the rest of this paper, we consider only standard mappings. Consider a system A which is an extension of B. Let α be a schedule of A and β be a schedule of B with A and B starting in the same initial state. From now on, by "starting in the same initial state" we mean that each data object is in the same start state in both A and B, and T_0 starts in its start state. We assume that the transactions and objects of B are user-visible. A schedule α of A is said to $F_{A,B}$**-simulate** a schedule β of B (or α is an $F_{A,B}$**-simulation** of β) if

(1) β consists of the operations in $\{F_{A,B}(\pi) \mid \pi \text{ is in } \alpha\}$,

(2) for each user-visible, non-orphan, non-access transaction U in α,[35] $\beta \mid U = \bar{F}_{A,B}(\alpha) \mid U$, where $\bar{F}_{A,B}(\alpha)$ is obtained from α by deleting each operation π of α such that $F_{A,B}(\pi) = \mathit{null}$, and

(3) for each user-visible object X, α and β are **X-write-equal**, i.e., the sequences of REQUEST_COMMIT's of write transactions on X in α are the same as the corresponding REQUEST_COMMIT's of write transactions on X in β.

For a nested transaction system, we define the **committed projection** of a schedule α as the sub-schedule of α that consists of the operations of all non-orphan descendents of the committed children of the root transaction.

Definition: Let a system A be an extension of a *serial* system B. A schedule α of A is **strongly-serially correct** if there exists a schedule β of system B such that, starting at the same initial state, the committed projection of α $F_{A,B}$-simulates β. $\square$

If α is strongly-serially correct, then by condition (3) of $F_{A,B}$-simulation, the "effects" of α and serial schedule β on the user-visible objects are the same. Hence, the definition of strongly-serial correctness reflects the intuition that each user-visible T should get the same response and produce the same effects as in a serial schedule. Note that condition (3) is called "WW-constraint" [IKM87] in the conventional serializability theory. For more general purpose, condition (3) should not be included, i.e., conditions (1) and (2) only will be enough for strongly serial correctness. Such definition will then apply to quorum consensus in replicated database systems or cautious scheduling schemes that are not restricted by the WW-constraint.

Our proof of strongly-serial correctness of the FFL schedules would thus involve showing that, given any such α, there existed a schedule β of the serial system such that α F_{fs}-simulated β. In order to

[35] U is thus a user-visible transaction of B.

prove the strongly-serial correctness of schedules of the BTL system, we consider in Section 7.6.3 mapping F_{Bf} (B-tree to flat) from the BTL system to the FFL system such that any committed projection γ of the former system F_{Bf}-simulates a schedule α of the latter. The strongly-serial correctness of the committed projections of a BTL schedule is then proved by composing the above two mappings F_{Bf} and F_{fs}.

7.6.2. Strongly-Serial Correctness of FFL System

A formal proof of serial correctness of the R/W locking schedules (or FFL schedules) is given in [FLM87]. Here we outline a proof of the *strongly-serial correctness* of the committed projections of FFL schedules.

First, we describe the standard mapping F_{fs} that maps the FFL system to the serial system [FLM87]. For every non-access transaction T in the FFL system, $F_{fs}(T) = T$. For every access transaction T in the FFL system, $F_{fs}(T) = nil$. For every operation π of a non-access transaction of the FFL system, $F_{fs}(\pi) = \pi$, except that F_{fs}(INFORM_COMMIT_AT($X.k$)OF(T)) = *null*, for all $X.k$ and T, since there is no INFORM_COMMIT operation in the serial system.

We say that T **commits** in a schedule α when COMMIT(T) occurs in α. We say that T', a proper descendent of T, **commits to** T in a schedule α when COMMIT(T'') occurs in α, where T'' is a child of T and an ancester of T', provided that all the ancestors of T' that are descendents of T'' commit in α.

Theorem 7.1: A well-formed R/W locking (or FFL) schedule is strongly-serially correct.

Proof: Given any well-formed FFL schedule α', extract the committed projection α of α' containing all operations of the transactions committed to the root T_0. Applying the procedure in Figure 7.7 to the sequence CREATE(T_0)αCOMMIT(T_0), we construct a serial schedule CREATE(T_0)βCOMMIT(T_0) such that α F_{fs}-simulates β.

```
$\beta \leftarrow$ empty sequence
delete all INFORM_COMMIT_AT($X.k$)OF($T$) operations in $\alpha$;
commit-depth-first-traversal( $T_0$ ),

where
Procedure commit-depth-first-traversal($T$) is :
begin
  delete CREATE($T$) from $\alpha$ and append it to $\beta$;
  while $\alpha$ has one or more operations of a child transaction of $T$
  do
    find the first operation $\pi$ in $\alpha$ that is either REQUEST_CREATE($T'$)
      or REPORT_COMMIT($T'$, $r$) or COMMIT($T'$) of some child transaction $T'$ of $T$;
    if $\pi$ is COMMIT($T'$)
    then commit-depth-first-traversal($T'$)
    else delete $\pi$ from $\alpha$ and append it to $\beta$;
  endwhile;
  delete the REQUEST_COMMIT($T$, $r$) and COMMIT($T$) operation from $\alpha$ and append it to $\beta$;
end
```

Figure 7.7 Procedure Commit-Depth-First-Traversal

In the following, even though $F_{fs}(T) = T$ for all T, we sometimes use different names for T and $F_{fs}(T)$.

To show that β is a serial schedule, we need to show that β satisfies the following conditions (See the definition of a serial scheduler in [FLM87]):

(1) The preconditions of each output operation[36] of each non-access transaction are satisfied in β.

(2) CREATE(T) occurs in β after the siblings of T that have been created have all committed, i.e., if T' is a sibling of T and CREATE(T') precedes CREATE(T) in β, then COMMIT(T') should also precede CREATE(T).

[36] Note that there is no precondition for input operations since they are initiated by other automata.

(3) For each REQUEST_COMMIT(T_R, r) for a read access T_R in β, the return value r is written by the last write access T_W to the same object that commits prior to REQUEST_COMMIT(T_R, r) in β

The rest of the proof consists of a series of lemmas. □

Lemma 7.2: Condition (1) above is satisfied.

Proof: By the construction of β, it is clear that the order in α of CREATE, REQUEST_CREATE's, REPORT_COMMIT's and REQUEST_COMMIT for each non-access transaction is preserved in β. This implies that the preconditions of each output operation, REQUEST_CREATE or REQUEST_COMMIT, of the non-access transactions are satisfied in β since they are satisfied in α. □

Lemma 7.3: Condition (2) above is satisfied.

Proof: By the construction of β, all operations of all descendents of a transaction T are clustered, so that they all appear after CREATE(T) and before COMMIT(T), with no other operations in between. Therefore COMMIT(T) always follows CREATE(T) with no CREATE(T') of a sibling T' of T in between. Hence, condition (2) follows. □

Lemma 7.4: Condition (3) above is satisfied.

We shall make use of Lemma 7.5 and Corollary 7.6 to prove Lemma 7.4. Recall the definition of "commits to" given earlier in this section.

Lemma 7.5: If T_a commits to T_b before T_c commits to T_d in α, and T_b is an ancestor of T_d, then T_a commits before T_c in β

Proof: By the construction of β, the child T_A of T_b that is an ancestor of T_a commits in β before the child T_B of T_b which is an ancestor of T_c, is created. Since the CREATE's of the transactions in β follow the depth-first order and COMMIT(T) always follows CREATE(T), T_a commits before T_c commits in β. □

Corollary 7.6: If T_a commits to T_b before T_c commits to T_b in α, then T_a commits before T_c in β.

Proof: Follows from Lemma 7.5 by letting $T_b = T_d$. □

Proof of Lemma 7.4: In β, each read access T_R to a data object $X.k$ returns the same value r as read by the corresponding read access in α. Because of the precondition of REQUEST_COMMIT(T, r) of a read access T, which says that all holders of writelocks on the data object must be ancestors (in the transaction tree) of T, the write access T_W that writes r in α is the last one that commits to the common ancestor T'' of T_R and T_W before the REQUEST_COMMIT of T_R in α. After T', the ancestor of T_W which is a child of T'', has committed, INFORM_COMMIT_AT($X.k$)OF(T') is issued by which the writelock held by T_W and the value written by T_W are passed to T''. T'' then becomes the *least(writelock-holder)* of the data object for T_R. Hence the value read by T_R is the one written by T_W.

In order for β to be a serial schedule, r should be the value written by the write transaction T_W' which is the last write on $X.k$ that commits before REQUEST_COMMIT(T_R, r) in β. It is sufficient to show that $T_W' = T_W$. We shall prove by contradiction that $F_{fs}(T_W) = T_W'$. Assume that $T_W' \neq T_W$. Let T_1 be the least common ancestor of T_W and T_R and let T_2 be the least common ancestor of T_W' and T_R. Since T_1 and T_2 are both ancestors of T_R, either T_1 is an ancestor of T_2 or vice versa.

First consider the case where T_2 is an ancestor of T_1. There are two possible subcases for α. First (Figure 7.8(a), T_W' commits to T_2 before T_W commits to T_1 and T_W commits to T_1 before T_R commits to T_1. Second, (Figure 7.8(b) in α, T_W commits to T_1 before T_R commits and T_R commits before T_W' commits to T_2. From Lemma 7.5, T_W commits before T_R in β in both cases. Because the commit order in β follows depth-first order of the transaction tree, in both cases, T_W' commits either before or after both T_W and T_R commit. Therefore, it is not possible for T_W' to be the last write on $X.k$ that commits before REQUEST_COMMIT(T_R, r) in β.

Next consider the case where T_1 is an ancestor of T_2. In α, there are two possible subcases. First, (Figure 7.8(c)) T_W' commits to T_2 before T_W commits to T_1, and T_W commits to T_1 before T_R commits.

However, this is not possible because at the time when T_W' has committed to T_2 and T_R has not committed, the write lock on X is held by a descendent of T_2, which is not an ancestor of T_1. Therefore, the precondition of REQUEST_COMMIT(T_W, r) is not satisfied and the T_W cannot commit at this time.

Second, (Figure 7.8(d)) T_W commits to T_1 before T_R commits, and T_R commits before T_W' commits to T_2. Since the precondition of REQUEST_COMMIT(T_W', r) requires that all readlock-holders be ancestors of T_W', T_R must commit to T_2 before T_W' commits. This means that T_W' commits to T_2 after T_R commits to T_2. By Corollary 7.6, T_R commits before T_W' in β. Therefore, it is not possible for T_W' to be the last write on X that commits before REQUEST_COMMIT(T_R, r). This completes our proof by contradiction. Hence we conclude that $T_W = T_W'$. □

The proof of Theorem 7.1 will be complete by proving the following:

Lemma 7.7: Let β be obtained from α by the procedure in Figure 7.7. Then α F_{fs}-simulates β.

We shall apply the following lemma to prove Lemma 7.7.

Lemma 7.8: Let T ($\neq T_a$, $\neq T_b$) be the least common ancestor of T_a and T_b. If T_a commits before T_b in β, then in α, T_a commits to T before T_b commits to T.

Proof: By the construction of β, the child T_1 of T which is an ancestor of T_a must commit before child T_2 of T which is an ancestor of T_b is created. Since the order of commits among siblings is preserved, T_2 commits after T_1 in α, and hence the lemma follows. □

Proof of Lemma 7.7: By construction of β, β consists of operations $F_{fs}(\pi)$ for all π in α, and its projection on any user-visible transaction T, $\beta|T$, is equal to $\alpha|T$. We now show that the order of COMMIT's for two write accesses to the same object in α are preserved in β. Let us assume the contrary. Let T_a and T_b be two write accesses which write the same object such that T_a commits to T_1 before T_b commits to T_2 in α, and T_b commits before T_a in β. Let T'' be the least common ancestor of T_1 and T_2. In order for T_b to commit, the preconditions require that the writelock held by T_a be passed to T''. Therefore, in α, T_a commits to T'' before T_b commits to T''. However, since T_b commits before T_a in β, by Lemma 7.8, T_b must commit to T'' before T_a commits to T'' in α, a contradiction. Since, for the same object, the order of COMMIT's of write accesses is the same as the order of REQUEST_COMMIT's of the same write accesses, we have proved that for the REQUEST_COMMIT's of the write accesses to the same object, their order in α is preserved in β. □

7.6.3. STRONGLY-SERIAL CORRECTNESS OF BTL SYSTEM

In this section, we consider the BTL system with transaction tree Y and a set of B^{link} trees. The non-access transactions in the FFL system are user-visible. Let K be the set of keys of the B^{link} trees in a BTL system. Clearly, K is the set of user-visible objects. We now construct a FFL system such that the BTL system is an extension of the FFL system. Let F_{Bf} be the standard mapping from the BTL system to the FFL system. Thus, F_{Bf} maps the B^{link} trees in the BTL system to their identically-named flat files in the FFL system. We shall refer to flat file $F_{Bf}(X)$ as X for each B^{link} tree X. Also, $F_{Bf}(T) = T$ if T is a non-access (user visible) transaction in the FFL system.

For each transaction T of type $R\text{-}AM(X)$, $F_{Bf}(T)$ is a read access in $M(X.k)$ to flat file object $X.k$ with $k = key(T)$. For each transaction T of type $W\text{-}AM(X)$, $F_{Bf}(T)$ is a write access in $M(X.k)$ to flat file object X with $k = key(T)$ and which writes the value of $data(T)$ into $X.k$. For an access (user invisible) transaction T in $VM(v)$, $F_{Bf}(T)$ is the *null* transaction, which does nothing.

For each operation π = CREATE(T) (REQUEST_COMMIT(T, r)) of a transaction T of type *R-AM* or *W-AM* $F_{Bf}(\pi)$ is the CREATE(O) (REQUEST_COMMIT(O, r)) of O where $O = F_{Bf}(T)$. Other operations of T are mapped to *null*.

For each operation π = REQUEST_CREATE(T') or REPORT_COMMIT(T', r) or REPORT_ABORT(T', r), where T' is a child of a transaction T of type *R-AM* or *W-AM*, $F_{Bf}(\pi)$ is the *null* operation, which does not change the state of the system.

For each operation π = INFORM_COMMIT_AT($X.k$)OF(T) of $VM(v)$ for v in B^{link} tree X, if $F_{Bf}(T)$ is not *null*, then $F_{Bf}(\pi)$ is the operation INFORM_COMMIT_AT($X.k$)OF($F_{Bf}(T)$), which is an operation of $M(X.k)$; else $F_{Bf}(\pi)$ is *null*. For each operation π = INFORM_ABORT_AT($X.k$)OF(T) of $VM(v)$ for v in B^{link} tree X, if $F_{Bf}(T)$ is not *null*, then $F_{Bf}(\pi)$ is the operation INFORM_ABORT_AT($X.k$)OF(FBf(T)), which is an operation of $M(X.k)$; else $F_{Bf}(\pi)$ is the null operation. For any other operation π of $VM(v)$, $F_{Bf}(\pi)$ is *null*. ☐

Theorem 7.9: A well-formed BTL schedule is strongly-serially correct.

Proof: To show the strongly-serial correctness of the BTL schedules, we shall find, for each schedule γ' of the system, a committed projection α in the FFL system such that the committed projection γ of γ' F_{Bf}-simulates α. By transitivity of the "F-simulates" relation, there exists a schedule with committed projection β in the serial system such that γ F_{Bs}-simulates β, where F_{Bs} is $F_{Bf} \cdot F_{fs}$, i.e., the composition of F_{Bf} and F_{fs}.

By the above definition of mapping F_{Bf}, the only difference between schedules γ and α is as follows: for each read (write) access $F_{Bf}(\pi)$ in α, π is the corresponding *R-AM* (*W-AM*) transaction which has sub-transactions that search through some vertices in the B^{link} tree and finally perform read (write); there may be additional transactions for vertex splitting and merging in α.

Therefore, if we can show that, in spite of concurrent splitting and merging, searching can lead to a successful read or write and that the read operation reads the value as seen by the least ancestor of the read/write access that holds the writelock on the data object, then γ F_{Bf}-simulates the committed projection α.

From the transition relation for search-read, the return value of a read operation is the value *s.map(least(writelock-holder))* which is the value as seen by the least ancestor of the read transaction that holds the writelock on the data object [LyM86]. It remains to show that searching always leads to the vertex that contains the data object. In the following, we first list out the conditions necessary for successful searching and then informally show that the BTL system satisfies the conditions.

In [GoS85], Goodman and Shasha define the "good states" of a search structure. We shall rephrase their definition for B^{link} tree as follows. A **good state** for a B^{link} tree satisfies the following condition:

(GS) each key is contained in exactly one leaf in the B^{link} tree and is reachable from the root.

The necessary conditions guaranteeing successful searching are as follows.

(1) Each search structure begins in a good state and each operation (e.g., search-read, search-write, split) on the search structure maps a good state to a good state.

(2) No operation π reduces the set of keys reachable (through links or backtracking) from a vertex that is not discarded.

(3) No transaction accesses a discarded vertex.

Condition (1) implies that when the search for a key starts from the root, and no other operations interfere with the search, then it always leads to the correct vertex.

Condition (2) enables the continuation of search after being interrupted by other operations like splitting without having to backtrack the search. The additional links between vertices of the B^{link} tree on the same "level" ensures that this condition is satisfied during splitting.

Conditions (1) and (2) are similar to those stated in [GoS85] and [ShG88] for the general link technique. Condition (3) and the part on backtracking in Condition (2) are additional requirements when concurrent merging operations are considered. Merging of two vertices removes one node and copies the contents of the removed vertex to the other vertex. The parent vertex is then updated accordingly. If merging is an atomic action, then it maps a good state to a good state. Conditions (2) and (3) ensure that when search for a key is interrupted by merging, the search will still succeed in finding the key.

To show that Condition (1) is satisfied, we note that the initial state of each B^{link} tree must be a good state by definition, and the only operations that affect the locations of keys are splitting and merging. A splitting operation distributes some keys in a vertex v to a new vertex which is reachable from v through a link. The merging operation moves all the keys of one vertex v to a sibling vertex v', and all those keys are reachable from the parent of v and v'.

Condition (2) is satisfied because no operation reduces the set of keys reachable through links from an un-discarded vertex v. A splitting process only reduces the keyset of a vertex v, but the keys removed from v are still reachable from v through a link.

Condition (3) is satisfied because the only operation which may remove a vertex is a merging operation. A merging operation copies all keys from a vertex v and appends them to a sibling vertex. Vertex v is kept until all transactions which might access v have committed. A write access to v after it is removed backtracks to the parent vertex of v, so that it can find the new location of the key it wants to write. □

So far, we have ignored transactions of type *install-pointer* and Merging Managers in the BTL system by mapping them to *null*. We need a more formal proof that they indeed perform their functions.

7.7. CONCLUSION

We have designed a concurrency control algorithm for a database system with B^{link} trees as search structures accessed by nested transactions. We applied the idea of resilient 2-phase locking [Mos85] for the resolution of conflicts between transactions and the B^{link} tree technique in [Sag86] to handle accesses to B-trees. The I/O automaton model developed by an MIT group was used in the specification and proofs of correctness of the system.

We have adopted the "strongly-serial" correctness as our correctness criterion, which, we believe, better captures our notion of correctness than "serially correct." We proved the strongly-serial correctness of both the FFL and BTL system.

There are tedious details in our work, but they are probably inevitable in any formal proofs of correctness for a complex system. We feel that automata are still more "comprehensible" than logic. In any case, the automaton model has forced us not to resort to hand waving.

We believe our proof approach making use of a mapping from the target system to a system that has been proven to be correct is useful and can be applied to other semantically-based concurrency control schemes. A similar approach has been used in [GoL87].

We addressed the issue of safety of our system but not that of liveness. Moss [Mos85] proposes a deadlock detection scheme for his nested transaction system, which should be applicable to our system. However, as pointed out in [LyM86], it is not easy to formalize the notion of liveness, which is the first step to be taken if we want to prove the correctness of the deadlock detection scheme using the I/O automaton model. It is left as an open problem for future research.

For future research, we also propose to eliminate the scheduler in the system. Right now every output (input) operation from a transaction automaton activates (is activated by) an input (output) operation of the scheduler automaton. Having this critical point in the scheduler automaton is in conflict with the principle of autonomy if we were to model a distributed system. Hence it would enhance the modelling power if we could distribute the control of the scheduler, in particular, the part that keeps track of the statuses of create, commit or abort operations of different transactions in the system.

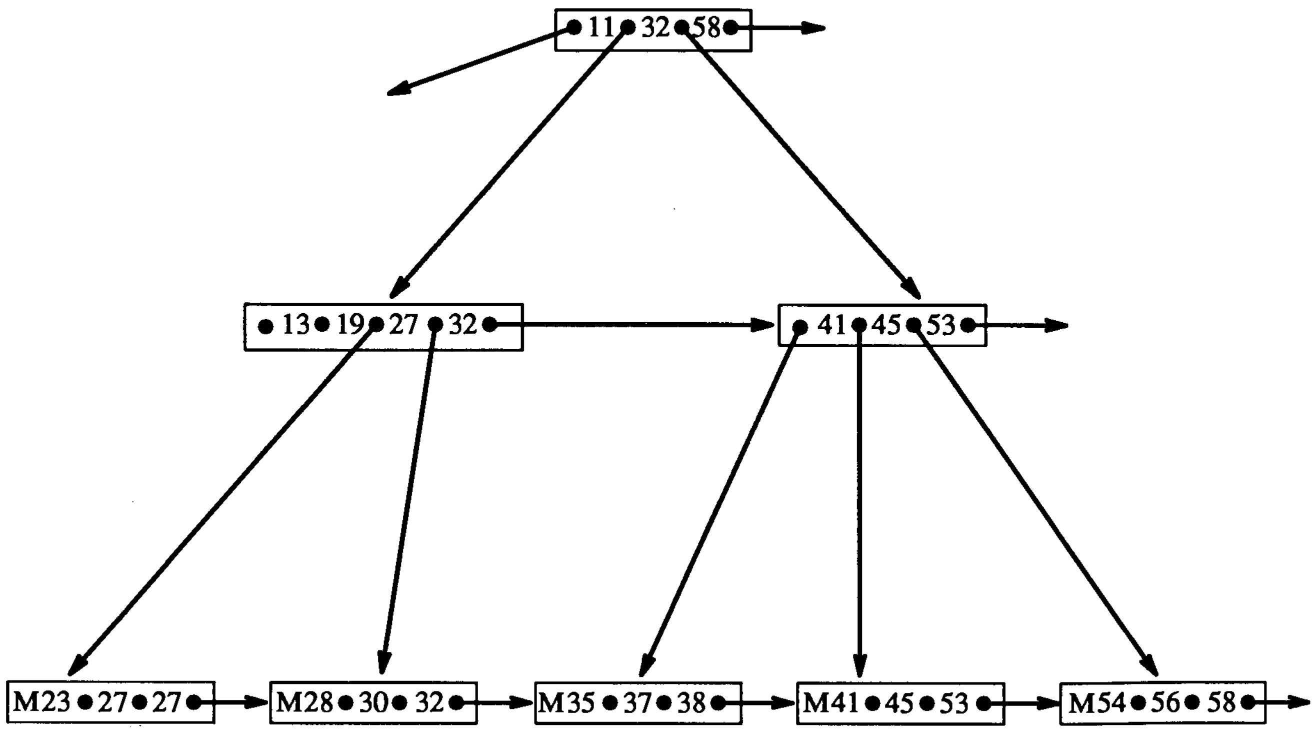


Figure 7.1 Part of a B^{link} tree

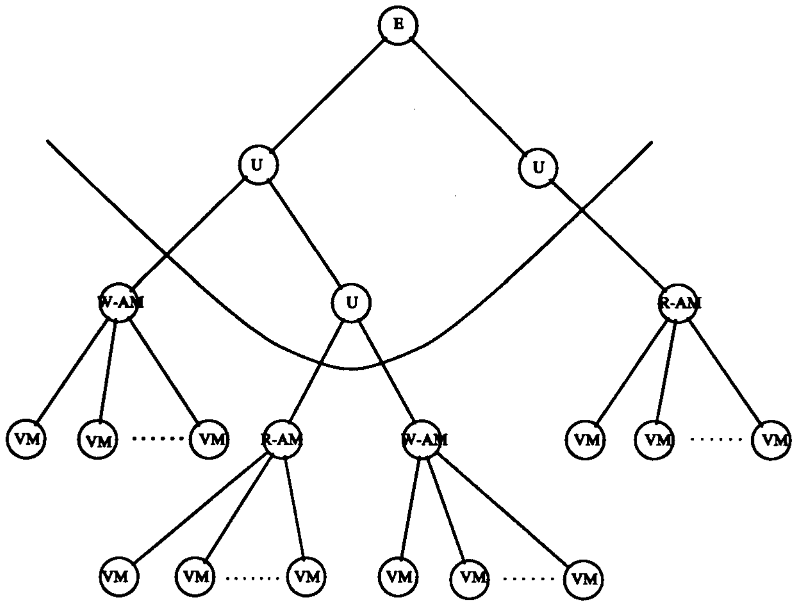


Figure 7.6 (a) Transaction tree of a BTL system

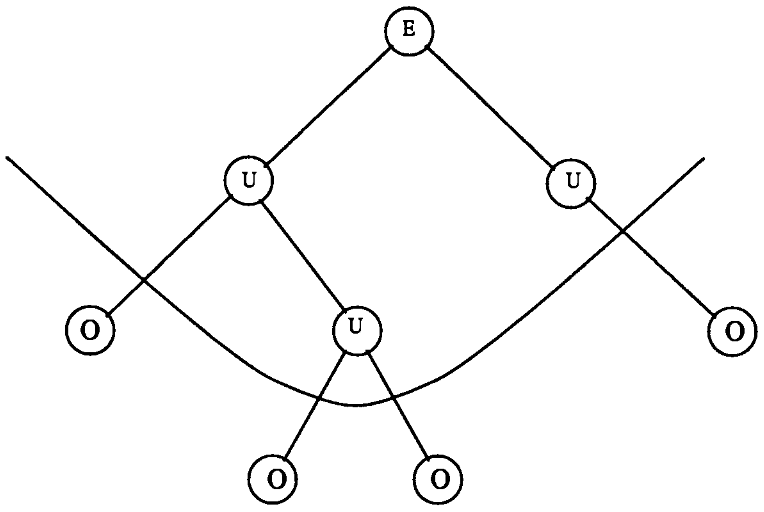


Figure 7.6 (b) Transaction tree of a FFL system

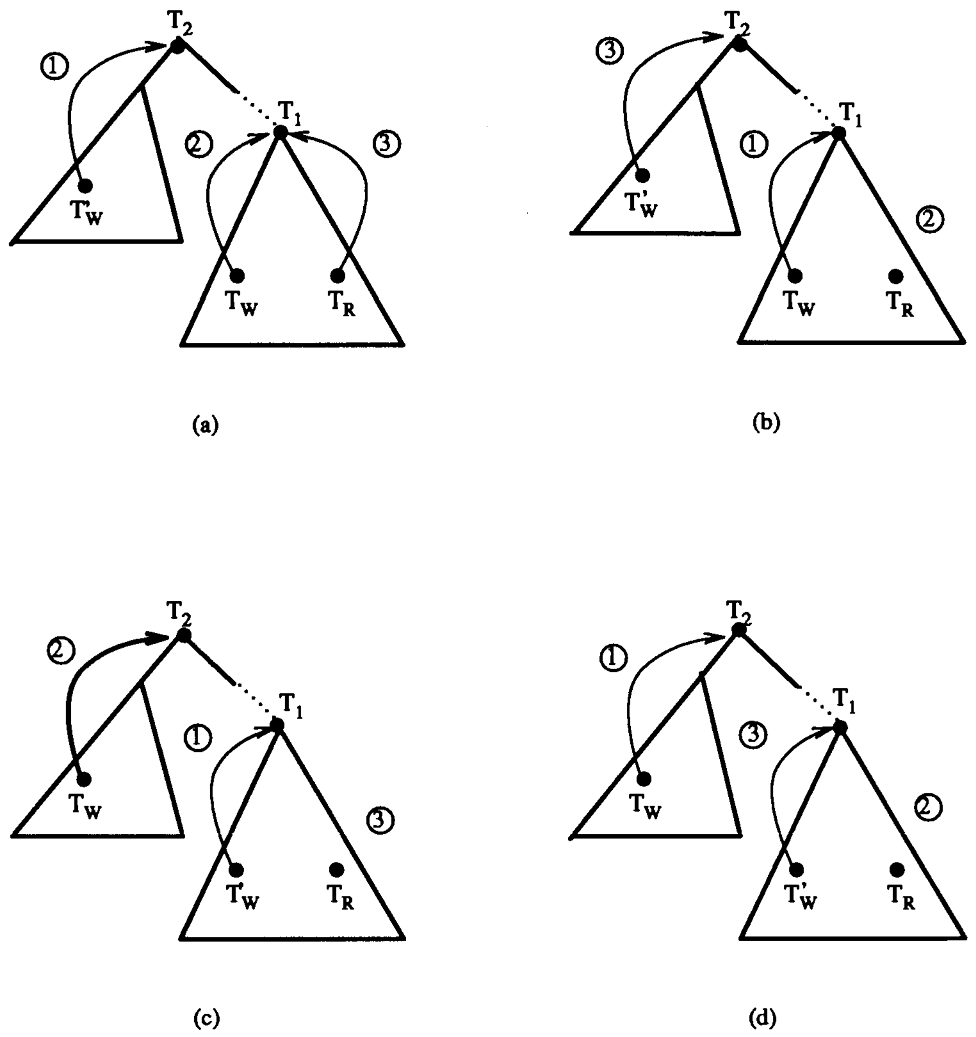


Figure 7.8 Cases in the proof of Lemma 7.5

CHAPTER 8

CONCLUSIONS

In this thesis, we studied the concurrency and availability of replicated distributed database systems. As the performance of any protocol designed for such systems very much depends on the characteristics of the system behaviour, there is unlikely to be a unique protocol which will meet the requirement or function well for all different applications. We looked into two different protocols, the Generalized Virtual Partition Protocol (GVP), and the Transaction Replication System (TRS), which are quite different in nature, and which will be suitable for different applications.

Given a complex system like GVP or TRS, it is important to give a rigorous proof of correctness. Therefore, the second major problem we attacked is the formal proof of correctness for complex protocols. We studied the formal modeling of database protocols, in the hope that it will lead to a satisfactory proof of correctness for the previously proposed protocols. Due to the complexity of the problem, we have not been completely successful in this effort. However, we have extended the work of some researchers at MIT, who make use of the I/O automaton model in the modeling and correctness proofs of database systems. In particular, we have tried the approach on concurrency control of nested transactions accessing B-trees. The notions of user invisible data and transactions are exhibited by the B-tree structures and tree searching. Based on these notions, we revised the definition of serial-correctness for nested transactions, which we call "strongly serial correctness". Our work also demonstrated the powerful concept of mapping from one system to another system, usually of less complexity, with the effect of simplifying the resulting proof.

REFERENCES

[AlS68] A. A. Albert and R. Sandler, *An Introduction to Finite Projective Planes*, Holt, Rinehart, and Winston, New York, 1968.

[AlD76] P. A. Alsberg and J. D. Day, A Principle for Resilient Sharing of Distributed Resources, *Proc. 2nd International Conference on Software Engineering*, , Oct 1976, 562-570.

[BeG81] P. A. Bernstein and N. Goodman, Concurrency Control in Distributed Database Systems, *ACM Computing Surveys 13*, 2 (June 1981), 185-221.

[BeG83] P. A. Bernstein and N. Goodman, The Failure and Recovery Problem for Replicated Databases, *Proc. 2nd ACM Symp. on Principles of Distributed Computing*, Montreal, Aug. 1983, 114-122.

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

[Che88] D. Cheung, A Study of the Availability and Serializability in a distributed database system, Ph.D. Thesis, School of Computing Science, Simon Fraser University, Burnaby, British Columbia, Jan 1988.

[ClS80] D. D. Clark and L. Svobodova, Design of Distributed Systems Supporting Local Autonomy, *Proc. COMPCON-IEEE*, , Spring 1980, 438-444.

[COK86] B. A. Coan, B. M. Oki and E. K. Kolodner, Limitations on Database Availability When Networks Partition, *Proc. 5th ACM Symp. on Principles of Distributed Computing*, Calgary, Alberta, Aug 1986, 187-194.

[DGS85] S. B. Davidson, H. Garcia-Molina and D. Skeen, Consistency in Partitioned Networks, *ACM Computing Surveys 17*, 3 (Sept 1985), 341-370.

[DoR82] D. Dolev and R. Reischuk, Bounds on Information Exchange for Byzantine Agreement, *Proc. 1st ACM Symp. on Principles of Distributed Computing*, Los Angeles, 1982, 132-140.

[DHS84] D. Dolev, J. Halpern and R. Strong, Fault-tolerant Clock Synchronization, *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, Vancouver, Aug 1984, 89-102.

[EaS83] D. Eager and K. C. Sevcik, Achieving Robustness in Distributed Database Systems, *ACM Trans. Database Systems 8*, 3 (Sept 1983), 354-381.

[ESC85] A. El-Abbadi, D. Skeen and F. Cristian, An Efficient, Fault-Tolerant Protocol for Replicated Data Management, *Proc. 4th ACM Symp. on Principles of Database Systems*, Portland, Mar 1985, 215-229.

[ElT86] A. El-Abbadi and S. Toueg, Availability in Partitioned Replicated Databases, *Proc. 5th ACM Symp. on Principles of Database Systems*, Cambridge, Mass., Mar 1986, 240-251.

[ElT89] A. El-Abbadi and S. Toueg, Maintaining Availability in Partitioned Replicated Databases, *ACM Trans. Database Systems 14*, 2 (June 1989), 264-290.

[Ell87] C. S. Ellis, Concurrency in Linear Hashing, *ACM Trans. Database Systems 12*, 2 (June 1987), 195-217.

[EGL76] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, The Notions of Consistency and Predicate Locks in a Database System, *Comm. ACM 19*, 11 (Nov. 1976), 624-633.

[FLM87] A. Fekete, N. Lynch, M. Merritt and W. Weihl, Nested Transaction and Read/Write Locking, *Proc. 6th ACM Symp. on Principles of Database Systems*, San Diego, 1987, 97-111.

[FLM88] A. Fekete, N. Lynch, M. Merritt and W. Weihl, Commutativity-Based Locking for Nested Transactions, *Book draft*, , Aug. 1988.

[FLP85] M. Fischer, N. Lynch and M. Paterson, Impossibility of Distributed Consensus with One Faulty Process, *J. ACM 32*, 2 (1985), 374-382.

[Fu89] A. Fu and T. Kameda, Concurrency Control of Nested Transactions Accessing B-trees, *Proc. 8th ACM Symp. on Principles of Database Systems*, Philadelphia, March 1989.

[Gar82] H. Garcia-Molina, Elections in a Distributed Computing System, *IEEE Trans. Comput. C-31*, 1 (Jan. 1982), 48-59.

[GaB85] H. Garcia-Molina and D. Barbara, How to Assign Votes in a Distributed System, *J. ACM 32*, 4 (Oct 1985), 841-860.

[GPD86] H. Garcia-Molina, F. Pittelli and S. Davidson, Applications of Byzantine Agreement in Database Systems, *ACM Trans. Database Systems 11*, 1 (March 1986), 27-47.

[GaK88] H. Garcia-Molina and B. Kogan, Node Autonomy in Distributed System, *Proc. Int'l. Symp. on Database in Parallel and Distributed Systems*, Austin, 1988, 158-166.

[GaK85] D. Gawlick and D. Kinkade, Varieties of Concurrency Control in IMS/VS Fast Path, Technical Report, Tandem Computers, Cupertino, CA, 1985.

[Gif79] D. K. Gifford, Weighted Voting for Replicated Data, *Proc. 7th ACM Symp. on Operating System Principles*, Pacific Grove, CA, Dec 1979, 150-162.

[GoL87] K. J. Goldman and N. A. Lynch, Quorum Consensus in Nested Transaction Systems, *Proc. 6th ACM Symp. on Principles of Distributed Computing*, Vancouver, Aug 1987, 27-41.

[GoS85] N. Goodman and D. Shasha, Semantically-based Concurrency Control for Search Structures, *Proc. 4th ACM Symp. on Principles of Database Systems*, Portland, 1985, 8-19.

[GLP75a] J. Gray, R. A. Lorie and G. R. Putzolu, Granularity of Locks in a Shared Data Base, *Proc. 1st Int'l Conf. on Very Large Data Bases*, Framingham, MA, Sept 1975, 428-451 .

[GLP75b] J. Gray, R. A. Lorie, G. R. Putzulo and I. L. Traiger, Granularity of Locks and Degrees of Consistency in a Shared Database, Research Report, IBM, Sept 1975.

[Gra79] J. N. Gray, Notes on Data Base Operating Systems, in *Operating Systems: An Advanced Course*, R. Bayer, R. M. Graham and G. Seegmuller (ed.), Springer-Verlag, Berlin and New York, 1979, 393-481.

[Gra88] J. Gray, The Cost of Messages, *Proc. ACM Symp. on Principles of Distributed Computing*, Toronto, August 1988, 1-7.

[HaH88] T. Hadzilacos and V. Hadzilacos, Transaction Synchronization in Object Bases, *Proc. 7th ACM Symp. on Principles of Database Systems*, Austin, Texas, March 1988.

[Har65] M. A. Harrison, *Introduction to Switching and Automata Theory*, McGraw-Hill, New York, 1965.

[Her86] M. P. Herlihy, Optimistic Concurrency Control for Abstract Data Types, *Proc. 5th ACM Symp. on Principles of Distributed Computing*, Calgary, Alberta, 1986, 206-217.

[Her87] M. Herlihy, Dynamic Quorum Adjustment for Partitioned Data, *ACM Trans. Database Systems 12*, 2 (June 1987), 170-194.

[HeW88] M. P. Herlihy and W. E. Weihl, Hybrid Concurrency Control for Abstract Data Types, *Proc. 7th ACM Symp. on Principles of Database Systems*, Austin, Texas, March 1988, 201-210.

[IKM87] T. Ibaraki, T. Kameda and T. Minoura, Serializability with Constraints, *ACM Trans. Database Systems 12*, 3 (Sept 1987), 429-452.

[JoS90] T. Johnson and D. Shasha, A Framework for the Performance Analysis of Concurrent B-Tree Algorithms, *Proc. 9th ACM Symp. on Principles of Database Systems*, Nashville, Tenn, Apr. 1990, 273-287.

[KoG87] B. Kogan and H. Garcia-Molina, Update Propagation in Bakunin Data Networks, *Proc. 6th ACM Symp. on Principles of Distributed Computing*, Vancouver, Aug 1987, 13-26.

[Koh78] Z. Kohavi, *Switching and Finite Automata Theory*, McGraw Hill, New York, 1978.

[KuL80] H. T. Kung and P. L. Lehman, Concurrent Manipulation of Binary Search Trees, *ACM Trans. Database Systems 5*, 3 (Sept 1980), .

[Lam78a] L. Lamport, The Implementation of Reliable Distributed Multiprocess Systems, *Comput. Networks* 2, (1978), 95-114.

[Lam78b] L. Lamport, Time, Clocks and the Ordering of Events in a Distributed Multiprocess Systems, *Comm. ACM 21*, (July 1978), 558-564.

[LSP82] L. Lamport, R. Shostak and M. Pease, The Byzantine Generals Problem, *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 382-401.

[LaM84] L. Lamport and P. M. Melliar-Smith, Byzantine Clock Synchronization, *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, Vancouver, Aug 1984, 68-74.

[LaS76] B. Lampson and H. Sturgis, Crash Recovery in a Distributed Storage System, Tech. Report, CS Lab, Xerox Parc, Palo Alto, California, 1976.

[LeY81] P. L. Lehman and S. B. Yao, Efficient Locking for Concurrent Operations on B-Trees, *ACM Trans. on Database Systems 6*, 4 (Dec 1981), 650-670.

[LiS80] B. Lindsay and P. G. Selinger, Site Autonomy Issues in R*: a Distributed Database Management System, Research Report, IBM Research Division, 1980.

[Lov73] L. Lovasz, Coverings and Colorings of Hypergraphs, *Proc. 14th Southeastern Conference on Combinatorics, Graph Theory and Computing*, Utilitas Mathematica, Winnipeg, Canada, 1973, 3-12.

[LuL84] J. Lundelius and N. Lynch, A new Fault Tolerant Algorithm for Clock Synchronization, *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, Waterloo, Ontario, Aug 1984, 75-88.

[LyM86] N. Lynch and M. Merritt, Introduction to the Theory of Nested Transactions, *Technical Report, MIT*, , June 1986.

[Mae85] M. Maekawa, A sqrt(N) Algorithm for Mutual Exclusion in Decentralized Systems, *ACM Trans. Computer Systems 3*, 2 (May 1985), 145-159.

[Mos81] J. E. B. Moss, Nested Transactions: An Approach to Reliable Distributed Computing, Ph.D. Thesis , Technical Report MIT/LCS/Tech. Rep.-260, 1981.

[Mos85] J. E. B. Moss, Nested Transactions: An Approach to Reliable Distributed Computing, The MIT Press Series in Information Systems, 1985.

[NeT88] G. Neiger and S. Toueg, Automatically Increasing the Fault-Tolerance of Distributed Systems (Preliminary Version), *Proc. 7th ACM Symp. on Principles of Database Systems*, Austin, Texas, 1988, 248-262.

[Pap79] C. H. Papadimitriou, The Serializability of Concurrent Database Updates, *J. ACM 26*, 4 (Oct. 1979), 631-653.

[PSL80] M. Pease, R. Shostak and L. Lamport, Reaching Agreement in the Presence of Faults, *J. ACM 27* , 2 (April 1980), 228-234.

[PeR88] P. Peinl and A. Reuter, High Contention in a Stock Trading Database: A Case Study, *ACM Sigmod International Conference on Management of Data*, , June 1988, 260-268.

[PiG87] F. Pittelli and H. Garcia-Molina, Recovery in A Triple Modular Redundant Database System, *Proc. 7th Int. Conf. on Distributed Comp. Sys.*, Berlin, Sept 1987, 514-520.

[PiG89] F. Pittelli and H. Garcia-Molina, Reliable Scheduling in a TMR Database System, *ACM Trans. Computer Systems 7*, 1 (Feb 1989), 25-60.

[Ree79] D. P. Reed, Implementing Atomic Actions on Decentralized Data, *Proc. 7th ACM Symp. on Operating Systems Principles*, Pacific Grove, CA, Dec. 1979, 66-74.

[Ree83] D. P. Reed, Implementing Atomic Actions on Decentralized Data, *ACM Trans. Comput. Syst. 1*, 1 (Feb 1983), 3-23.

[Reu82] A. Reuter, Concurrency on High-Traffic Data Elements, *Proc. 1st ACM Symp. on Principles of Database Systems*, Los Angeles, March 1982, 83-92.

[Sag86] Y. Sagiv, Concurrent Operations on B*-trees with Overtaking, *J. of Computer and System Sciences 33*, 2 (1986), 275-296.

[ScS83] R. D. Schlichting and F. B. Schneider, Fail-stop processors: An Approach to Designing Fault-tolerant Computing Systems, *ACM Trans. Comput. Syst. 1*, 3 (Aug 1983), 222-238.

[ShG88] D. Shasha and N. Goodman, Concurrent Search Structure Algorithms, *ACM Trans. on Database Systems 13*, 1 (March 1988), 53-90.

[Ske82a] D. Skeen, Crash Recovery in Distributed Database Management System, Ph.D. Thesis, EECS Dept., Univ. of Calif., Berkeley, 1982.

[Ske82b] D. Skeen, Nonblocking Commit Protocol, *Proc. 1st ACM SIGMOD Conf. on Management of Data*, Orlando, FL , June 1982, 133-147.

[Sto79] M. Stonebraker, Concurrency Control and Consistency of Multiple Copies in Distributed INGRES, *IEEE Transaction on Software Engineering SE-3*, 3 (May, 1979), 188-194.

[Tho79] R. H. Thomas, A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases, *ACM Trans. Database Systems 4*, 2 (June 1979), 180-209.

[VLS87] K. Vidyasankar, W. Litwin and Y. Sagiv, Concurrency and Trie Hashing, , 1987.

[Vid87] K. Vidyasankar, Serializability of Nested Transactions, Technical Report #8702, Dept. of Computer Science, Memorial University of Newfoundland, May 1987.

[Wed74] H. Wedekind, On the selection of access paths in a database system, in *Data Base Management*, J. W. Klimbie and K. L. Koffeman (ed.), North Holland, Amsterdam, 1974, 385-397.